Helpin' Red

Table of contents

Helpin'Red

Help, tutorials and examples for the Red programming language

Thrown together by Ungaretti. Still evolving...

Version 1.7 Built: 1/27/2019 6:39 AM

You may download the contents of this website in PDF, MS-Word and Windows Help App formats.
Check the downloads page. There you will also find the **Rededitor**, a fool-proof editor that runs your scripts with just one click.

Czech translation by Tovim.

Tradução para português - Portuguese translation.

Suggestions, corrections and collaborations are most welcome!! Post them at https://gitter.im/red/docs @ungaretti, or send a private message there @ungaretti.

This work is created using HelpNDoc software.

Next topic >

Downloads

File:	Size:	**MD5 Hash** (see below how to check it)
REDEDITOR 1.1 *** Run a Red script by pressing "play" button! **New!** Compile options added!	3432665	FDD67784B883CFADACDD2F9AECB980A5
helpin.red in PDF format	-	-
helpin.red in MS Word format	-	-
helpin.red in Windows Help app (Chm) *	**	**
helpin.red HelpNDoc project	-	-

* Help app (Chm) may raise issues with firewalls and anti-virus softwares! Also, to make it work, you must right-click on the downloaded file, chose *properties* and check "*unblock*".

** It's a pain to change the hash every time I want to update Helpin'Red website, and there are virtually no downloads of this, so I'll update the .chm file, but I won't update the hash anymore. if you want a safe download, contact me at gitter.

*** Rededitor and Makeshift IDE are zip files that contains executables (Notepad++ and Red), so it may also raise issues with firewalls and anti-virus. The hash and size are for the zip archive.

I certainly don't add malware to my files, but who knows what hackers might do, so, just to be sure, I add the size and the MD5 hash of Rededitor. I know MD5 is not the safest hash, but it is small, and along with the size of the file should make you sure enough that the file you're downloading is the same files I created. Hash is not needed for PDF or Word, and I can't add a hash for the HelpNDoc project as it would change the moment I write it down in this page.

To find the size and the MD5 hash of a file, run the Red script below. It opens a GUI file selector, so it is pretty easy to use.

```
Red []
a: request-file
prin "Hash= " print checksum a 'MD5
prin "Size= " print size? a
```

You may even type it at the console:

```
>> b: request-file                              ; the GUI file selector
opens here
== %/C/Users/André/Documents/mytestfile.txt

>> print checksum b 'MD5
#{E054964EFB5ECAA5BF89164B988A62F7}

>> print size? b
2574
```

< Previous topic   Next topic >

Introduction

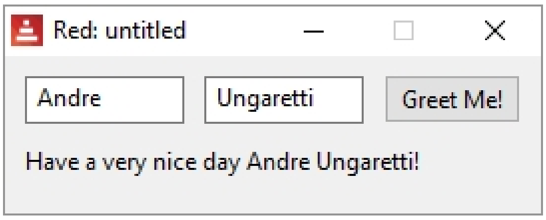
About Red

- Both Red and Red/System are published under the BSD license. The runtime is published under the BSL license.
- Red is a programming language that fits in a single executable file with about 1MB. No install, no setup.
- Red is free and open-source.
- Red is interpreted, but can compile you code and generate single standalone executables.
- Red does some compiling before interpreting, and so turns out to be quite fast.
- Red is simple. Its code is clean and has no bloat at all.
- Red is under development (alpha) as of october 2018, but aims at:
    - being multi-plataform;
    - having cross-platform native GUI system, with a UI dialect and a drawing (graphics) dialect;
    - being a full-stack programming language, that is, from very low to very high level.
- Red is the open-source evolution of Rebol. If you want to try some of the features that are not yet available in Red, you should download Rebol and try it. However, Red is the future.
- Red is being developed by a group of people led by Nenad Rakocevic.
- Recently, Red raised substantial funds from an ICO and a Red Foundation was set up in Paris, France, so it's here to stay.

A taste of Red:

```
Red [needs: view]
view [
    f1: field "First name"
    f2: field "Last name"
    button "Greet Me!" [
        t1/text: rejoin ["Have a very nice day " f1/text " " f2/text
```

```
"!"]
    ]
    return
    t1: text "" 200
]
```



If I got you interested, you should really take a look at Short Red Examples, by Nick Antonaccio.

About this work:

This is an evolution of the Red Language Notebook.
I chose to use HelpNDoc software to develop a more friendly and useful interface.

Notes:

- I use Windows, so this work is based on this Operating System.
- I'm not an experienced or even a good Red programmer, in fact, I'm not a programmer at all.
- English is not my native language.
- This isn't a complete reference for the Red language (yet?).
- I did not use the best coding style in many examples. Please, take a look at Red's coding style guide.
- I try to make my work original, but some text was copied and pasted from Red's official documentation and I based some examples on what I found at:
    - red-by-example.org by Arie van Wingerden and Mike Parr
    - mycode4fun.com.uk by Alan Brack
    - redprogramming.com by Nick Antonaccio

  Also, a lot of help came from the Red community at gitter.im/red/home. Thank You all!!!

- If you can't find something on the existing Red Documents, there is always www.rebol.com.

< Previous topic   Next topic >

Helpin'Red conventions and notations

1- Syntax highlight and scripts

I find that syntax highlight is very helpful for beginners as there are so many predefined words in Red and its code is so concise. Whenever possible I use syntax highlighted code taken from Notepad++[1].

```
Red []
a: "Hello"
b: 123
c: [33 "fox"]
print c
```

[1] - To copy and paste highlighted code from Notepad++ I use a plugin called NppExport.

The console output is represented by a gray background. When examples are given as console-typed commands, I highlight the user-typed input using bold typeset. This can avoid confusion, as sometimes you may want to copy and paste text from the examples, and it may not work as expected.

```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

I also add a line between commands to make it more readable, and sometimes comments and colored highlights. These are added by me during edition, so be careful when copying and pasting.

```
>> a: make hash! [a 33 b 44 c 52]
== make hash! [a 33 b 44 c 52]
                      ;this empty line doesn't exist in the console
>> select a [c]
== 52
                      ;this empty line doesn't exist in the console
>> select a 'c
== 52                 ;comments and highlights are added by me
later, during edition
```

< Previous topic Next topic >

Getting started

The first thing is, of course, to download the Red executable. You may get the latest version from here.

When you execute it (double click), it simply opens the console (a.k.a. REPL) on your desktop.

Instructions on how to run scripts are described at the "Hello world" - run and compile chapter, but first, I think you should choose a text editor.

Choose an editor

You may just write your scripts on any text editor that outputs pure text files, like Notepad, then download the Red executable from Red's website and run them using the command line, but that is not very friendly. There are quite a few options that will make it much easier. Please take a look at Rededitor.

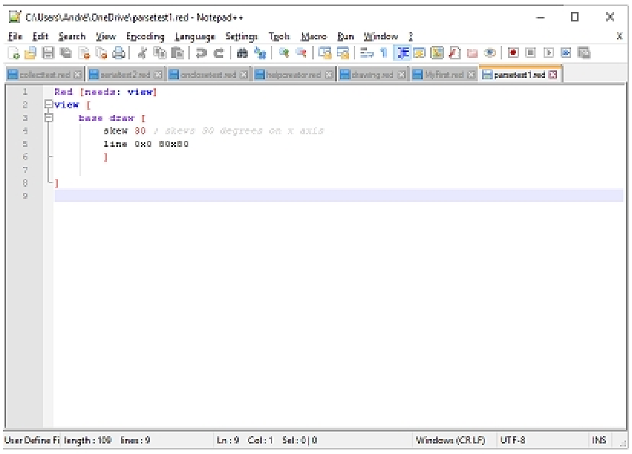
Red's website suggests:

- Visual Studio Code with Red extension .
- Browser-based Cloud9 editor (setup instructions for Red).

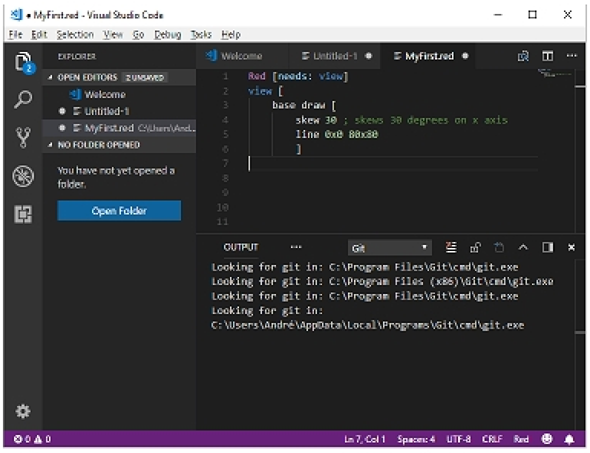
I add Notepad++ to these suggestions, because it's a lightweight, very popular editor. Red prides itself for being a single-file with no install and no setup. Well, if you like that about Red, you will appreciate using Notepad++, specially if configured as Rededitor.

Throughout this work I use Notepad++ (Rededitor).

I also made a chapter about setting up Visual Studio Code. It's a more complete editor for programming, with many features that Notepad++ doesn't have.



Notepad++

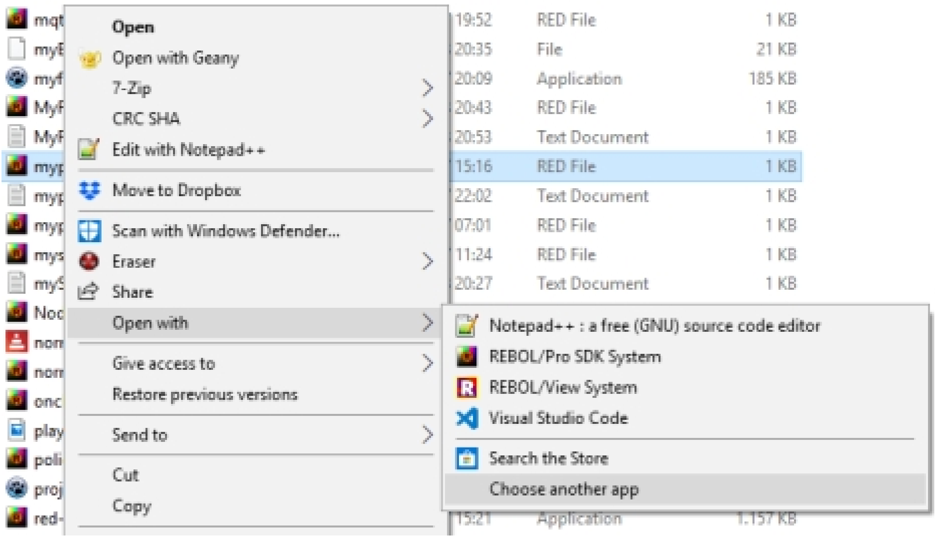


Visual Studio

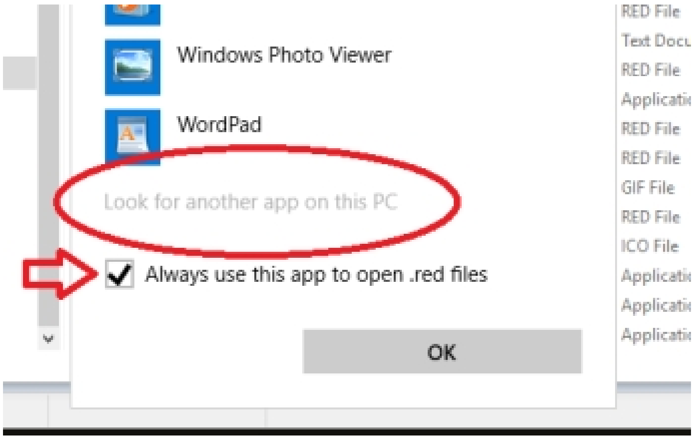
Some information you may find useful:

The first time it runs, Red creates files at `C:\ProgramData\Red\` . If you install a new release or built of Red, I advise you to wipe out the files in that directory, otherwise, unless you specify the path to the new release, Windows will keep using the old release as default.

A Red script is a pure text file. It may have any extension, but its a good idea to give them a .red extension, as later, when you use text editors, you will want them to recognize the language you are using. You will probably also want Windows to associate files with .red extension to the Red executable. The easiest way to do that is to right click on a text file with .red extension and choose "*Open with/Choose another app*":



Then navigate to "*Look for another app on this PC*", check the "*Always use this app to open .red files*" box, click on "*Look for another app on this PC*" and select your Red executable. Every file with extension .red will be associated with the Red executable now.



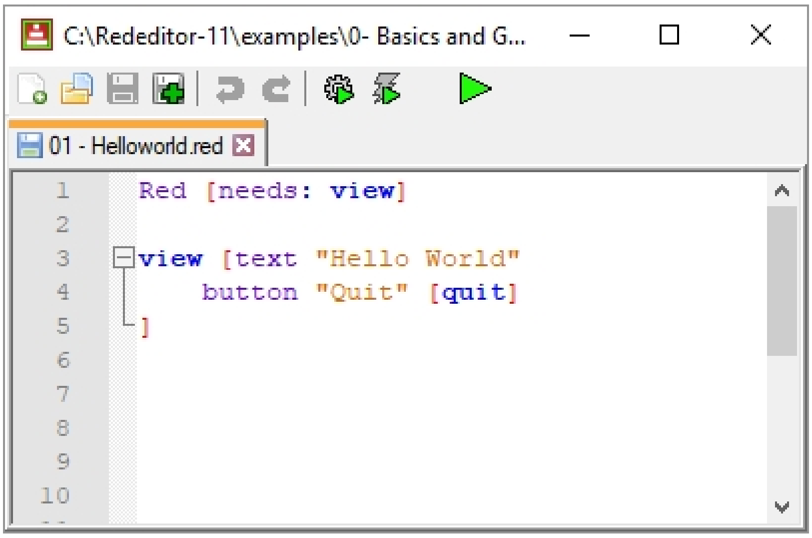
< Previous topic

Next topic >

Rededitor

For Windows, but works surprisingly well in Linux using Wine.

Everything you need to get started with Red, including Red itself!
Just press the play button to run your script! *



*the very first run may take a while as the Red executable compiles the GUI console.

After much trial and error with Notepad++ configuration, I came up with a setup that is clean, lean, and allows you to save & run a Red script by simply pressing a "play" button.

It has all the nice features of Notepad++, plus syntax highlight for Red and the necessary plugins. Everything is packed in a zip file along with a copy of the Red executable. This zip extracts to a folder that is portable and self-sufficient, meaning you can clone it just by coping and pasting.

I called this package, very creatively, Rededitor. You can get it in the Downloads page.

 - **Save and run -** interpreted mode.

 - New

 - Open

 - Save

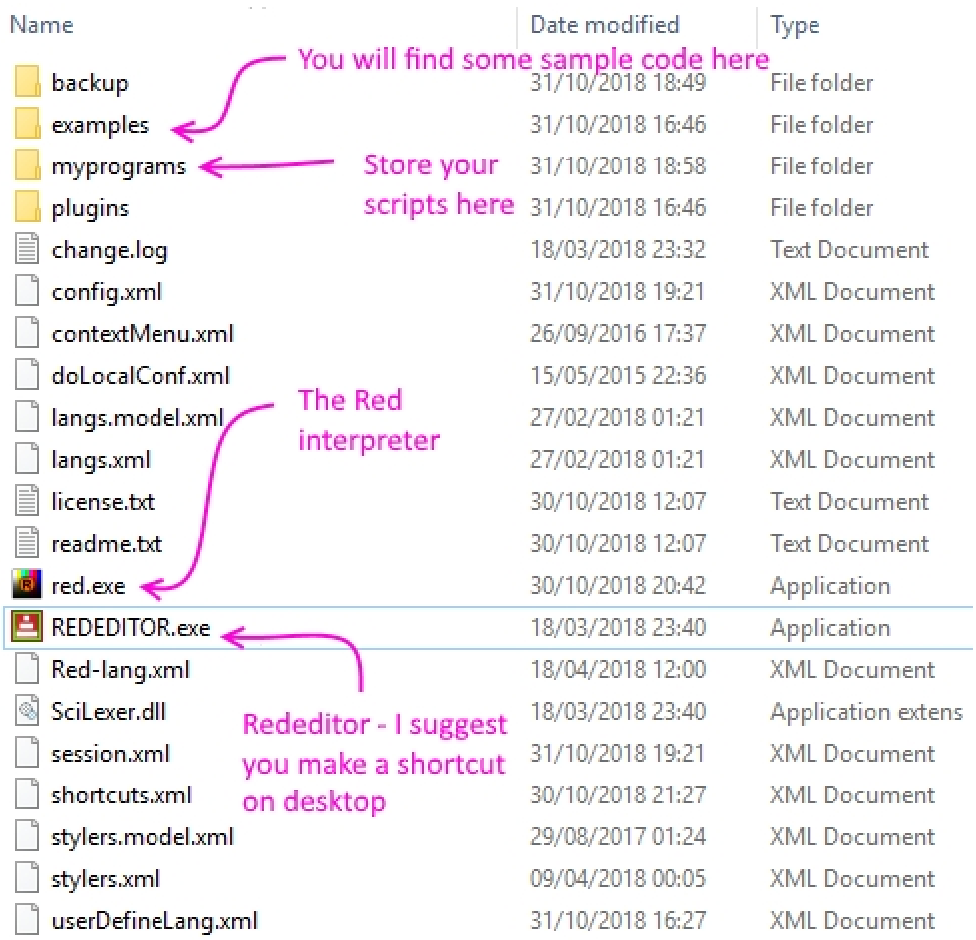
 - Save as...

 - **Development quick compile with DLL -** Saves, compiles and run your script. Compiles with `-c` option (look at [this chapter]), meaning that compilations are fast (except for the very first one as Red takes about a minute to create the DLL and some support files).

 - **Release compile to standalone exe -** Saves, compiles and run your script. Compiles with `-r` option (again, look at [this chapter]), meaning that creates a standalone executable file. Always takes about a minute to compile.

I suggest you tick on `Settings/Preferences.../Hide menu bar` to make it look even cleaner, like the screenshot above. You can toggle the menu bar back by pressing the `alt` key or `F10`. I don't make the hidden menu default because it might be confusing

After downloading of the zip, extract the Rededitor folder. Inside you will have this:

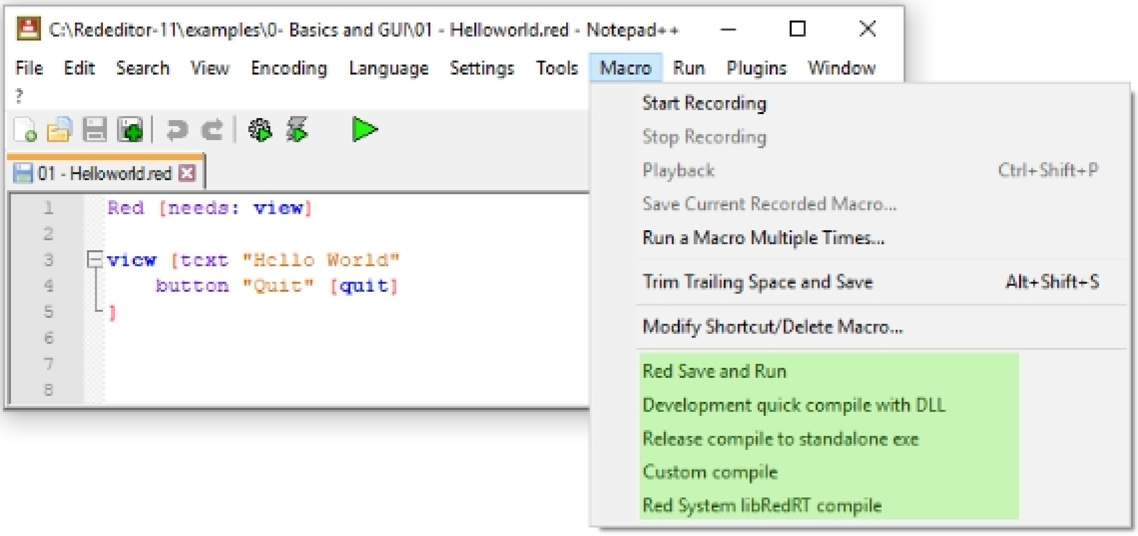


Notes:

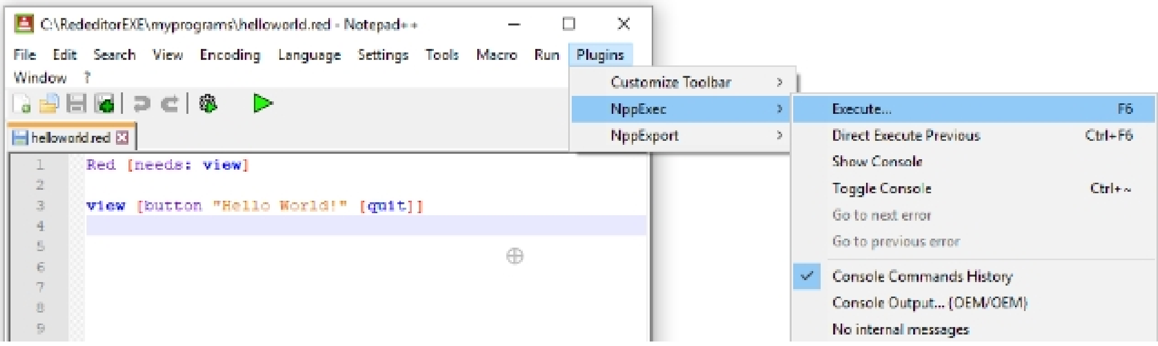
- Remember to regularly update the Red interpreter with the latest Red release, renamed to "**red.exe**".

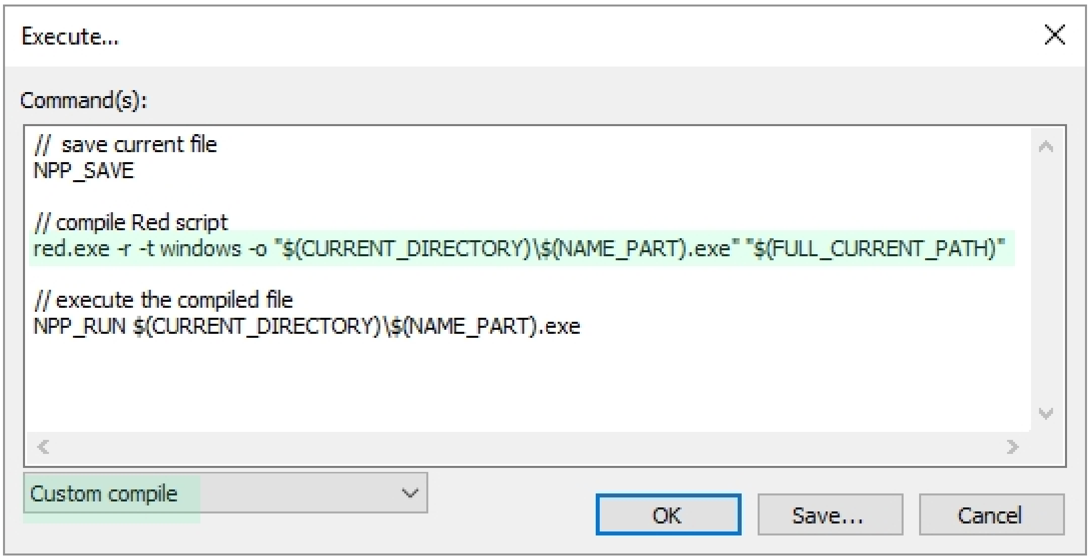
- The compiling features,  and  (not the "**Save and run**") have issues with some characters in the path. They do not work, for example, on my "André" folder, I get: "Cannot access source file: C:\Users\Andr ├® \Documents\Rededitor\myprograms\helloworld.red." So, pay attention where you place your Rededitor if you want it to compile scripts.

- The run and compile features are also available in the **Macro** menu:

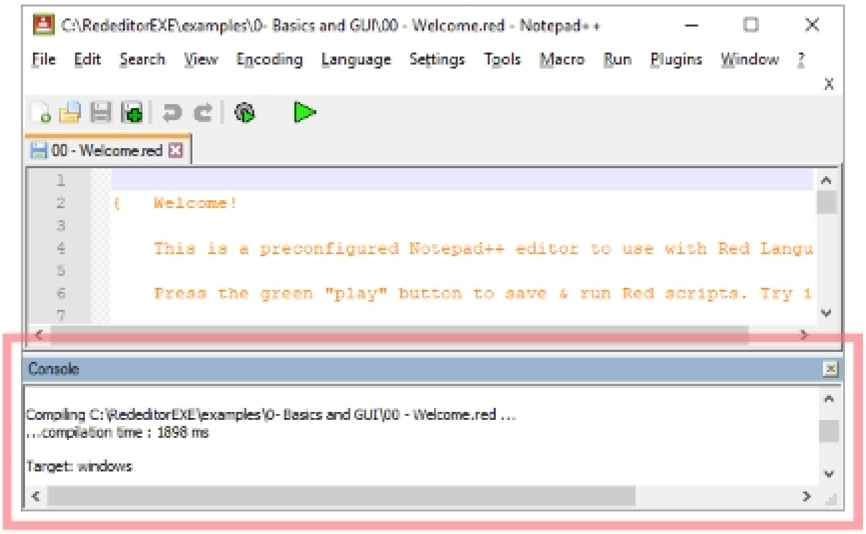


- There you will find a "**Custom compile**". You may change the parameters of this compilation on "*Plugins / NppExec / Execute...*", choosing the "Custom compile" script and editing it.





- There is also the **Red System libRedRT compile** macro. This one uses the `-u` option. I created it to do some tests with [Red Computer Vision library]() by François Jouen.
- When you compile scripts, Rededitor shows a "console" panel. Unfortunately, that is **not** Red's console. It displays Red's **output**, like prints and probes, but it cannot be used for input. This console is disabled in the **Save and run** feature, since the GUI console is shown.



- I'm afraid the examples packed with Rededitor leave a lot to be desired. I can't bring myself to create simple scripts for all topics, and many of them are text-based to be used with console, so don't lend themselves for compilation. Hope to improve that in the future.
- Rededitor License:

  Rededitor is just a pre-configured Notepad++ with 3 plugins: "Customize Toolbar", "NppExec" and "NppExport. Please, refer to Notepad++'s "license.txt" in Rededitor's folder.

  As far as I'm concerned you can do whatever you want with Rededitor as long as you respect Notepad++ license.

The only actual change made to the program itself (Notepad++) was changing its icon.

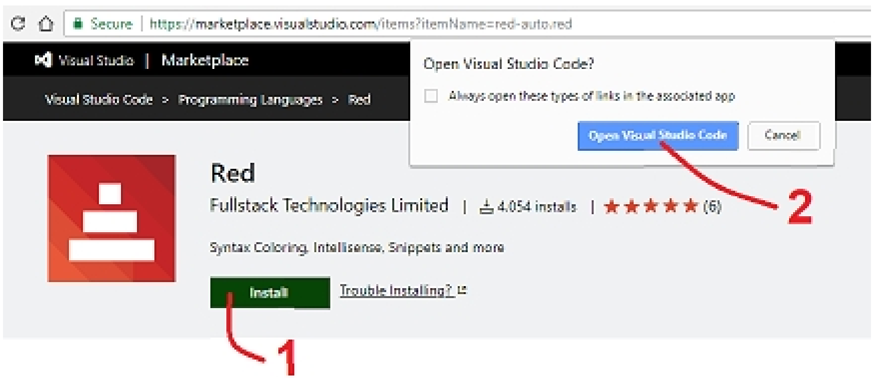
< Previous topic Next topic >

Setup - Visual Studio

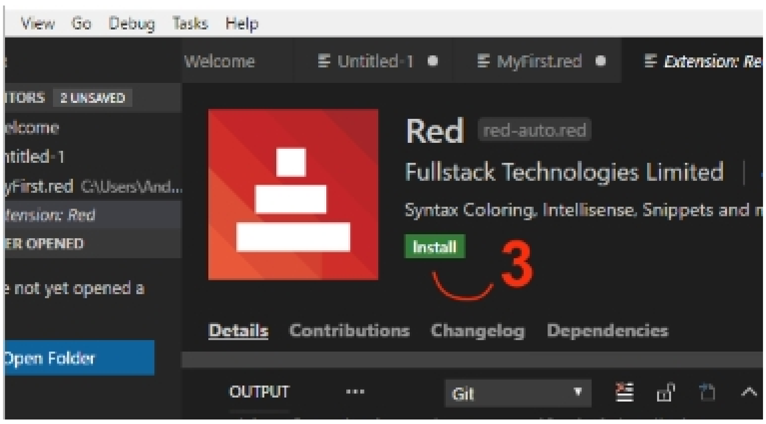
Installing Visual Studio with Red extension seem to me as being very straightforward. First you must run the Red executable, this page says "For Windows user, need to run `red.exe --cli` first"), so, open the command prompt make sure you run the Red executable with the --cli option at least once before installing Visual S. Look here how to do that.

Then download Visual Studio from here and install it like any other software.

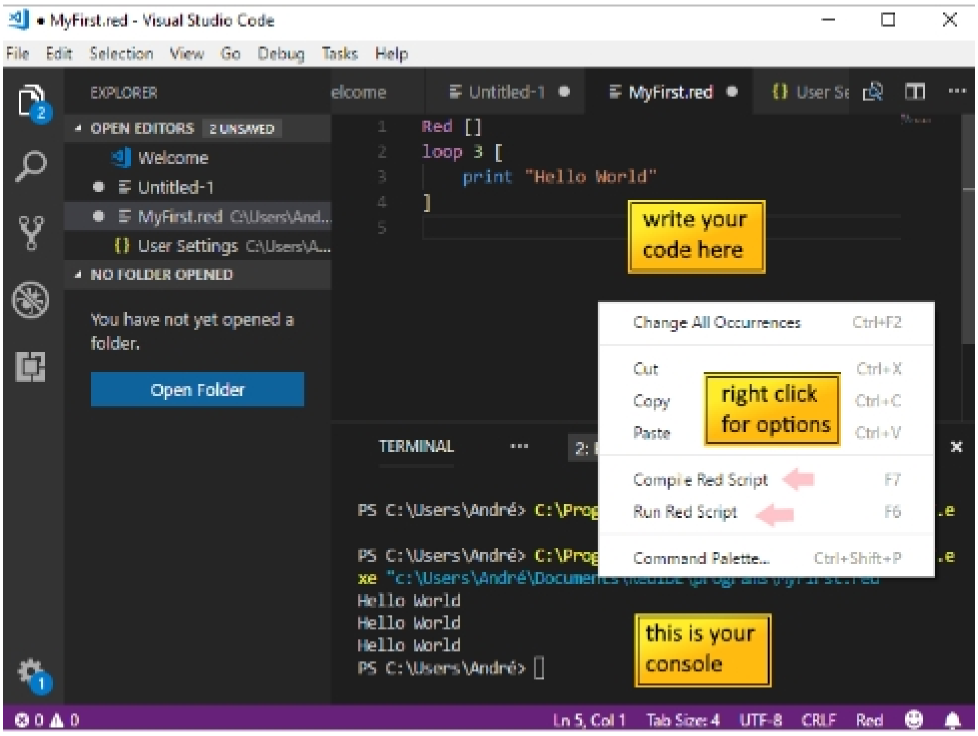
Then open this page (Red extension) and click on Install. You will probably be prompted to "Open Visual Studio Code" . Click on it too:

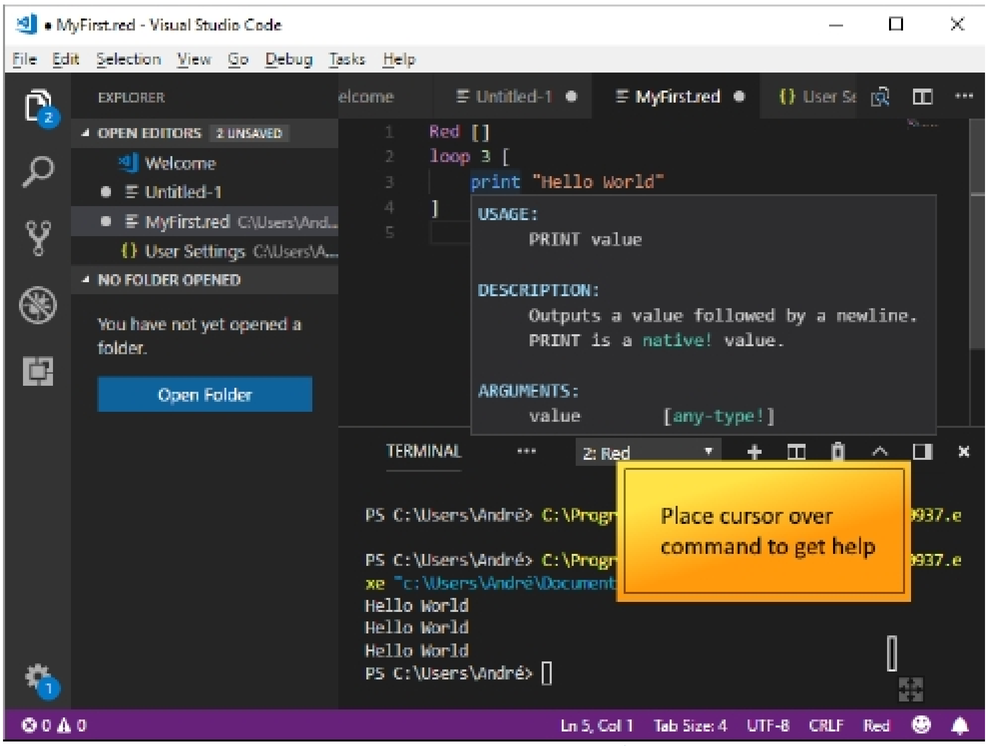


Visual Studio will open with a button to install the Red extension. Click on this install button and... you are done! I had to close Visual Studio and open it again for changes to take effect. Maybe you will need to do that too.



Some basic tips on how to use Visual Studio:





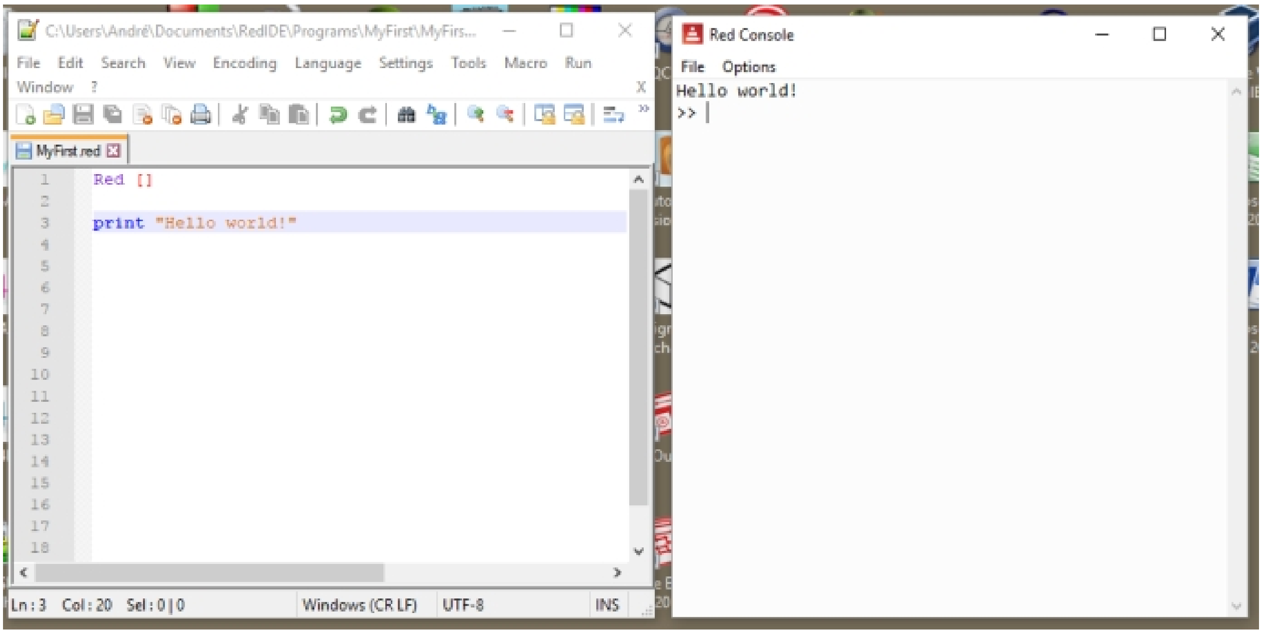
< Previous topic Next topic >

"Hello world" - run and compile

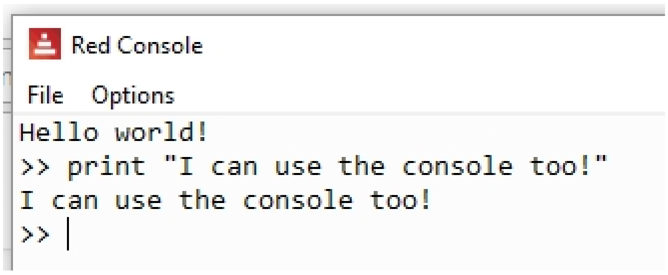
Console "Hello world":

Write the code below on Rededitor, save it as "MyFirst.red" in the "myprograms" folder and execute it.

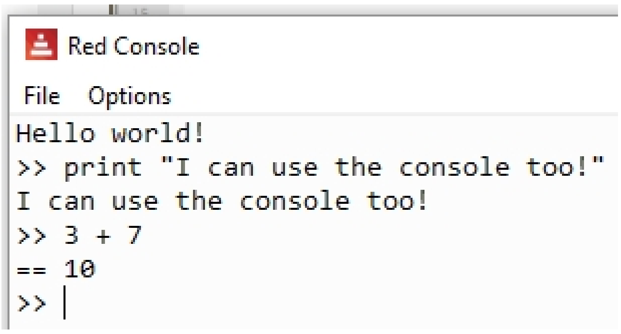
You should have:



The window on the right is the console, sometimes called REPL. Click there, type `print "I can use the console too!"` and press enter:



Now type 3 + 7 and press enter:



Notice that you must have a space between words. Spaces are the delimiters and without them you get errors:

```
Hello world!
>> print "I can use the console too!"
I can use the console too!

>> 3 + 7
== 10

>> 3+7          ;no spaces!!!!!
*** Syntax Error: invalid integer! at "3+7"
*** Where: do
*** Stack: load
```

Notice that after 3+7 I wrote `;no spaces!!!!!` . Red ignores words that come after a semi-colon, that's one way to make comments to your code.

Back to the program (aka. *script*):

Interpreted programming languages execute one line of code at a time. Programs for interpreted languages are called "scripts". Red is not really interpreted, as it does some compiling before running (sort of), but its programs are generally called scripts anyway.

On the first line we have `Red [ ]`. As we mentioned before, every Red script must start with `Red`, Not RED nor red, but `Red`. Following `Red` we have brackets. In Red, anything inside brackets is called a "block". This first block is intended to contain information about your program. This information is mostly optional with a few exceptions, the most relevant being the declaration of libraries (more on that in a while).

A nice first block would be:

```
Red [
      title: "Hello World"
      author: "My name"
      version: 1.1
      purpose {
            To print a greeting to the planet.
            Notice that multi-line text goes
```

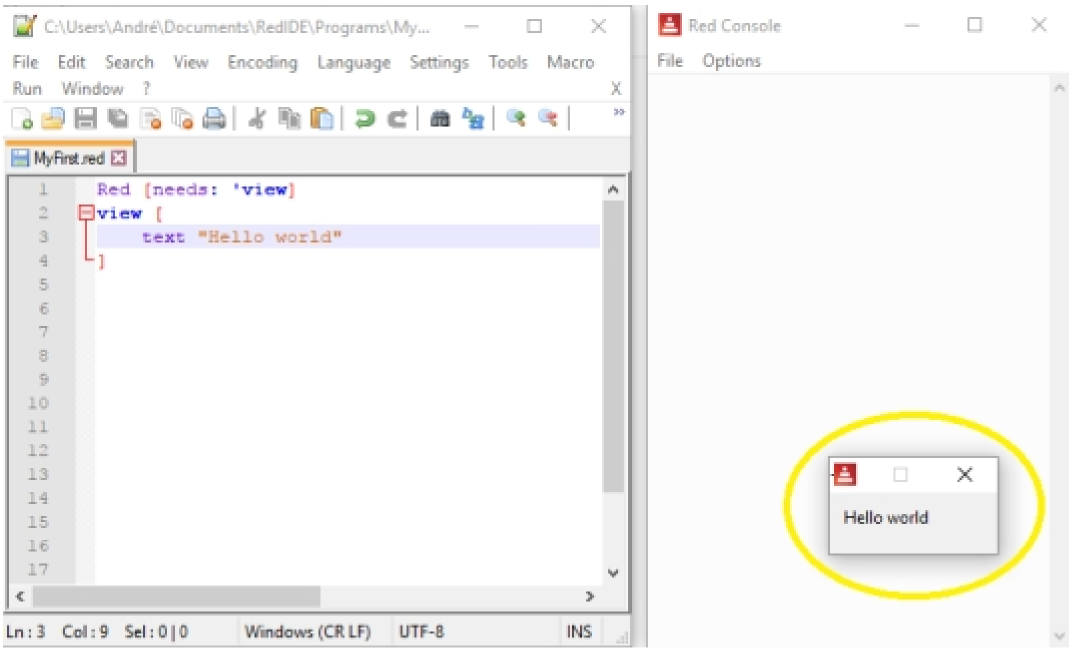
```
                inside curly brackets.
                }
          ]

    print "Hello World!"
```

Anything before the `Red[ ]` is ignored by the interpreter:

```
Lots of things may be written here.
The interpreter only considers what is
written after the...

Red [ ]
print "It works anyway!"
```

"Hello world" with graphic user interface - GUI:

One of the most striking features of Red is it's easy-to-use graphic interface. It makes a very clever use of the Operating System's own graphic APIs. A simple "Hello world" with GUI would be:



Notice I wrote `needs: 'view` in the header block (apostrophe is optional). That tells Red to load the "view" graphic library. This is not necessary if you are using the GUI console, as the "view" library is already loaded, but I think it's a good idea to include it anyway.

Compiling your "Hello world" to an executable file:

To compile your script, you must execute Red followed by one or more options and the name of the script. The most common options are `-c` and `-r`.

`-c` creates an executable, but also creates a DLL and some other support files. That executable is not standalone, it must have the DLL in the same folder to run. The main advantage of using `-c` is that, once the DLL and support files are created (may take a minute or two), the subsequent compilations are quite fast. That means you may change the script and quickly recompile it.

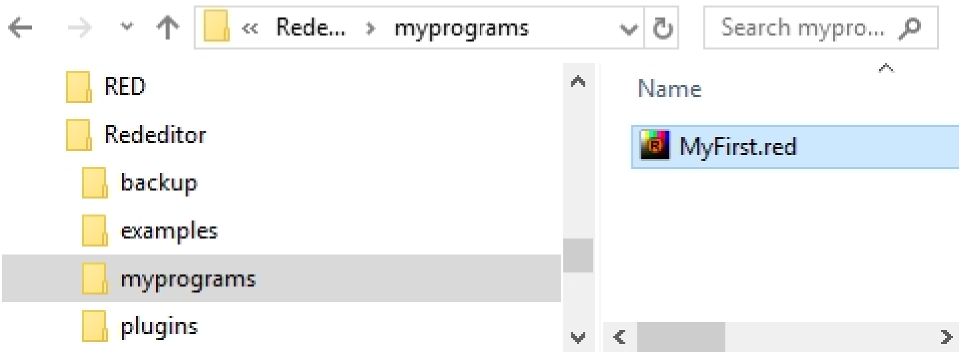
`-r` , on the other hand, creates a standalone executable, but it does the full compilation every time, so it takes longer to recompile if you change your script.

On **Rededitor**, you already have macros that save, compile and run your script. You may use the  - **Development quick compile with DLL** (uses `-c` option) or the  - **Release compile to standalone exe** (uses `-r` option).

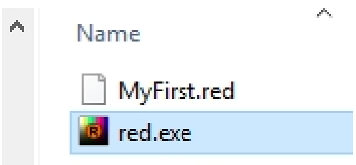
CLI compiling:

You can create an executable from your GUI "Hello World".

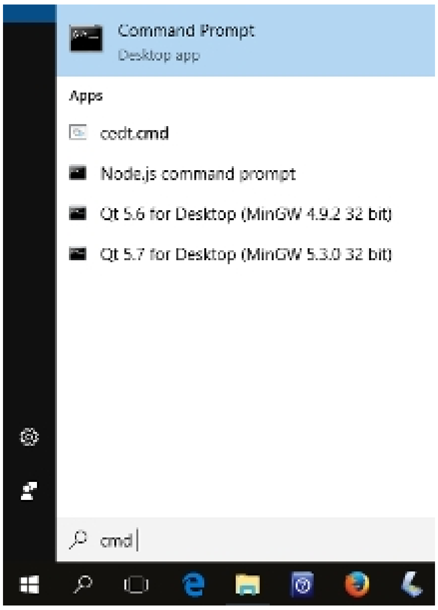
If you saved the GUI program above as "MyFirst.red" in the "myprograms" folder of Rededitor, you should have something like this in your computer:



For the sake of clarity, make a copy of your Red executable and paste it in the same folder as your program, otherwise the results of the compilation will be in the Rededitor folder, lost among all those files.



Open the Command Prompt window. If you don't know how, write "cmd" in Window's search field and click on the Command Prompt icon:

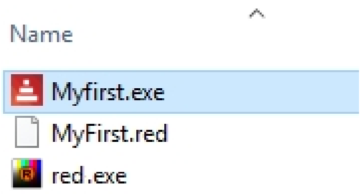


In the Command prompt, type the path to your Red executable (the executable you just copied in the "myprograms" folder), followed by `-r -t windows` and the name of your program:

```
C:\Users\André\Documents\Rededitor\myprograms> red.exe -r -t windows Myfirst.red
```

Note: If you compile to windows, i believe you must always load the GUI library (use `needs: view`). If you just want a program that runs on CLI alone, you may use MSDOS (default) as target.

Red will give you a series of messages in the Command Prompt and, after about a minute you will have the standalone executable in your "programs" folder:



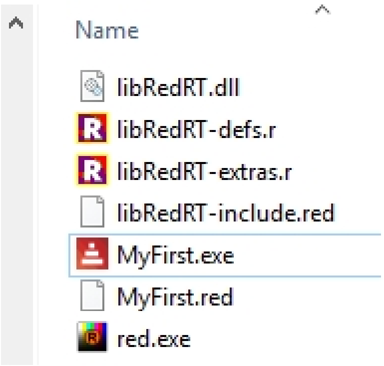
Double click on it and you will have your GUI "Hello World" message on your screen.

The `-t windows` is not really needed, as the default (MSDOS) will give you very similar results. Try both.

You could compile the MyFirst.red program using only the `-c` (compile) option:

```
C:\Users\André\Documents\Rededitor\myprograms> red.exe -c Myfirst.red
```

You will then have the following files in your "myprograms" folder:



The only two files you need to run your program are the **libRedRT.dll** and your program's executable, in this case **MyFirst.exe**.

However, when your run your executable, you will notice that Red keeps a very annoying Command Prompt window open as the program runs. If you want to avoid this use the target option `-t`. The option -t compiles it to a specific platform.

```
C:\Users\André\Documents\Rededitor\myprograms> red.exe -c -t windows Myfirst.red
```

This will result in those same extra files, including the DLL, but it won't open the Command Prompt while your program runs.

Extra notes on compiling:

[Red Wiki about issues]

I found that the compiled version of a program may not behave as the interpreted one. I had problems with "print" statements I left for debugging, so I guess calling console commands in executable mode is not ok. I also had problems with global variables (words) inside functions, the compiler does not seem to recognize them as global variables. I solved this last problem in two different ways:

1. I "declared" my variables, that is: I assigned values to the variables (words) in the beginning of my program. The values are not important, as they change later.

2. I used the "-e" compiler option (see in "Compiler options" below).

You should be able to compile to the platforms listed below but, as of this writing, Red is still evolving, and you may find some issues (e.g. compiling to android does not seem to work yet).

From Red's [github page]:

Cross-compilation targets:

```
MSDOS         : Windows, x86, console (+ GUI) applications
Windows       : Windows, x86, GUI applications
```

```
WindowsXP    : Windows, x86, GUI applications, no touch API
Linux        : GNU/Linux, x86
Linux-ARM    : GNU/Linux, ARMv5, armel (soft-float)
RPi          : GNU/Linux, ARMv5, armhf (hard-float)
Darwin       : macOS Intel, console-only applications
macOS        : macOS Intel, applications bundles
Syllable     : Syllable OS, x86
FreeBSD      : FreeBSD, x86
Android      : Android, ARMv5
Android-x86  : Android, x86
```

Compiler options:

```
-c, --compile                  : Generate an executable in the working
                                 folder, using libRedRT. (developement
mode)
-d, --debug, --debug-stabs     : Compile source file in debug mode. STABS
                                 is supported for Linux targets.
-dlib, --dynamic-lib           : Generate a shared library from the source
                                 file.
-e, --encap                    : Compile in encap mode, so code is
interpreted

                                 at runtime. Avoids compiler issues.
Required

                                 for some dynamic code.
-h, --help                     : Output this help text.
-o <file>, --output <file>     : Specify a non-default [path/][name] for
                                 the generated binary file.
-r, --release                  : Compile in release mode, linking
everything

                                 together (default: development mode).
-s, --show-expanded            : Output result of Red source code
expansion by

                                 the preprocessor.
-t <ID>, --target <ID>         : Cross-compile to a different platform
                                 target than the current one (see targets
                                 table below).
-u, --update-libRedRT          : Rebuild libRedRT and compile the input
script

                                  (only for Red scripts with R/S code).
-v <level>, --verbose <level>  : Set compilation verbosity level, 1-3 for
                                 Red, 4-11 for Red/System.
-V, --version                  : Output Red's executable version in x.y.z
                                 format.
--config [...]                 : Provides compilation settings as a block
                                 of `name: value` pairs.
--cli                          : Run the command-line REPL instead of the
                                 graphical console.
```

```
--no-runtime                   : Do not include runtime during Red/System
                                 source compilation.
--red-only                     : Stop just after Red-level compilation.
                                 Use higher verbose level to see compiler
                                 output. (internal debugging purpose)
```

There is also `-e` option. See"Extra notes on compiling" above.

Running Red on system's console:

To run Red on system's console, open cmd prompt, change directory to the folder where you have your Red executable and type its name followed by `--cli` . Note it's two dashes. I have `red-063.exe`, so:

```
 C:\Users\André\Documents\RedIDE>red-063.exe --cli
--== Red 0.6.3 ==--
Type HELP for starting information.
>>
```

Passing arguments to a Red script:

Everything on the command line that follows the script file name is passed to the script as its argument. Those arguments are stored on system/options/args as a block.

This script was saved as "arguments.red":

```
Red []
probe system/options/args
```

Executed from CLI:

```
C:\Users\André\Documents\RedIDE\programs>red-063.exe arguments.red foo
boo loo
```

Output of script on Red's console is:

```
["foo" "boo" "loo"]
>>
```

< Previous topic   Next topic >

Built-in help

Red has an exceptional built-in help. There is a large amount of information you can get about the language and about your own code just typing a few commands on the console.

function! ? (or help) Red-by-example

Gives information about all of Red's reserved words and also about your own code. You may also type `help`, but `?` is, of course, shorter. `?` by itself prints information about how to use help.

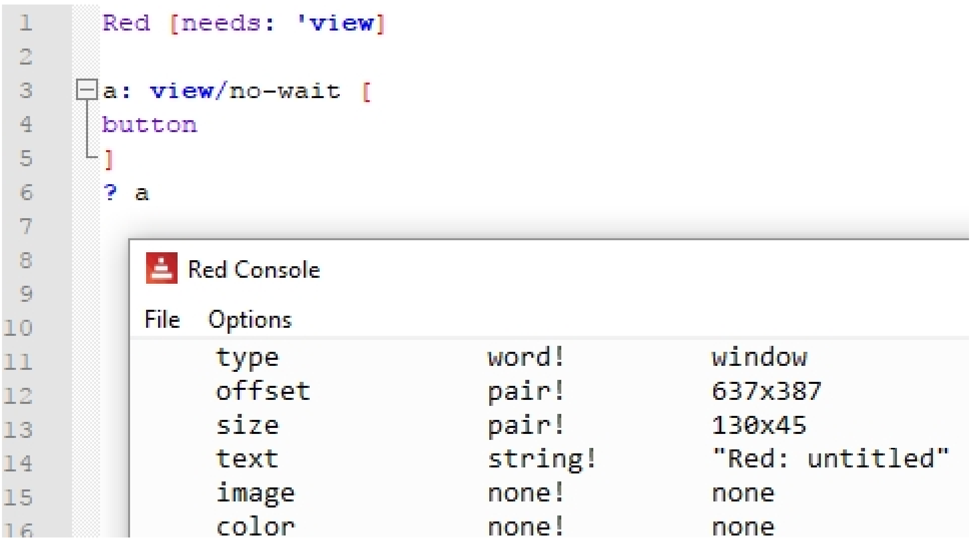
```
>> ? now
USAGE:
     NOW

DESCRIPTION:
     Returns date and time.
     NOW is a native! value.

REFINEMENTS:
     /year        => Returns year only.
     /month       => Returns month only.
     /day         => Returns day of the month only.
     /time        => Returns time only.
     /zone        => Returns time zone offset from UCT (GMT) only.
     /date        => Returns date only.
     /weekday     => Returns day of the week as integer (Monday is day
1).
     /yearday     => Returns day of the year (Julian).
     /precise     => High precision time.
     /utc         => Universal time (no zone).

RETURNS:
     [date! time! integer!]
```

```
>> a: [1 2 3]
== [1 2 3]

>> help a
A is a block! value: [1 2 3]
```

```
>> help block!
    a                 length: 3  [1 2 3]
    cancel-captions   length: 3  ["cancel" "delete" "remove"]
```

```
>> a: function [a b] [a + b]
== func [a b][a + b]

>> ? a
USAGE:
    A a b
DESCRIPTION:
    A is a function! value.
ARGUMENTS:
    a
    b
```

You can get information about complex objects:



If you don't know exactly what you are looking for, "?" will perform a search for you:

```
>> ? -to
     hex-to-rgb      function!     Converts a color in hex format to a 
tuple value; returns NONE if it f...
     link-sub-to-parent function!     [face [object!] type [word!] old 
new /local parent]
     link-tabs-to-parent function!     [face [object!] /init /local 
faces visible?]
```

You can find all defined words of a given datatype!

```
>> ? tuple!
    Red              255.0.0
    white            255.255.255
    transparent      0.0.0.255
```

```
black              0.0.0
gray               128.128.128
; ... the list is too long!
```

function! what [Red-by-example]

Prints a list of globally-defined functions. Try it!

function! source [Red-by-example]

Shows the source code for a mezzanine function or a user created function.

Try `source replace` .

mezzanine functions

Red interpreter has:

- the native functions which are "embedded" in the interpreter and are executed at a very low level;
- and mezzanine functions which, even though they are part of Red interpreter (come in the Red executable) are created using Red, that is, they have a source code you can read using `source`.

function! about [Red-by-example]

Display version number and build date.

< Previous topic   Next topic >

Notes on syntax

- Red is case insensitive, but there are few exceptions, the most relevant is that a program must begin with **Re**d (not REd or red).
- `new-line` characters are mostly ignored by Red interpreter. A relevant exception is a `new-line` inside a string.
- Red is a functional language, meaning that it evaluates results. The evaluation order is not usual and you may be interested in looking at the Evaluation chapter.

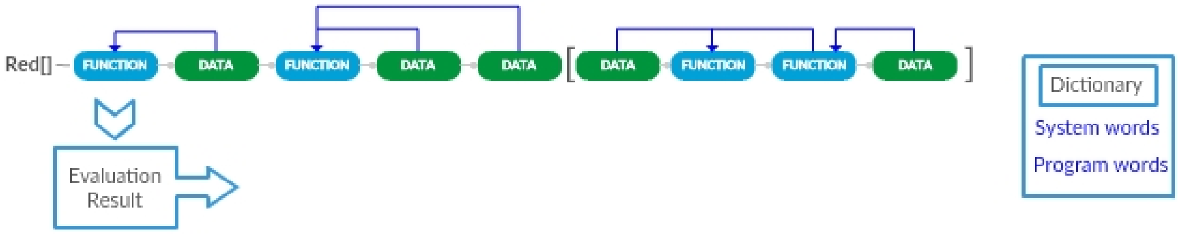
(the following topics may prove to be inacurate, but so far they have explained Red behavior pretty well)

- A Red program is a long chain of "words". Basically, these words may be either "data" or "actions".
- "words" are separated by one or more whitespaces .
- Red keeps a dictionary with predefined words (built-in functions) and user-created words.
- "words" may be grouped into "blocks" by enclosing them with brackets. "Blocks" are not necessarily routines, they are just a group of words that may, or may not, be evaluated by an "action".
- all the program data is inside the program itself. If external data is required, it is added to the program's chain of "words".
- every word must have a value while evaluated. This value may come from:
    - the word itself, if it is data;
    - evaluation, if the word is an action;
    - another word or block. This is achieved by adding a colon after the word, with no spaces, followed by the data or block we want to associate it with (e.g. `myRoom: 33`).
- I find that in Red, you may say that **the variable is assigned to the data, and not the other way around**. In fact, **there are no "variables" in Red, just words that get assigned to data**.
- Copying words (variables) in Red may be tricky. When you want truly independent copies , you should use the word `copy` to . See Copying chapter.

- As with copying, clearing a series (notice that all strings are series) is also tricky. Simply assigning "" (empty string) or zero to it may not produce the expected results. Red's logic makes it seem to "remember" things in unexpected ways. So to clear a series you should use the built-in function `clear`.

- every word has a datatype. Red has a remarkably large number of datatypes. They are listed in the Datatypes chapter.

Somewhat simplified view of Red's flow:



Note: The function that picks data from before it (the third from right to left) refers to infix operators like "+", "-" , "*" , "/" etc.

Refinements

Many actions in Red allow "refinements". A refinement is declared adding "/<refinement>" to the command (built-in function) and it modifies its behavior.

Commenting your code:

All text after a semi-colon (;) in a line is ignored by the interpreter. There is also the built-in function `comment` . A group of words after comment will also be ignored by the interpreter. This group of words must be enclosed by `" "`, `{ }` or by `[ ]` .

I also note that any text written in the source code before the Red "prologue" (Red [...]) at the beginning is also ignored by the interpreter, but I'm not sure this is a safe way to add information to your code.

Examples of comments:

```
It seems to me that text written here(before the prologue)
is ignored by the interpreter. It may (or may not) be a good
way to add information about your script.

Red [                                  ; Here the prologue begins.
  Author: "Ungaretti"                  ; You may add comments after a ";"
  Date: "september 2018"               ; but one-line only.
  Purpose: "to show how to comment the code"
]

; A good prologue should be informative

comment [ This is a multiline comment
within brackets. Not only it doesn't look good
```

```
but it may cause errors - if you add a comma here for example]

print "End of first comment."

comment " This is a comment."       ; if you use quotes, comments are
                                    ; limited to one line.

print "End of second comment."

comment { This is the best way to write
a multi-line comment using "comment" word}

print "End of third comment."

{bizarrely, the interpreter seems to ignore text
written within curly braces even without the use of
the "comment" keyword". This looks elegant to me,
but be careful!}

print "End of the fourth, strange, comment."
```

```
End of first comment.
End of second comment.
End of third comment.
End of the fourth, strange, comment.
```

< Previous topic Next topic >

Using words

Since a Red program is a series of words, its a good idea to take a closer look at them.

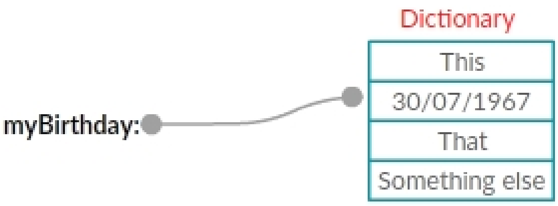
word

A word by itself (not data) does not mean much to Red. Every word must have a value associated to it while evaluated. This value may come from the evaluation of an expression or from the "dictionary". In this later case, it may be data or an action.

```
>> myBirthday
*** Script Error: myBirthday has no value
```

word:

The colon after a word associates it with something in the dictionary. It is the classic "assignment" of other programming languages. By the way, this word-colon group (e.g. "myword:") is a `set-word!` datatype.



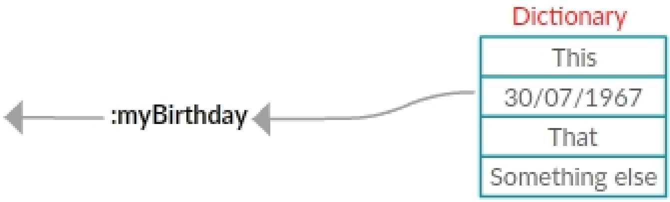

```
>> myBirthday: 30/07/1963
== 30-Jul-1963
>> print myBirthday
30-Jul-1963
```

Words may be associated with code (action) too:

```
>> a: [print "hello"]
== [print "hello"]
>> do a
hello
```

:word

The colon before a word makes it return whatever is associated with it in the dictionary without any evaluation. Values and actions are returned "as is". By the way, this is a `get-word!` datatype.

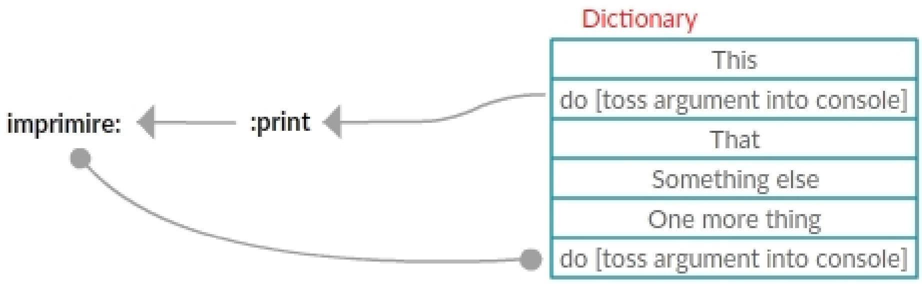



```
>> myBirthday: 30/07/1963
== 30-Jul-1963
>> partyDay: :myBirthday
== 30-Jul-1963
>> print partyDay
30-Jul-1963
```

If a word is associated with an action, a colon before it makes it return the whole code of this action. This creates an interesting situation if you use it with Red's built-in functions:

```
>> imprimire: :print
== make native! [[
    "Outputs a value followed by a newline"
    ...
>> imprimire "hello"
hello
```

What happened above is that "imprimire" now has the same functionality as `print` . Something like this:



Important notes:

- the `:word` syntax is also used a way to access data in a series, as described in the [Blocks & Series chapter;](#)
- if you redefine built-in functions in Red, you may cause a crash, not because of the change itself, but because all the internal functions that rely on the original meaning of that word may not work properly.

'word

Returns the word itself, that is: just a group of letters (but not a string! Just a symbol). This is a `lit-word!` datatype.

```
>> print something
*** Script Error: something has no value
*** Where: print
*** Stack:

>> print 'something
something
```

```
>> type? :print
== native!
>> type? 'print
== word!
```

/word

The slash before a word turns it into a refinement. Obviously, this is a `refinement!` datatype.

native! set

Assigns a value to a word. It seems to me as being the same as the colon after the word...

```
>> set 'test 33
== 33
```

...except that you may set many words at once:

```
>> set [a b c] 10
== 10
>> b
== 10
```

native! unset

A previously defined word can be unset at any time using unset:

```
>> set 'test "hello"
```

```
== "hello"
>> print test
hello
>> unset 'test
>> print test
*** Script Error: test has no value
```

< Previous topic Next topic >

Evaluation

There is a good description of Rebol's evaluation here. It's pretty much the same for Red. I'll not repeat that explanation, instead, I'll describe how I see Red's evaluation from my personal point of view. Again, this may prove inaccurate, but so far it explains Red's behavior pretty well.

Red, the furious evaluator

Once triggered, Red will start reading a text from left to right (→) executing all operations it can find. If it recognizes an operation that requires arguments, it picks the arguments from this main text as needed to come to a final value. Take a look at the concept of evaluable groups and argument picking. Red considers text (strings) as a block of characters, so this main text of Red code is just a big block for Red, even without brackets or quotes.

What triggers Red's fury?

Red is triggered by the "command" `do`. You don't always have to actually type `do`, when you run a script or press `enter` at the console, what is happening is that you are applying an implicit `do` to the text ahead. In the case of a script, the evaluation only begins after the interpreter finds the characters "`Red [`"

An interesting consequence of all this is that, although it's not generally considered good practice, you can actually execute text:

```
>> do "3 + 5"
== 8

>> 3 + 5 ;same thing. The "do" is implicit and input is text (but not a string! datatype).
== 8
```

If it's an evaluation, what is the result?

The result of a Red interpretation is the resulting value of the last evaluable group. Of course you can do all sorts of interesting things along the way, as writing files, reading web pages and creating beautiful drawings on your screen, but the value returned by Red (if there is one) is this last result.

```
>> do "3 + 5 7 * 8 print 69"
69
```

What halts Red's fury?

The end of the text (code) and comments, of course.

But also, Red's evaluation skips blocks inside the main text (blocks within the main block), just leaves them as they are. It only evaluates them if they are an argument of an operation, noting that this operation may be another `do`:

```
>> do {print "hello"  7 + 9  [8 + 2]} ; the last result is the unevaluated block
hello
== [8 + 2]

>> do {print "hello"  7 + 9  print [8 + 2]}
hello
10

>> do {print "hello"  7 + 9  do[8 + 2]}
hello
== 10
```

You will find out that, to develop Red scripts, sometimes you need the resulting values of all evaluable groups in a block, not just the last one. You can achieve that with [reduce](). It returns a block with all the results. However, It' **not** as if you applied a `do` to each evaluable group inside the block, as you can see here:

```
>> reduce [3 + 5 7 * 8 print 69]
69
== [8 56 unset]

>> reduce [3 + 5 7 * 8 "print 69"] ; do "print 69" should print 69!
== [8 56 "print 69"]
```

Math evaluation order

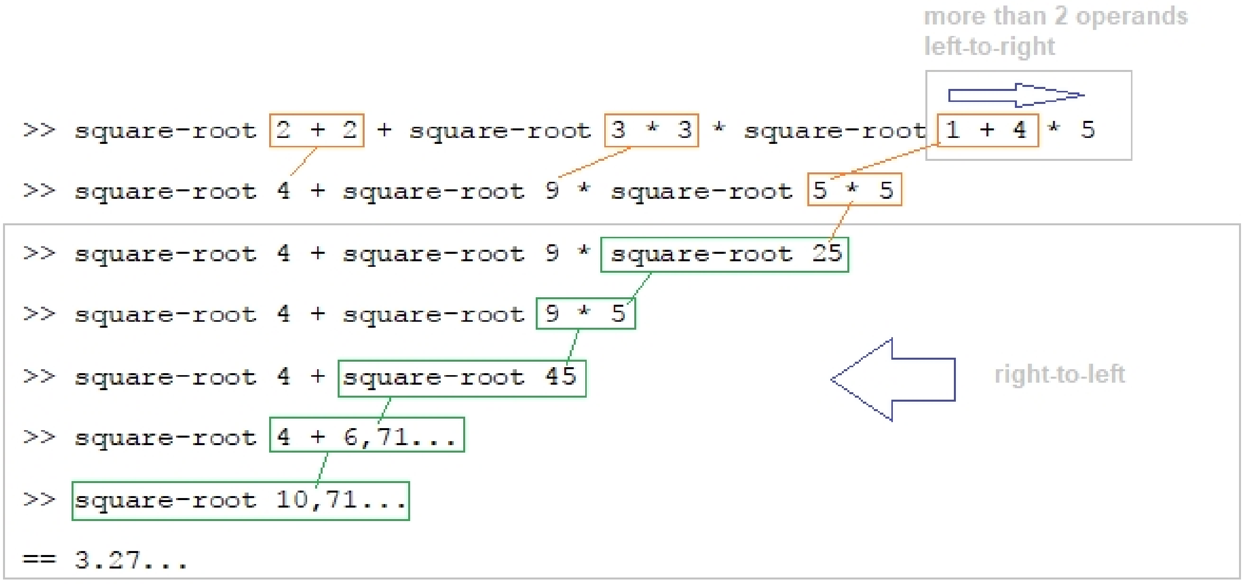
I'm still looking for a simple rule to explain Red's math evaluation sequence. For the moment, I have two favorite candidates. The first is very straightforward and easy to use. The second is not very practical, but gives a view of how (I think) the Red interpreter "thinks", and so I believe it is a good idea to take a look at it to grasp some concepts that may be useful.

1) My favorite rule for the moment:

1- All operations with [infix operators]() that have only values (not functions) as operands are evaluated first. If these infix expressions have more than two operands they are evaluated from left to right (→) with no precedence (i.e., multiplication doesn't automatically get computed before addition) .

2- Then the whole expression is evaluated from right to left (←).

```
>> square-root 2 + 2 + square-root 3 * 3 * square-root 1 + 4 * 5
== 3.272339214155429
```



2) My second favorite, the 3 concepts explanation:

This seems to work and I think that's somehow what the interpreter does.

It's not a simple rule and I think it may not be formally accurate, as I'm not sure that every infix operator has an exact correspondent function operation.

Concept 1: Left to right always →

In Red, things are evaluated from left to right. There is no "order of precedence" as in other languages (i.e., multiplication doesn't automatically get computed before addition). However, you may enclose the functions in parentheses to force precedence.

```
>> 2 + 3 * 5
== 25 ; not 17!
```

Not only expressions, but the whole code of a program is evaluated from left to right.

Infix operators

"+", "-", "*", "/" are called infix operators. They correspond to the functions `add` , `multiply`, `divide` and `subtract`, which need two arguments. So:

`3 + 2` is the same as `add 3 2`

`5 * 8` is the same as `multiply 5 8` ...

...and so on.

`2 + 3 * 5` is just a more readable form of `multiply add 2 3 5` . Red's interpreter does the conversion for you.

Concept 2: Evaluable groups.

When you have a chunk of code, there are groups of words that are evaluable, that is, can be reduced to basic datatypes. For example `[square-root 16 8 + 2 8 / 2 77]` is actually made of 4 evaluable groups: `square-root 16` ; `8 + 2` ; `8 / 2` and `77`. You can use `reduce` to "see" the values of evaluable groups:

```
>> a: [ square-root 16 8 + 2 8 / 2 77]
```


```
>> reduce a
== [4.0 10 4 77]
```

Concept 3: Functions pick their arguments from the evaluable groups

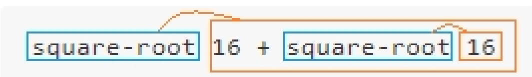
A function takes its arguments from the evaluable groups ahead of it, from left to right (think of infix operators as syntax sugar for their function counterparts). A function that needs 1 argument, take the next evaluable group; a function that needs 2 arguments, take the next 2 evaluable groups, and so on. Notice that a function may use an evaluable group that has another function in it. In this case, it holds its evaluation until the argument function is evaluated, and then use the result.

Again, no precedence rules, just left to right.

A consequence of that is that an expression like this...

```
square-root 16 + square-root 16
```

...is **not** 8, as many would expect, but 4.47213595499958, because what Red sees is:



(or even: `square-root add 16 square-root 16`)

That is: One function that has one argument and **one** evaluable group (which happens to have a function in it).

To obtain that intuitive 8, one must use parentheses:

```
>> (square-root 16) + square-root 16
== 8.0
```

Another example, mixing an infix operator and its corresponding function:

```
>> reduce [add 8 + 2 * 3 8 / 2 divide 16 / 2 2 * 2]
== [34 2]
```

```
>> reduce [add 8 + 2 * 3 8 / 2 divide 16 / 2 2 * 2]
== [34 2]
```

Other explanations:

These are some other "rules" I have seen discussed:

#1

"Left-to-right and operators take precedence over functions and if an infix operator sees a function as its second operand, evaluates it"

#2

"In general, expressions are evaluated from left to right; however, within each expression evaluation occurs from right to left".

#3

"Each expression takes as many arguments as it should, each argument in turn may be another expression and Red will parse the expressions until they all have a full set of arguments".

< Previous topic Next topic >

Some pitfalls of Red learning:

Red is very productive. It's the most productive programming language I know. You can get so much done using so little code! It's also very easy to use after you learn it, but I would like to comment here some of the issues I found in the process. You can't really avoid these pitfalls, but your journey may be easier if you are aware of them.

 # 1 - New way of thinking. It takes longer to learn than expected:

Red's productivity comes with a price. Although the basic examples are easy, it seems to me that it's very hard do real programming in Red without grasping its major concepts. Red is not made of some basic building blocks that you put together as you please, in Red everything is interconnected. Evaluations, datatypes and dialects permeate all coding. Working with the concept of "code is data and data is code" takes practice to get used to. It's like learning a foreign language, you kind of absorb it by repetition.

 # 2 - Wrong datatypes in arguments:

A word in Red may have one of the many, many datatypes available, but functions expect a very definite set of datatypes in its arguments. You will soon find that bug where a seemingly innocent "variable" is crashing your script or giving unexpected results for no apparent reason. A very good idea is to start your debugging by checking the datatype of your arguments. One basic approach would be inserting some "`print type? <variable>`" in your code when things go wrong. You can find out what datatypes your function expects typing "`? <function>`" in the console.

 # 3 - Dialects use only dialect commands:

You will soon use the built-in dialects of Red, as VID (for GUI), parse or draw, and you will try to insert common Red structures inside the dialect block. Bad idea. Dialects may (or may not) have their own commands to let you use regular Red inside their block, but you can't just insert a loop or a branch without proper coding. For example, in VID, you may use `do [<Red code>]` but other dialects require that you use external functions and then evaluate results using `compose`. More on that later, for now, just beware.

So:

```
Red [needs: view]
parse [xxx] [only parse commands here]
view [
    only view commands here
    draw[only draw commands here]

]
```

< Previous topic   Next topic >

Console input and output

Note: console input and output may cause problems if you compile your programs. This makes sense: if you compile it, the console is simply not there! Red Wiki about issues

native! print Red-by-example MyCode4fun

`print` sends data to the console. After the data, it sends a `newline character` to the console. It evaluates its argument before printing it, that is , it applies a `reduce` to the argument before printing.

```
Red []

print "hello"
print 33
print 3 + 5
```

```
hello
33
8
```

native! prin Red-by-example MyCode4fun

`prin` also sends data to the console, but it does NOT send the `newline character` . It evaluates its argument before printing it.

```
Red []

prin "Hello"
prin "World"
prin 42
```

```
HelloWorld42
```

function! probe Red-by-example MyCode4fun

`probe` prints its argument **without evaluation** and **also returns it**. Remember that `print` evaluates its argument. `probe` prints and returns the argument "as it is", so to speak. It's able to show expressions that would cause print to give an error.
It may be used for debugging as a way of showing code (by printing) without changing it.

```
>> print [3 + 2]
5

>> probe [3 + 2]
[3 + 2]
```

```
== [3 + 2]

>> print probe [3 + 2]
[3 + 2]
5
```

```
>> a: [circle 5x4 10]
== [circle 5x4 10]

>> print a
*** Script Error: circle has no value
*** Where: print
*** Stack:

>> probe a
[circle 5x4 10]
== [circle 5x4 10]
```

Described also here, following `mold`.

function! **input** Red-by-example MyCode4fun

Inputs a **string** from the console. Notice that any number typed on console are converted to a string.`newline character` is removed.

```
Red []

prin "Enter a name: "
name: input
print [name "is" length? name "characters long"]
```

```
John
John is 4 characters long
```

routine! **ask** Red-by-example MyCode4fun

Same as `input`, but displays a string.

```
Red []

name: ask "What is your name: "
prin "Your name is "
print name
```

```
What is your name: John
Your name is John
```

< Previous topic Next topic >

Running code

Of course you may save your script as a file and run it from command prompt, as an argument of the Red executable, like this:

```
C:\Users\you\whatever> red-063.exe myprogram.red
```

This will launch the Red interpreter, open the console (REPL) and run your script.

But once the Red environment is running, you can execute code using the built-in function `do` .

native! **do** Red-by-example MyCode4fun

Evaluates the code in its arguments. In other words: executes the code. This argument can be a block, a file, a function or any other value.

```
>> do [loop 3 [print "hello"]]
hello
hello
hello
```

Check the Files chapter before you proceed here.

For example, if you saved a Red script as myprogram.txt you may execute it from the console by typing this:

```
>> do %myprogram.txt
```

Note that in this example the Red interpreter and the text file must be in the same folder, otherwise you must set your paths right.

Also, if you type:

```
>> a: load %myprogram.txt
```

And then:

```
>> do a
```

...your program will run normally.

`do`, `load` and `save` are better understood if you think of Red's console as the screen of some old computer from the 80's running some variation of basic language. You can `load` your program, `save` it, or `do` (execute) it.

You can also load and execute functions saved as text :

```
>> do load %myfunction.txt
```

Notice that you can do all this **from inside a Red program!** So it's a powerful command.

< Previous topic   Next topic >

Stopping code

function! quit Red-by-example MyCode4fun

Stops evaluation and exits the program.

If you type this on the GUI console (REPL), it closes. If you type this on the Command Line Interface, you just exit the Red interpreter.

/return => Stops evaluation and exits the program with a given status. .

```
quit/return 3   ;hands the value 3 to the Operating System
```

On windows if you compile a red app that uses e.g., `quit/return 55` and after running the .exe enter in cmd 'echo %errorlevel%', it will print 55 (or whatever you set).

function! halt Red-by-example MyCode4fun

I think this one just stops (halts) the execution of the script. The documentation says it returns the value 1.

routine! quit-return Red-by-example

Stops evaluation and exits the program with a given status. Seems to me as exactly the same as `quit/return`, but it's a `routine!` datatype, not a `function!` Go figure.

VID DLS on-close Red-by-example MyCode4fun

VID event. Runs a piece of code when you close a GUI window. Mentioned also in GUI Advanced topics.

Run the following program and when you close the window (close the program), it will print "bye!" at the console:

```
Red [needs: view]

view [
      on-close [print "bye!"]
      button [print "click"]
]
```

Control-C

Pressing control-C stops the execution and exits the interpreter in the Command Line Interface, but not on the GUI console.

< Previous topic Next topic >

Datatypes

It may be a good idea to take a look first at the chapters about series, as some examples use built-in functions listed there.

native! type? Red-by-example

Returns the datatype of a value or the datatype of what is assigned to a word in the dictionary:

```
>> type? 33
== integer!

>> type? "house"
== string!

>> birthday: 30/07/1963
== 30-Jul-1963
>> type? birthday
== date!
```

Basic Datatypes:

♦ none! Red documentation Red-by-example

The equivalent of "null" in other programming languages. A non-existing data.

```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]
>> pick a 7
== none
```

♦ logic! Red documentation Red-by-example

Aside from the classic `true` and `false`, Red recognizes `on` , `off`, `yes` and `no` as `logic!` datatype.

```
>> a: 2 b: 3
== 3
>> a > b
== false
```

```
>> a: on
== true
>> a
== true
```

```
>> a: off
== false
>> a
== false
```

```
>> a: yes
== true
>> a
== true
```

```
>> a: no
== false
>> a
== false
```

Notice that, as far as I know, everything that is not `false` , `off` or `no` is considered `true`:

```
>> if "house" [print "It's true!"]
It's true!

>> if 0 [print "It's true!"]
It's true!

>> if [] [print "It's true!"]
It's true!

>> if [false] [print "It's true!"]  ;bizarre!
It's true!
```

♦ string! Red documentation Red-by-example

A series of chars within quotes " " or curly brackets {}. If your string spans over more than one line, curly brackets are mandatory.

Strings are series, and can be manipulated using the the commands described in the chapters about them.

```
>> a: "my string"
== "my string"
```

```
>> a: {my string}
== "my string"
```

```
>> a: {my
{ string}       ;the first "{" is not a typo, is how the console shows it. Try!
== "my^/string"
>> print a
my
string
```

```
>> a: "my new                          ;trying to span over more than one line
*** Syntax Error: invalid value at {"my new}
```

♦ char! Red documentation Red-by-example

Preceded by # and within quotes, `char!` values represent a Unicode code point. They are integer numbers in the range hexadecimal 00 to hexadecimal 10FFFF. (0 to 1,114,111 in decimal.)

#"A" is a `char!`

"A" is a `string!`

It may undergo math operations.

```
>> a: "my string"
== "my string"
>> pick a 2
== #"y"
>> poke a 3 #"X"
== #"X"
>> a
```

```
== "myXstring"
```

```
>> a: #"b"
== #"b"
>> a: a + 1
== #"c"
```

♦ integer! Red documentation Red-by-example

32 bit whole signed numbers. From −2,147,483,648 to 2,147,483,647. If a number is outside this range, Red assigns it a float! datatype.

Note: Dividing 2 integers gives a truncated result:

```
>> 7 / 2
== 3
```

♦ float! Red documentation Red-by-example

64 bit floating point numbers. Represented by numbers with a period or using the e-notation.

```
>> 7.0 / 2
== 3.5
```

```
>> 3e2
== 300.0
```

```
>> 6.0 / 7
== 0.8571428571428571
```

♦ file! Red documentation Red-by-example

Preceded by %. If you are not using the current path, you should add the path within quotes. The path uses forward slashes (/), and back slashes (Windows format) are converted automatically.

```
>> write %myfirstfile.txt "This is my first file"
```

```
>> write %"C:\Users\André\Documents\RED\mysecondfile.txt" "This is
my second file"
```

♦ path! Red documentation Red-by-example

Used to access items inside larger structures using "/". Can be used in many different situations, for example:

```
>> a: [23 45 89]
== [23 45 89]
>> print a/2
45
```

Slashes "/" are also used to access objects and refinements. I don't know the inner workings of the Red interpreter, but it seems to me that those are cases of the path! type.

♦ time! Red documentation Red-by-example

Time is expressed as hours:minutes:seconds.subseconds. Notice that seconds and subseconds are separated by a period, not a colon. You can access each one with a refinement. Check the chapter about Time and timing.

```
>> mymoment: 8:59:33.4
== 8:59:33.4
>> mymoment/minute: mymoment/minute + 1
== 60
>> mymoment == 9:00:33.4
```

```
>> a: now/time/precise      ; a datatype is time!
== 22:05:46.805
>> type? a
== time!
>> a/hour
== 22
>> a/minute
== 5
>> a/second
```

```
== 46.805              ;second is a float!
```

♦ date! Red documentation Red-by-example

Red accepts dates in a variety of formats:

```
>> print 31-10-2017
31-Oct-2017
>> print 31/10/2017
31-Oct-2017
>> print 2017-10-31
31-Oct-2017
>> print 31/Oct/2017
31-Oct-2017
>> print 31-october-2017
31-Oct-2017
>> print 31/oct/2017
31-Oct-2017
>> print 31/oct/17          ;only works if the year is the last
field, but be careful: 1917 or 2017?.
31-Oct-2017
```

Red also checks if dates are valid, even considering leap years.
You can refer to day, month or year using refinements:

```
>> a: 31-oct-2017
== 31-Oct-2017
>> print a/day
31
>> print a/month
10
>> print a/year
2017
```

♦ pair! Red documentation Red-by-example

Represents points in a cartesian coordinate system (x y axys). Represented by integers separated by "x" e.g. 23x45.

```
>> a: 12x23
== 12x23
>> a: 2 * a
== 24x46
```

```
>> print a/x
24
>> print a/y
46
```

♦ percent! [Red documentation] [Red-by-example]

Represented by adding the "%" symbol after the number.

```
>> a: 100 * 11.2%
== 11.2
>> a: 1000 * 11.3%
== 113.0
```

♦ tuple! [Red documentation] [Red-by-example]

A tuple! is a list of 3 up to 12 bytes (bytes range from 0 to 255) separated by periods.
Notice that 2 numbers separated by a period is a float! not a tuple!
Tuples are useful to represent things like version numbers, IP addresses , and colours (example: 0.255.0).
A tuple! is not a series, so most series operations give an error when applied. Some operations that can be performed on a tuple! are: random, add, divide, multiply, remainder, subtract, and, or, xor, length?, pick (not poke), reverse.

```
>> a: 1.2.3.4
== 1.2.3.4
>> a: 2 * a
== 2.4.6.8
>> print pick a 3
6
>> a/3: random 255
== 41
>> a
== 2.4.41.8
```

Words datatypes:

When you use `type?` to determine the datatype of a word, you usually get the datatype of the value assigned to that word, as in:

```
>> test: 33.8
== 33.8
>> type? test
== float!
```

However, the word itself (in this case "test") may assume different datatypes, depending on context:

	datatype
word	word!
word**:**	set-word!
:word	get-word!
'word	lit-word!
/word	refinement!

```
>> to-word "test"
== test

>> make set-word! "test"
== test:

>> make get-word! "test"
== :test

>> make lit-word! "test"
== 'test
```

Datatype classes - ♦ number! and ♦ scalar!

Some datatypes are classes of datatypes:

Any of the following datatypes is also a `number!` datatype: `integer!`, `float!`, `percent!`

And any any of the following datatypes is also a `scalar!` datatype: `char!`, `integer!`, `float!`, `pair!`, `percent!`, `tuple!`, `time!`, `date!`

< Previous topic Next topic >

Hash! vector! and map!

I think these are special datatypes that deserve a special chapter for them. They may improve the quality and speed of your work considerably.

Hash! and vector! are high performance series, i.e., they are faster when dealing with large sets.

I suggest you take a look at the Blocks & Series chapters before studying this.

♦ hash! Red-by-example

`hash!` is a series that is "hashed" to make searches faster. Since "hashing" consumes resources, it is not worth creating a `hash!` for a series that will be searched just a few times. However, if your series will be constantly searched, consider making it a `hash!` . Rebol website claims searches may be 650 times faster than on a regular series.

```
>> a: make hash! [a 33 b 44 c 52]
== make hash! [a 33 b 44 c 52]

>> select a [c]
== 52

>> select a 'c
== 52

>> a/b
== 44
```

Nothing new really, it's just a series.

♦ vector! Red-by-example

Vectors are high performance series of `integer!` ,`float!`, `char!` or `percent!`

To create a vector you must use `make vector!`

While `hash!` allow you to perform searches faster, `vector!` allows faster math operations as they can be performed on the entire series at once.

```
>> a: make vector! [33 44 52]
== make vector! [33 44 52]

>> print a
33 44 52

>> print a * 8
264 352 416
```

Notice that you could not do that on a regular series:

```
>> a: [2 3 4 5]
== [2 3 4 5]

>> print a * 2
*** Script Error: * does not allow block! for its value1 argument
*** Where: *
*** Stack:
```

♦ map! & action! put Red documentation Red-by-example

Maps are high performance dictionaries that associate keys with values (key1: val1 key2: val2 ... key3: val3).

Maps are **not** series. You can't use most of series' built-in functions (commands) on them.

To set and retrieve values from the dictionary we use `select` (from series) and a a special action: `put`.

```
>> a: make map! ["mini" 33 "winny" 44 "mo" 55]
== #(
    "mini" 33
    "winny" 44
    "mo" 55
...

>> print a
"mini" 33
"winny" 44
"mo" 55

>> print select a "winny"
44

>> put a "winny" 99
== 99
```

```
>> print a
"mini" 33
"winny" 99
"mo" 55
```

How to native! extend a map!

Since `map!` is not a series and so you can't use things like `append`, `poke` or `insert`, how do you add new items to it? The answer is the built-in function `extend`.

```
>> a: make map! ["mini" 33 "winny" 44 "mo" 55]
== #(
    "mini" 33
    "winny" 44
    "mo" 55
)

>> extend a ["more" 23 "even more" 77]

   ...

>> probe a
#(
    "mini" 33
    "winny" 44
    "mo" 55
    "more" 23
    "even more" 77
)
   ...
```

< Previous topic

Next topic >

Other datatypes:

More information on these datatypes can be found at Red documentation and Red-by-example.

♦ issue!

Series of characters used to sequence symbols or identifiers for things like telephone numbers, model numbers, serial numbers, and credit card numbers. An issue! has to start with the character "#". Most characters can be used inside an `issue!`, a notable exception being the slash "/".

```
>> a: #333-444-555-999
== #333-444-555-999

>> a: #34-Ab.77-14
== #34-Ab.77-14
```

♦ url!

Represented by *<protocol>***://***<path>*

```
>> a: read http://www.red-lang.org/p/about.html
== {<!DOCTYPE html>^/<html class='v2' dir='ltr' x
```

♦ email!

Used to identify email addresses. No detailed syntax-checking is performed, it must only contain an @ character.

```
>> a: myname@mysite.org
== myname@mysite.org

>> type? a
== email!
```

♦ image!

To create a image! you must use make image!
The external image formats supported are GIF, JPEG, PNG and BMP.
When you load an image file, the data is typed as image! It is unlikely that you will create image with text, but the format would be:

```
>> a: make image! [30x40 #{ ; here goes the data...
;You can change or get information from your image using the actions
that apply to series:
>> a: load %heart.bmp
== make image! [30x20 #{
    00A2E800A2E800A2E800A

>> print a/size
30x20

>> print pick a 1            ; getting the RGBA data of pixel 1
0.162.232.0

>> poke a 1 255.255.255.0    ; changing the RGBA data of pixel 1
== 255.255.255.0
```

♦ block!

Any series within brackets.

♦ paren!

Any series within parentheses.

♦ refinement!

Preceded by "/" - indicate a variation in the use or an extension of the meaning of a `function!`, `object!`, `file!` or `path!`.

♦ action!

Is the datatype of all "actions" in red, e.g. `add` , `take` , `append`, `negate` etc.

```
>> action? :take ; Colon is mandatory.
== true
```

To get a list of all `action!` words type:

```
>> ? action!
```

♦ op!

Is the datatype of infix operators , like `+` or `**`.

♦ routine!

Used to link to external code

♦ binary!

Is a series of bytes. It's the raw storage format and it can encode data such as images, sounds, strings (in formats like UTF and others), movies, compressed data, encrypted data and others.

The source format may be on base 2, 16 or 64. I'm not sure which is the default in Red,

The source format is: #{...}

#{3A1F5A} ; base 16

2#{01000101101010} ; base 2

64#{0aGvXmgUkVCu} ; base 64

♦ word!

The mother of all datatypes. When a word is created it has this datatype.

♦ datatype!

Is the datatype of all the `datatype!` words listed in this chapter.

♦ event!

This datatype is explained in the Event! mouse position and key pressed.

♦ function!

♦ object!

♦ handle!

♦ unset!

♦ tag!

♦ lit-path!

♦ set-path!

♦ get-path!

♦ bitset!

♦ typeset!

♦ error!

♦ native!

< Previous topic Next topic >

Datatypes conversion:

Red documentation

action! to

Converts one `datatype!` to another, e.g. an `integer!` to a `string!` , a `float!` to an `integer!` and even a `string!` to a `number!`.

```
>> to integer! 3.4
== 3
```

```
>> to float! 23
== 23.0
```

```
>> to string! 23.2
== "23.2"
```

```
>> to integer! "34"
== 34
```

function! to-time

Converts values to `time!` datatype.

```
>> to-time [22 55 48]
== 22:55:48
```

```
>> to-time [22 65 70]
== 23:06:10
```

```
>> to-time "11:15"
```

```
== 11:15:00
```

native! as-pair

Converts two `integer!` or `float!` into `pair!`. Note that this is not exactly a "conversion" as we are creating a new value from two values that may even be of different datatypes, as is the case when we "join" a `float!` and an `integer!` into a `pair!`.

```
>> as-pair 11 53
== 11x53
```

```
>> as-pair 3.2 5.67
== 3x5
```

```
>> as-pair 88 12.7
== 88x12
```

function! to-binary

Convert to `binary!` value. It seems that it's not a base converter, but a datatype converter.

```
>> to-binary 8
== #{00000008}
```

```
>> to-binary 33
== #{00000021}
```

< Previous topic Next topic >

Accessing and formatting data

native! get Red-by-example

Every word in Red, the native ones and the ones you create, go into a dictionary. If the word is associated with an expression, the dictionary keeps the whole expression that may or may not be evaluated depending on the type of call that fetch the word

If you want to know what is the dictionary description of a word, you use `get` . Notice that when you refer to a word in Red (the word itself, not the value) you precede it with a quote ('). `get` gives you the "meaning" even of Red's native words, but returns an error if used on a value, e.g. `integer!` `pair!` `tuple!` :

```
>> get 'print
== make native! [[
    "Output...

>> get 'get
== make native! [[
    "Return...

>> a: 7
== 7

>> get 'a
== 7

>> a: [7 + 2]
== [7 + 2]

>> get 'a
== [7 + 2]

>> get 8
*** Script Error: get does not allow integer! for its word argument
```

action! mold Red-by-example MyCode4fun

`mold` turns a `datatype!` (i.e. a `block!`, an `integer!` a `series!` etc.) into a string and **returns** it:

```
>> type? 8
== integer!

>> type? mold 8
== string!

>> print [4 + 2]
6

>> print mold [4 + 2]
[4 + 2]
```

Refinements
/only - Exclude outer brackets if value is a block!
/all - Return value in loadable format
/flat - Exclude all indentation
/part - Limit the length of the result, where limit is an integer!

action! form [Red-by-example](#) [MyCode4fun](#)

`form` also turns a `datatype!` into a string, but depending on the type, the resulting text might **not** contain extra type information (such as [ ] { } and "") as would be produced by `mold`. Useful for [String and text manipulation](#).

```
Red []
print "---------MOLD---------"
print mold {My house
      is a very
      funny house}
print "---------FORM---------"
print form {My house
      is a very
      funny house}
print "---------MOLD---------"
print mold [3 5 7]
print "---------FORM---------"
print form [3 5 7]
```

```
---------MOLD---------
"My house^/^-is a very^/^-funny house"
---------FORM---------
My house
    is a very
    funny house
---------MOLD---------
[3 5 7]
---------FORM---------
3 5 7
```

Allows the refinement `/part` to limit the number of characters.

Main uses for mold and form:

`mold` is basically used to turn a series into code that can be saved and interpreted later

`form` is basically used to generate regular text from a series

```
>> a: [b: drop-down data[ "one" "two" "three"][print a/text]]
== [b: drop-down data ["one" "two" "three"] [print a/text]]

>> mold a
== {[b: drop-down data ["one" "two" "three"] [print a/text]]}

>> form a
== "b drop-down data one two three print a/text"
```

function! **??** Red-by-example

Prints a word and the value it refers to, in molded form.

```
>> cat: 33
== 33
>> ?? cat
cat: 33
```

function! **probe** Red-by-example MyCode4fun

`probe` prints its argument without evaluation **but also returns it**. Remember that `print` evaluates its argument. `probe` prints and returns the argument "as it is", so to speak.
It may be used for debugging as a way of showing code (by printing) without changing it.

```
>> print [3 + 2]
5

>> probe [3 + 2] [3 + 2]
== [3 + 2]

>> print probe [3 + 2]
[3 + 2]
5
```

native! **reduce** Red-by-example MyCode4fun

Evaluates expressions inside a block and returns a new block with the evaluated values. Take a look at the chapter about evaluation.

```
>> a: [3 + 5 2 - 8 9 > 3]
== [3 + 5 2 - 8 9 > 3]

>> reduce a
== [8 -6 true]

>> b:[3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]
== [3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]

>> reduce b
== [8 11 true [6 + 6 3 > 9]]          ;it does not evaluate
expressions of blocks inside blocks

>> b
== [3 + 5 2 + 9 7 > 2 [6 + 6 3 > 9]]    ;the original block remains
unchanged.
```

/into => Put results in out block, instead of creating a new block.

Here I quote Vladimir Vasilyev (@9414):

" Imagine that block is a piece of paper, and some words are written on it. Initially they are just scribbles and sets of letters with symbols - "London" is a 6-letter word. But if you "infer" their meaning, then they become something else - London is the capital of Great Britain.

This is kinda the same with Red. [a] is a list of paper with one word written on it, reduce "infers" the meaning of all words (of all expressions, to be specific), and a brings forward its meaning."

```
>> London: "the capital of Great Britain"
== "the capital of Great Britain"

>> paper: [London]
== [London]

>> paper
== [London]

>> reduce paper   ; reduce "returns" evaluation result.
== ["the capital of Great Britain"]
```

```
>> probe paper
[London]          ; this is "returned" (could be assigned to a word, for
example).
== [London]       ; this is the "output" of probe (printed).

>> print paper    ; print reduces (evaluates) and prints.
the capital of Great Britain

>> type? first paper
== word!

>> type? first reduce paper
== string!
```

function! **collect** and **keep** Red-by-example MyCode4fun

Collect in a new block all the values passed to `keep` function from the body block.
In other words: creates a new block keeping only the values determined by `keep`, usually values that fulfill some condition.

```
Red []

a: [11 "house" 34.2 "dog" 22]
b: collect [
      foreach element a [if string? element [keep element]] ;keeps string
elements
      ]
print b
```

```
house dog
```

/into => Insert into a buffer instead (returns position after insert).

syntax: collect/into [........] <existing output block>

```
Red []

c: ["one" "two"]                    ; creating the output block with
some elements
a: [11 "house" 34.2 "dog" 22]    ; a generic series
collect/into [
      foreach element a [if scalar? element [keep element]] ;keeps
numbers of a
      ] c                                    ;appends them into c
print c
```

```
one two 11 34.2 22
```

native! compose [Red-by-example](#) [MyCode4fun](#)

Returns a copy of a block, evaluating only `paren!` (things inside parenthesis).
Compose is very important for the [DRAW dialect](#);

```
Red []

a: [add 3 5 (add 3 5) 9 + 8 (9 + 8)]
print compose a              ;print evaluates everything!!
probe compose a              ;probe prints "as is"
```

```
8 8 17 17
[add 3 5 8 9 + 8 17]
```

/deep => Compose nested blocks.

```
Red []

a: [add 3 5 (add 3 5) [9 + 8 (9 + 8)]]
probe compose a
probe compose/deep a
```

```
[add 3 5 8 [9 + 8 (9 + 8)]]
[add 3 5 8 [9 + 8 17]]
```

/only => Compose nested blocks as blocks containing their values.

/into => Put results in out block, instead of creating a new block.

syntax: compose/into [........] <existing output block>

```
Red []

a: [add 3 5 (add 3 5) 9 + 8 (9 + 8)]
b: []
compose/into a b
probe b
```

```
[add 3 5 8 9 + 8 17]
```

[< Previous topic](#) [Next topic >](#)

Math and logic

Most of Red's math and logic is usual, except maybe the order of evaluation.

Interesting notes:

- input to Red may use a period **or** a coma as decimal separator for `float!`:

```
>> 5,5 + 9.2     ; notice the coma in the first number and the period
in the second
== 14.7          ; Red always uses a period for its output of floats
```

- if you want to use apostrophes for readability, Red ignores them:

```
>> 5'420'120,00 * 2
== 10840240.0
```

- you may evaluate strings using `do`:

```
>> do "2 + 5"
== 7
```

Below I list the operators (words) used for calculations, adding notes that I find useful. Most of them have no need for a detailed description.

Math

The basics:

The following group have a both a **functional** (e.g. add) and an **infix** operator (e.g. "+") . They accept number! char! pair! tuple! or vector! as arguments (except power?).

Note that if you use the functional operator, it goes before the operands (e.g.: 3 + 4 <=> add 3 4).

I'll try to give examples using more complex datatypes than integers and floats:

action! add or op! +

```
>> add 3x4 2x3
== 5x7

>> now/time + 0:5:0    ; added five minutes to current time
== 7:16:27
```

action! subtract or op! -

```
>> subtract 33 13
== 20

>> 3.4.6 - 1.2.1
== 2.2.5

>> now/month - 3   ;is october now
== 7
```

action! multiply or op! *

```
>> multiply 3x2 2x5
== 6x10

>> 2.3.4 * 3.7.2
== 6.21.8
```

action! divide or op! /

```
>> divide 3x5 2
== 1x2      ;truncate result because pair! is made of integer!

>> divide 8 3 ;truncate result because both are integer!
== 2

>> 8 / 3.0  ;3.0 is a float! so result is float!
== 2.666666666666667
```

action! power or op! **

```
>> 3 ** 3
== 27
```

action! absolute

Evaluates an expression and returns the absolute value, that is, a positive number.

```
>> absolute 2 - 7
== 5
```

action! negate

Invert the signal of a value, that is: positive <=> negative

```
>> negate 3x2
== -3x-2
```

float! pi

3,141592...

action! random

Returns a random value of the same type as its argument.

If argument is an integer, returns an integer between 1 (inclusive) and the argument (inclusive).

If argument is a float, returns a float between 0 (inclusive) and the argument (inclusive).

If the argument is a series, it shuffles the elements.

```
>> random 10
== 2

>> random 33x33
== 13x23

>> random 1
== 1

>> random 1.0
== 0.07588539741741744

>> random "abcde"
== "cedab"

>> random 10:20:05
== 8:02:32.5867693
```

Refinements:

/seed - Restart or randomize. I think the use of this is if your random function is called many times within a program. In this case it may not be so random unless you reestart it with a seed.

/secure - TBD: Returns a cryptographically secure random number.

/only - Pick a random value from a series.

```
>> random/only ["fly" "bee" "ant" "owl" "dog"]
== "fly"

>> random/only "aeiou"
== #"o"
```

action! round

Returns the nearest integer value. Halves (e.g. 0,5) are rounded away from zero by default.

```
>> round 2.3
== 2.0

>> round 2.5
== 3.0

>> round -2.3
== -2.0

>> round -2.5
== -3.0
```

Refinements:

/to - You supply the "precision" of your rounding:

```
>> round/to 6.8343278 0.1
== 6.8

>> round/to 6.8343278 0.01
== 6.83

>> round/to 6.8343278 0.001
== 6.834
```

/even - Halves (e.g. 0.5) are rounded not "up" as default, but towards the even integer.

```
>> round/even 2.5
== 2.0          ;not 3
```

/down - Simply truncates the number, but keeps the number a `float!`.

```
>> round/down 3.9876
== 3.0

>> round/down -3.876
== -3.0
```

/half-down - Halves round toward zero, not away from zero.

```
>> round/half-down 2.5
== 2.0

>> round/half-down -2.5
== -2.0
```

/floor - Rounds in negative direction.

```
>> round/floor 3.8
== 3.0

>> round/floor -3.8
== -4.0
```

/ceiling - Rounds in positive direction.

```
>> round/ceiling 2.2
== 3.0

>> round/ceiling -2.8
== -2.0
```

/half-ceiling - Halves round in positive direction.

```
>> round/half-ceiling 2.5
== 3.0

>> round/half-ceiling -2.5
== -2.0
```

native! square-root

Takes any `number!` as argument.

Remainders etc.:

action! **remainder** or op! *//* (* see "%" operator below)

Takes `number!` `char!` `pair!` `tuple!` and `vector!` as arguments. Returns the rest of dividing the first by the second value.

```
>> remainder 15 6
== 3

>> remainder -15 6
== -3

>> remainder 4.67 2
== 0.67

>> 17 // 5
== 2

>> 4.8 // 2.2
== 0.3999999999999995
```

op! %

Returns what is left over when one value is divided by another. Seems to me as the same as remainder, look at the examples:

```
>> remainder 11x19 3
== 2x1

>> 11x19 % 3
== 2x1

>> 11x19 // 3
*** Script Error: cannot compare 2x1 with 0     ; WHAT?!
*** Where: <
*** Stack: mod
```

function! modulo

From the documentation: "Wrapper for MOD that handles errors like REMAINDER. Negligible values (compared to A and B) are rounded to zero". Can't really figure this one out.

```
>> modulo 9 4
== 1

>> modulo -15 6
== 3

>> modulo -15 -6
== 3

>> modulo -15 7        ;?????
== 6

>> modulo -15 -7       ;?????
== 6
```

Logarithms etc.:

function! exp

Raises *e* (the natural number) to the power of the single argument.

native! log-10

Returns the logarithm base 10 of the argument.

native! log-2

Returns the logarithm base 2 of the argument.

native! log-e

Returns the logarithm base *e* of the argument.

Trigonometry:

All the trigonometric functions with long names (`arccosine`, `cosine` etc) use degrees as default, but accept the refinement `/radians` to use this unit. The short name versions (`acos`, `cos` etc.) take radians as arguments and require it to be a `number!`

function! **acos** or native! **arccosine**

function! **asin** or native! **arcsine**

function! **atan** or native! **arctangent**

Returns the trigonometric arctangent.

function! **atan2** or native! **arctangent2**

Returns the angle of the point y/x in radians, when measured counterclockwise from a circle's x axis (where 0x0 represents the center of the circle). The return value is between -pi and +pi.

function! **cos** or native! **cosine**

function! **sin** or native! **sine**

function! **tan** or native! **tangent**

Extras:

native! **max**

Returns the greater of two arguments. Arguments may be scalar! or series!

I'm not sure how it selects the greater series, but is seems to choose the series with the first greater value from left to right.

```
>> max 8 12
== 12
```

```
>> max "abd" "abc"
== "abd"

>> max [1 2 3] [3 2 1]
== [3 2 1]

>> max [1 2 99] [3 2 1]
== [3 2 1]
```

In a `pair!` comparison, it returns the greater for each element:

```
>> max 12x6 7x34
== 12x34
```

native! min

Returns the smaller of two arguments. Notes for `max` apply here too.

action! odd?

Returns `true` if argument (`integer!`) is odd, and `false` otherwise.

action! even?

Returns `true` if argument (`integer!`) is even, and `false` otherwise.

native! positive?

`true` if greater than zero. Note: `false` if zero.

native! negative?

`true` if less than zero. Note: `false` if zero.

native! zero?

`true` only if zero.

function! **math**

Evaluates a block! using the normal mathematical rules of precedence, that is, divisions and multiplications are evaluated before additions and subtractions and so on. As of november 2018, math dialect unfinished and may produce unexpected results!

function! **within?**

It has 3 arguments of the pair! type. The first is a point's coordinates (origin in the upper left corner). The other two describe a rectangle, the first is its upper left origin, and the second is the width and height. If the point is inside or at the edge, returns `true`, otherwise returns `false` .

native! **NaN?**

Returns true if the argument is 'not a number',otherwise false.

native! **NaN**

Returns TRUE if the number is Not-a-Number.

function! **a-an**

Returns the appropriate variant of "a" or "an" (simple, vs 100% grammatically correct).

Logic

action! **and~** or op! **and (infix)**

native! **equal?** or op! **=**

native! **greater-or-equal?** or op! **>=**

native! **greater?** or op! **>**

native! **lesser-or-equal?** or op! **<=**

native! **lesser?** or op! **<**

native! **not**

native! **not-equal?** or op! **<>**

action! **or~** or op! **or (infix)**

native! **same?** or op! **=?**

Returns `true` if the arguments refer to the same data (object, string etc.), that is, it they both refer to the same space in memory.

```
>> a: [1 2 3]
== [1 2 3]

>> b: a          ; b points to the same data as a
== [1 2 3]

>> a =? b
== true          ; they are the same
```

```
>> c: [1 2 3]
== [1 2 3]

>> c =? a        ; c is equal to a, but is not the same data in
memory.
== false
```

native! **strict-equal?** or op! **==**

Returns `true` if the arguments are exactly equal, with same datatype same lower-case/uppercase (strings) etc.

```
>> a: "house"
>> b: "House"
>> a = b
== true

>> a == b
== false
```

< Previous topic Next topic >

Other bases

native! to-hex Red-by-example MyCode4fun

Converts an `integer!` to a hex `issue!` datatype (with leading # and 0's).

```
>> to-hex 10
== #0000000A

>> to-hex 16
== #00000010

>> to-hex 15
== #0000000F
```

/size => Specify number of hex digits in result.

```
>> to-hex/size 15 4
== #000F

>> to-hex/size 10 2
== #0A
```

native! enbase and native! debase, Red-by-example MyCode4fun

These are used do code and decode binary-coded strings.
These are not for number conversion and, honestly, I don't understand the use for them, but here is how they work:

```
>> enbase "my house"
== "bXkgaG91c2U="

>> probe to-string debase "bXkgaG91c2U="
"my house"
== "my house"
```

/base => Binary base to use. It may be 64 (default), 16 or 2.

```
>> enbase/base "Hi" 2
== "0100100001101001"

>> probe to-string debase/base "0100100001101001" 2
"Hi"
== "Hi"
```

native! dehex Red-by-example

Converts URL-style hex encoded (%xx) strings.

```
>> dehex "www.mysite.com/this%20is%20my%20page"
== "www.mysite.com/this is my page" ; Hex 20 (%20) is space
```

```
>> dehex "%33%44%55"
== "3DU"
; %33 is hex for "3", %44 is hex for "D" and %55 is hex for "U".
```

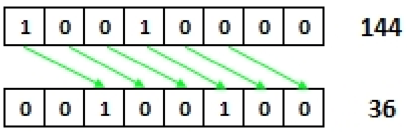
Bitwise functions:

Bitwise functions work at the binary level of values:

op! >> Red-specs Red-by-example

right shift - documentation says: "lowest bits are shifted out, highest bit is duplicated".

```
>> 144 >> 2
== 36
```

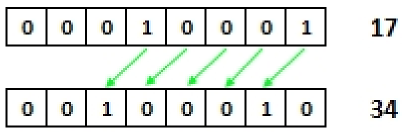


I could not figure out how to duplicate the highest bit if it's 1. I tried 32 bit words, but Red converts them to floats.

op! << Red-specs Red-by-example

left shift - highest bits are shifted out, zero bits are added to the right.

```
>> 17 << 1
== 34
```

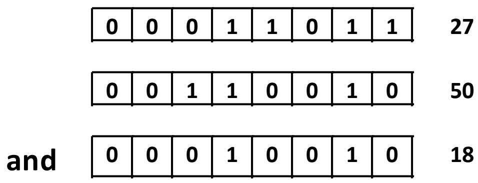


op! **>>>** [Red-specs] [Red-by-example]

logical shift - lowest bits are shifted out, zero bits are added to the left. I could not figure out how this is different from `>>`.

op! **and** & **and~** [Red-specs] [Red-by-example]

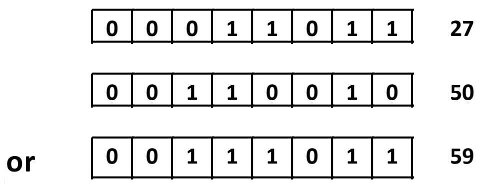
```
>> 27 and 50
== 18
```



The functional version (not infix) of `and` is `and~`

op! **or** & **or~** [Red-specs] [Red-by-example]

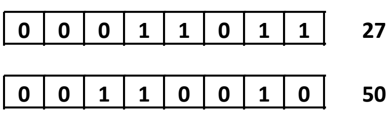
```
>> 27 or 50
== 59
```



The functional version (not infix) of `or` is `or~`

op! **xor** & **xor~** [Red-specs] [Red-by-example]

```
>> 27 xor 50
== 41
```



xor	0	0	1	0	1	0	0	1	**41**

The functional version (not infix) of `xor` is `xor~`

action! **complement** Red-specs Red-by-example

todo -

```
todo
```

< Previous topic Next topic >

Cryptography

native! checksum Red-by-example

Computes a checksum, CRC, hash, or HMAC.
Arguments may be `string!` `binary!` or `file!`

```
Red []

print "-----------  MD5  --------------"
print checksum "my house in the middle of our street" 'MD5
print "----------  SHA1  --------------"
print checksum "my house in the middle of our street" 'SHA1
print "---------  SHA256 --------------"
print checksum "my house in the middle of our street" 'SHA256
print "---------  SHA384  -------------"
print checksum "my house in the middle of our street" 'SHA384
print "---------  SHA512  -------------"
print checksum "my house in the middle of our street" 'SHA512
print "---------  CRC32  --------------"
print checksum "my house in the middle of our street" 'CRC32
print "----------  TCP  --------------"
print checksum "my house in the middle of our street" 'TCP
```

```
-----------  MD5  --------------
#{41F2FF19E5D7DF3B0E79FA9687C08397}
```

```
----------  SHA1  --------------
#{E97AE5E15E8EC1B87B0113E6A4758AAAE6E26901}
```

```
---------  SHA256 --------------
#{
98E2A2BFF328D893161CA6B6F50BA64D544026BD8C24C2022BE7007832714BA4
}
```

```
---------  SHA384  -------------
#{
2EAEA11D12F4CE8BE3CDE33DDED08765BFDCE1F277CF8E2126F7B1B6D4D17E31
96D05D2427576C348A0FECF63537B7D3
}
```

```
---------  SHA512  -------------
#{
0FAA749EAAEC728A6D821B85AC49CBE96DCE59E3FDC8E1005A3256A4CCE6797A
11603E9DB6B870C166057CF5EFBABB2365A87F37CDF2C8C1BF86DC8CE6D948C9
}
```

```
---------  CRC32  --------------
-1630692232
```

```
----------  TCP  --------------
13706
```

/with => Extra value for HMAC key or hash table size; not compatible with TCP/CRC32 methods.

I believe hash is not implemented in Red 0.63 and I could not figure out how HMAC works.

< Previous topic Next topic >

Blocks & Series

Don't miss the series' page at Red-by-example.

Blocks

Red is built on "blocks". Essencially anything delimited by brackets is a block: [one block], [another block [block within a block]]

Series

Series are group of elements. They are an essential topic on Red Programming. In fact, data and even Red programs themselves are series. The elements of a series can be anything inside the Red lexicon: data, words, functions, objects, and other series.

```
>> myFirstSeries: ["John" "Mary" 33 55 [9.2 8]]
== ["John" "Mary" 33 55 [9.2 8]]
```

Strings etc.

Notice that strings are treated by Red as series of characters, and so the techniques used to manipulate series are also used for string operations. However, since string manipulation is so important, there is a special Strings and text manipulation chapter.

Actually, a lot of datatypes are also series that can be manipulated with the built-in functions (commands) described in the following chapters.

Arrays

Toomas Vooglaid's matrix DLS

Other languages have a data type called array. It is not difficult to realize that a one dimensional array is simply a series (not really, see comment), and multi-dimensional arrays are series that contain other series as elements.

Here is an example of a 3 x 2 array:

```
>> a: [[1 2][3 4][5 6]]
== [[1 2] [3 4] [5 6]]
```

To access its elements, you may use "/":

```
>> a/1
== [1 2]

>> a/1/1
== 1

>> a/3/2
== 6
```

The following script creates a 5 by 5 two dimensional array, inserts a number in it and prints some results:

```
Red [needs: 'view]
size: 5x5
matrix: make block! size/x
loop size/x [
    row: make block! size/y
    loop size/y [append row none]
    append/only matrix row
]
new-line/all matrix on ;just for pretty printing...
                       ; adds new lines after each row block
matrix/3/4: 23
probe matrix
print matrix/3/4
```

```
[
    [none none none none none]
    [none none none none none]
    [none none none 23 none]
    [none none none none none]
    [none none none none none]
]
23
>>
```

Using variable as keys for series:

Suppose you want to refer to the 4th element of a series using a value associated with a word. You can't use the word directly, you must use the `:word` syntax:

```
>> a: ["me" "you" "us" "them" "nobody"]
== ["me" "you" "us" "them" "nobody"]
>> b: 4
== 4
>> a/b         ;this does not work as expected!!!
== none
>> a/:b        ;this works!
== "them"
```

It seems words are not evaluated by default to allow their use as keys.

By the way, this also works:

```
>> a/(b)       ;this also works!
== "them"
```

< Previous topic   Next topic >

Series navigation

- The first element of a series is called "`head`". As we will see, it may not be the "`first`" as we manipulate the series;
- AFTER the last element of a series there is something called "`tail`". It has no value.
- Every series has an "**entry index**". The best definition of it is "where the usable part of this series begin". **Many operations with series have this "entry index" as a starting point**. You can move the **entry index** back and forth to change the result of your operations.
- Every element of the series have an index number, starting with 1 (not zero!) at the first element.
- Starting from the position of the **entry index**, the elements of the series have an alias: "`first`" for the first, "`second`" for the second and so on until "`fifth`".

Note: I made up the name "**entry index**". It is not in the documentation. I have seen the "**entry index**" being called just "index", but I dislike that, as it may cause confusion with the index number of the elements. It is a somewhat subtle concept. Nowvorto @nowvorto_twitter suggest it should be called "**first index**" since this index always points to the element returned by `first` command, noting that `head` index will always be 1. This makes sense, and I may change it in the future.

action! head? action! tail? action! index? Red-by-example

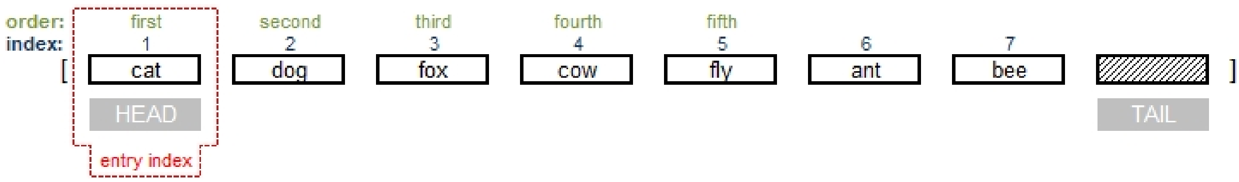
These built-in functions return information about the position of the **entry index**. If the **entry index** is at the head, `head?` returns `true`, otherwise `false`. The same logic applies to `tail?` . `index?` returns the index number of the **entry index** location.

The following examples will make their use clear.

Lets create the series **s** having the strings **"cat" "dog" "fox" "cow" "fly" "ant" "bee"** :

```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

We will have something that look like this:




```
>> head? s
== true
```

```
>> index? s
== 1
```

```
>> print first s
cat
```

action! head action! tail Red-by-example on head Red-by-example on tail

`head` moves the **entry index** to the first element of the series, the head.

`tail` moves the **entry index** to position after the last element of the series, the tail.

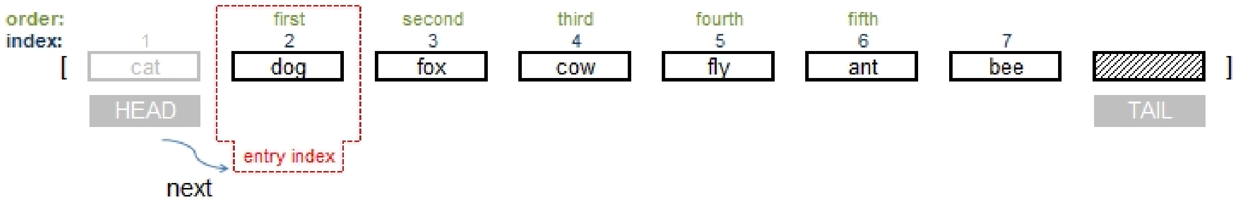
`head` and `tail` by themselves don't change the series, `head` only **returns** the whole series and `tail` **returns** nothing. To change the series you must do an assignment, e.g. `list: head list`

action! next Red-by-example

`next` moves the **entry index** one element towards the tail. Notice that `next` only **returns** the changed series, does not modify it. Therfore, simply repeating `next` on the same series will not make the **entry index** go further than the second position, because you would be doing it on the original series, where the **entry index** is still over the first element. So for most practical uses, we reassign the series to a word (variable). In our example it would be: `s: next s.`

```
>> s: next s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

Now we have:




```
>> print s
dog fox cow fly ant bee

>> head? s
== false

>> print first s
dog

>> index? s
== 2
```

Notice that even though the first element is now "dog", the index remains 2!

action! **back** Red-by-example

`back` is the opposite of `next`: moves the **entry index** one element towards the head. If you use `back` in our **s** series "cat" is brought back from oblivion into the series again! It was never deleted!

This means that Red did not discard any part of the old **s**. This is part of the peculiarities of Red: the data remains there, embedded in the code.

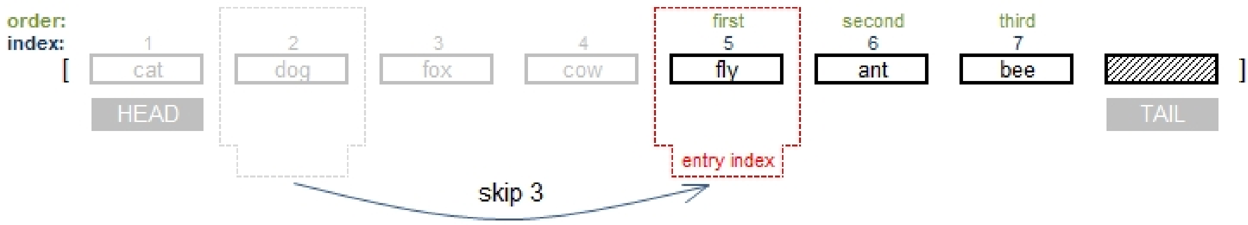
After you moved forward the index of our series **s**, even if you assign it to another word (variable) like **b**(b: s) you can still perform back and negative skip operations on **b** and retrieve the "hidden" values of **s** because **b** points to the same data as **s**.

If you want to avoid that, you must create your new variable using `copy`

Like I mentioned before, in Red, unlike other languages, the variable (word) is assigned to the data and not the other way around.

action! **skip** Red-by-example MyCode4fun

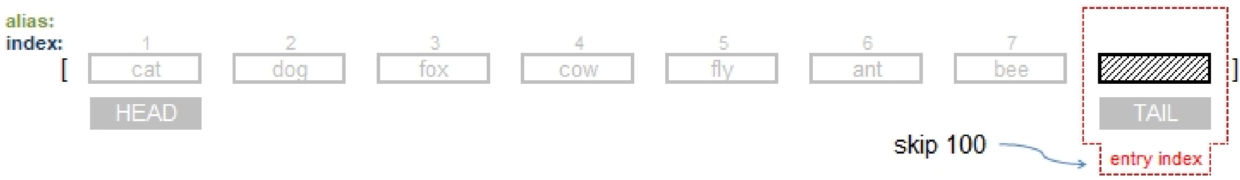
Moves the **entry index** a given number of elements towards the tail.




```
>> s: skip s 3
== ["fly" "ant" "bee"]

>> print s
fly ant bee

>> print first s
fly

>> print index? s
5
```

If the number of skips is larger then the number of elements in the series, the **entry index** stays at the tail.

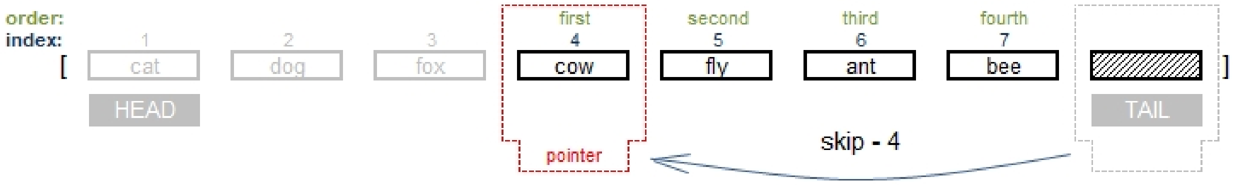
```
>> s: skip s 100
== []
```




```
>> tail? s
== true

>> index? s
== 8
```

You can do negative skips to restore elements of the series:

```
>> s: skip s -4
== ["cow" "fly" "ant" "bee"]
```




```
>> print first s
cow

>> print index? s
4
```

< Previous topic Next topic >

Series "getters"

There are so many commands to manipulate series that I have split them into two chapters: one for the built-in functions (commands) that get information from a series, that I call "getters", and another for those that change the series directly.

The "getter" commands only return values, without altering the series. Notice that any "getter" command may be used to change the series if you reassign the series to the returned value.

action! length? Red-by-example MyCode4fun

Returns the size of a series from the current index to the end.

```
>> a: [1 3 5 7 9 11 13 15 17]
== [1 3 5 7 9 11 13 15 17]

>> length? a
== 9

>> length? find a 13         ;see the command "find"
== 3                         ;from "13" to the tail there are 3
elements
```

function! empty? Red-by-example MyCode4fun

Returns `true` if a series is empty, otherwise returns `false`.

```
>> a: [3 4 5]
== [3 4 5]

>> empty? a
== false

>> b:[]
== []

>> empty? b
== true
```

action! **pick** [Red-by-example](#) [MyCode4fun](#)

Picks the value from a series at the position given by the second argument.

pick [⓪①②③④⑤] 4 == ③

```
>> pick ["cat" "dog" "mouse" "fly"] 2
== "dog"
```

```
>> pick "delicious" 4
== #"i"
```

action! **at** [Red-by-example](#) [MyCode4fun](#)

Returns the series at a given index.

```
>> at ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] 4
== ["cow" "fly" "ant" "bee"]
```

action! **select** and action! **find** [Red-by-example on select](#) [Red-by-example on find](#) [MyCode4fun on select](#) [MyCode4fun on find](#)

Both search a series for a given value. The search goes from left to right, except if `/reverse` or `/last` is used.

When they find a match:

- `select` returns the next element from the series after the match;

```
>> select ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
== "fly"
```

- `find` returns a series that starts in the match and goes all the way to tail.

```
>> find ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
== ["cow" "fly" "ant" "bee"]
```

An example of select:

```
>> movies: [
    title "Gone with the wind"
    star "Scarlet Something"
    quality "pretty good"
    age "very old"
    ]
>> print select movies 'quality
pretty good
```

Is interesting to note that a "shortcut" for select is the path notation:

```
>> print movies/star
Scarlet Something
```

/part

Limits the length of the area to be searched to a given number of elements. In the image below, the search area is highlighted:

```
>> select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
3
== none

>> select/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["fox"]
3
== "cow"
```

```
>> find/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]  3

== none

>> find/part ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"]
4
== ["cow" "fly" "ant" "bee"]
```

/only

Treat a series search value as a block, so it looks for a block inside the search area.

```
>> find/only ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] ["cow"
"fly"]    ;finds nothing
```

```
== none

>> find/only ["cat" "dog" "fox" ["cow" "fly"] "ant" "bee" ] ["cow"
"fly"]    ;finds the block
== [["cow" "fly"] "ant" "bee"]
```

/case

To perform a case sentive search. Upper and lower case become relevant.

/skip

Treats the series as a set of records, where each record has a fixed size. Will only try to match against each first item of such a record.

I highlighted below the "records" in yellow and the match in red:

```
>> find/skip ["cat" "dog"  "fox" "dog"  "dog" "dog" "cow" "dog"
"fly" "dog"  "ant" "dog"  "bee" "dog"] ["dog"]  2
== ["dog" "dog" "cow" "dog" "fly" "dog" "ant" "dog" "bee" "dog"]
```

/same

Uses `same?` as comparator. This comparator returns true if the two objects have the same identity:

```
>> a: "dog" b: "dog"
== "dog"
>> same? a b
== false          ;each is associated with a string with "dog", but not
the same string.
>> b: a
== "dog"
>> same? a b      ;both refer to the very same string
== true
```

/last

Finds the last occurrence of the key, from the tail

```
>> find/last [33 11 22 44 11 12] 11
== [11 12]
```

/reverse

The same as `/last` , but from the current index that can be set, for example by the built-in function `at` .

find/tail

Normally `find` returns the result including the matched item. With `/tail` the returned is the part AFTER the match, similarly to `select`

```
>> find ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] "fly"
== ["fly" "ant" "bee"]

>> find/tail ["cat" "dog" "fox" "cow" "fly" "ant" "bee" ] "fly"
== ["ant" "bee"]
```

find/match

Match always compares the key to the beginning of the series. Also, the result is the part AFTER the match.

```
>> find/match ["cat" "dog" "fox" "cow" "fly" "ant" "bee"] "fly"
== none          ;no match

>> find/match ["cat" "dog" "fox" "cow" "fly" "ant" "bee"] "cat"
== ["dog" "fox" "cow" "fly" "ant" "bee"]                  ;match
```

function! last Red-by-example MyCode4fun

Returns the last value of the series.

```
>> last ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== "bee"
```

function! extract Red-by-example MyCode4fun

Extracts values from a series at given intervals, **returning** a new series.

```
>> extract [1 2 3 4 5 6 7 8 9] 3
== [1 4 7]

>> extract "abcdefghij" 2
== "acegi"
```

/index

Extracts values starting from a given position.

/into

Append the extracted values to a given series.

```
>> newseries: [] ;creates empty series - necessary as extract/into
does not initialize a series
== []

>> extract/into "abcdefghij" 2 newseries
== [#"a" #"c" #"e" #"g" #"i"]

>> extract/into ["cat" "dog" "fox" "cow" "fly" "ant" "bee" "owl"] 2
newseries
== [#"a" #"c" #"e" #"g" #"i" "cat" "fox" "fly" "bee"]
```

action! **copy** Red-by-example MyCode4fun

See Copying chapter.

Sets

native! **union** Red-by-example MyCode4fun

Returns the result of joining two series. Duplicate entries are only included once.

```
>> union [3 4 5 6] [5 6 7 8]
== [3 4 5 6 7 8]
```

/case

Use case-sensitive comparison

/skip

Treat the series as fixed size records.

```
>> union/case [A a b c] [b c C]
```

```
== [A a b c C]
```

With the /skip refinement, only the first element of each group (size given by argument) is compared. If there are duplicate entries, the record of the first series is kept:

```
>> union/skip [a b c c d e e f f] [a j k c y m e z z] 3
== [a b c c d e e f f]

>> union/skip [k b c c d e e f f] [a j k c y m e z z] 3
== [k b c c d e e f f a j k]
```

native! difference Red-by-example

Returns only the elements that are not present in both series.

```
>> difference [3 4 5 6] [5 6 7 8]
== [3 4 7 8]
```

/case

Use case-sensitive comparison

/skip

Treat the series as fixed size records.

native! intersect Red-by-example

Returns only the elements that are present in both series:

```
>> intersect [3 4 5 6] [5 6 7 8]
== [5 6]
```

/case

Use case-sensitive comparison

/skip

Treat the series as fixed size records.

native! unique Red-by-example MyCode4fun

Returns the series removing all duplicates:

```
>> unique [1 2 2 3 4 4 1 7 7]
== [1 2 3 4 7]
```

Allows the refinements:

/skip

Treat the series as fixed size records.

native! **exclude** Red-by-example

Returns a series where the second argument elements are removed from the first argument series.

```
>> a: [1 2 3 4 5 6 7 8]
== [1 2 3 4 5 6 7 8]

>> exclude a [2 5 8]
== [1 3 4 6 7]

>> a
== [1 2 3 4 5 6 7 8]
```

I could not find it in documentation, but I think the returned series is a list of non-repeated elements:

```
>> exclude "my house is a very funny house" "aeiou"
== "my hsvrfn"

>> exclude [1 1 2 2 3 3 4 4 5 5 6 6] [2 4]
== [1 3 5 6]
```

/case

Use case-sensitive comparison

/skip

Treat the series as fixed size records.

< Previous topic　　　　Next topic >

Series "changers"

These commands change the original series:

action! clear Red-by-example MyCode4fun

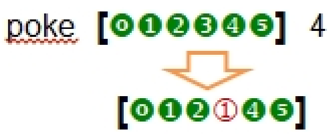
Deletes all elements from the series.

Simply assigning " " (empty string) or zero to a series may not produce the expected results. Red's logic makes it seem to "remember" things in unexpected ways. To really clear it, use `clear`.

```
>> a: [11 22 33 "cat"]
== [11 22 33 "cat"]

>> clear a
== []

>> a
== []
```

action! poke Red-by-example MyCode4fun

Changes the value of a serie's element at the position given by the second argument to the value of the third argument.



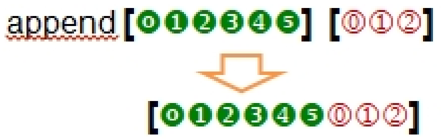

```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> poke x 3 "BULL"
== "BULL"

>> x
== ["cat" "dog" "BULL" "fly"]
```

```
>> s: "abcdefghijklmn"
== "abcdefghijklmn"

>> poke s 4 #"W"
== #"W"

>> s
== "abcWefghijklmn"
```

action! append

Red-by-example MyCode4fun

Inserts the values of the second argument at the end of a series. Changes only the original first series.




```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> append x "HOUSE"
== ["cat" "dog" "mouse" "fly" "HOUSE"]

>> x
== ["cat" "dog" "mouse" "fly" "HOUSE"]
```

```
>> x: ["cat" "dog" "mouse" "fly"]
== ["cat" "dog" "mouse" "fly"]

>> y: ["Sky" "Bull"]
== ["Sky" "Bull"]

>> append x y
== ["cat" "dog" "mouse" "fly" "Sky" "Bull"]

>> x
== ["cat" "dog" "mouse" "fly" "Sky" "Bull"]
```

```
>> append "abcd" "EFGH"
== "abcdEFGH"
```

/part

Limits the number of elements appended to the series.

```
>> append/part ["a" "b" "c"] ["A" "B" "C" "D" "E"]  2
== ["a" "b" "c" "A" "B"]
```

/only

Appends series A with series B, but B goes in as a series (block).

```
>> append/only ["a" "b" "c"] ["A" "B"]
== ["a" "b" "c" ["A" "B"]]
```

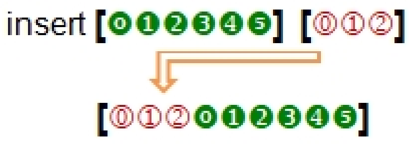
/dup

Appends series A with series B a given number of times. I think it should not be called dup from "duplicate" as it can triplicate, quadrupicate...

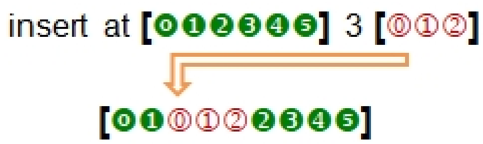
```
>> append/dup ["a" "b" "c"] ["A" "B"] 3
== ["a" "b" "c" "A" "B" "A" "B" "A" "B"]
```

action! insert   [Red-by-example](#)   [MyCode4fun](#)

It is like `append`, but the addition is done at the current entry index (usually the beginning). While `append` **returns** the series from `head`, `insert` **returns** it after the insertion. This allows to chain multiple `insert` operations, or help calculate the length of the inserted part, but `a: insert a something` will not change "a"!




```
>> a: "abcdefgh"
== "abcdefgh"

>> insert a "OOO"
== "abcdefgh"

>> a
== "OOOabcdefgh"
```




```
>> a: "abcdefgh"
== "abcdefgh"

>> insert at a 3 "OOO"
== "cdefgh"

>> a
== "abOOOcdefgh"
```

/part

Inserts only a given number of elements from the second argument.

/only

Allows insertion of blocks as blocks, not their elements.

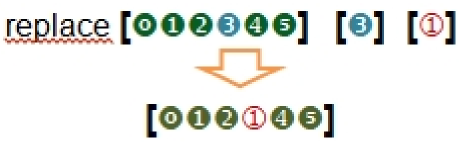
/dup

Allows the insertion to be repeated a given number of times.

```
>> a: "abcdefg"
== "abcdefg"

>> insert/dup a "XYZ" 3
== "abcdefg"

>> a
== "XYZXYZXYZabcdefg"
```

function! replace Red-by-example MyCode4fun

Replaces an element of the series.




```
>> replace ["cat" "dog" "mouse" "fly" "Sky" "Bull"] "mouse" "HORSE"
```

```
== ["cat" "dog" "HORSE" "fly" "Sky" "Bull"]
```

/all

Replaces all ocurrences.

```
>> a: "my nono house nono is nono nice"
== "my nono house nono is nono nice"

>> replace/all a "nono " ""
== "my house is nice"
```

action! sort [Red-by-example](#) [MyCode4fun](#)

Sorts a series.


```
>> sort [8 4 3 9 0 1 5 2 7 6]
== [0 1 2 3 4 5 6 7 8 9]
```

```
>> sort "sorting strings is useless"
== " eeggiiilnnorrsssssssttu"
```

/case

Perform a case-sensitive sort.

/skip

Treat the series as fixed size records.

/compare

Comparator offset, block or function. (?)

/part

Sort only part of a series.

/all

Compare all fields. (?)

/reverse

Reverse sort order.

/stable

Stable sorting. (?)

action! remove [Red-by-example](#) [MyCode4fun](#)

Removes the first value of the series.


```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> remove s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

/part

Removes a given number of elements.


```
>> s: "abcdefghij"
== "abcdefghij"

>> remove/part s 4
== "efghij"
```

Notice that you can do the same with `remove at [0 1 2 3 4 5] 2`.

native! remove-each [Red-by-example]

Like `foreach`, it sequentially executes a block for each element of a series. If the block returns `true`, it removes the element from the series:

```
Red []

a: ["dog" 23 3.5 "house" 45]
remove-each i a [string? i] ;removes all strings
print a
```

```
23 3.5 45
```

```
Red []

a: "   my house in the middle of our street"
remove-each i a [i = #" "] ;removes all spaces
print a
```

```
myhouseinthemiddleofourstreet
```

action! take [Red-by-example] [MyCode4fun]

Removes the FIRST element of a series and gives this first element as **return**.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take s
== "cat"

>> s
== ["dog" "fox" "cow" "fly" "ant" "bee"]
```

/last

Removes the LAST element of a series and gives this last element as **return**.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/last s
== "bee"
```

```
>> s
== ["cat" "dog" "fox" "cow" "fly" "ant"]
```

`take/last` and `append` can be used to perform stack (queue) operations.

/part

Removes a given number of elements from the start of the series and gives them as **return**.

```
>> s: ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 3
== ["cat" "dog" "fox"]

>> s
== ["cow" "fly" "ant" "bee"]
```

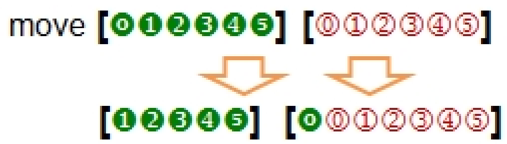
/deep

Documentation says "Copy nested values". I could not figure it out.

action! move

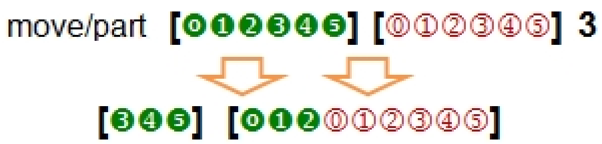
Red-by-example MyCode4fun

Moves one or more elements from the first argument into the second argument. Changes both original arguments.



/part

To move more than one element.




```
>> a: [a b c d]
== [a b c d]

>> b: [1 2 3 4]
== [1 2 3 4]
```

```
>> move a b
== [b c d]

>> a
== [b c d]

>> b
== [a 1 2 3 4]

>> move/part a b 2
== [d]

>> a
== [d]

>> b
== [b c a 1 2 3 4]
```

`move` can be used combined with other built-in functions (commands) to move things inside a single series. For example:

```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]

>> move a tail a
== [2 3 4 5 1]

>> move/part a tail a 3
== [5 1 2 3 4]
```

action! change Red-by-example MyCode4fun

Changes the first elements of a series and returns the series after the change. Modifies the first original series, not the second.


```
>> a: [1 2 3 4 5]
== [1 2 3 4 5]

>> change a [a b]
== [3 4 5]
```

```
>> a
== [a b 3 4 5]
```

/part
Limits the amount to change to a given length.

/only
Changes a series as a series.

/dup
Repeats the change a specified number of times

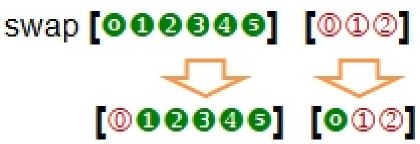
function! alter [Red-by-example](#) [MyCode4fun](#)

Either appends or removes an element from a series. If `alter` does NOT find the element in a series, it appends it and returns `true`. If it finds the element, removes it and returns `false`.

```
>> a: ["cat" "dog" "fly" "bat" "owl"]
== ["cat" "dog" "fly" "bat" "owl"]

>> alter a "dog"
== false

>> a
== ["cat" "fly" "bat" "owl"]

>> alter a "HOUSE"
== true

>> a
== ["cat" "fly" "bat" "owl" "HOUSE"]
```

action! swap [Red-by-example](#)

Swaps the first elements of two series. **Returns** the first series, but changes both:




```
>> a: [1 2 3 4]   b: [a b c d]
```

```
>> swap a b
== [a 2 3 4]

>> a
== [a 2 3 4]

>> b
== [1 b c d]
```

With `find` , for example, it can be used to swap any element of two series and even elements within a single series:

```
>> a: [1 2 3 4 5] b: ["dog" "bat" "owl" "rat"]
== ["dog" "bat" "owl" "rat"]

>> swap find a 3 find b "owl"
== ["owl" 4 5]

>> a
== [1 2 "owl" 4 5]

>> b
== ["dog" "bat" 3 "rat"]
```

action! **reverse** Red-by-example MyCode4fun

Reverses the order of the elements of a series:

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

>> reverse "abcde"
== "edcba"
```

/part limits the reverse to the number of elements of the argument:

```
>> reverse/part "abcdefghi" 4
== "dcbaefghi"
```

< Previous topic Next topic >

Copying

WARNING FOR BEGINNERS: If you are assigning the value of a word (variable) to another word (variable) in Red, COPY IT!

```
>> var1: var2          ;only if you are sure about it

>> var1: copy var2     ;may save you hours of debugging
```

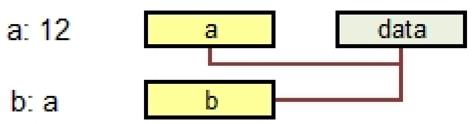
action! **copy** Red-by-example MyCode4fun

Assigns a copy of the data to a new word.

It may be used to copy series and objects.

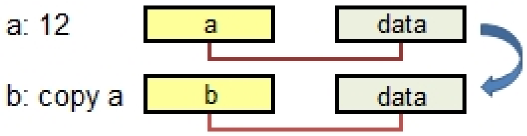
It is **not** used on single items such as: integer! float! char! etc. For these, we can simply use the colon.

First lets look at a simple assignment:




```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> b: s
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 4
== ["cat" "dog" "fox" "cow"]

>> b
== ["fly" "ant" "bee"]            ;b changes!!
```

Now with `copy`:




```
>> s: [ "cat" "dog" "fox" "cow" "fly" "ant" "bee" ]
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> b: copy s
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]

>> take/part s 4
== ["cat" "dog" "fox" "cow"]

>> b
== ["cat" "dog" "fox" "cow" "fly" "ant" "bee"]
```

If you have a nested series (e.g. a block within your series) `copy` does not change the reference to these nested series. If you want to create an independent copy in this case, you must use the refinement `/deep` to create a "deep" copy.

/part

Limits the length of the result, where length is a `number!` or `series!`

```
>> a: "my house is a very funny house"
>> b: copy/part a 8
== "my house"
```

/types

Copies only specific types of non-scalar values.

/deep

Copies nested values, as mentioned above.

< Previous topic   Next topic >

Looping

native! loop Red-by-example MyCode4fun

Executes a block a given number of times.

```
Red[]

loop 3 [print "hello!"]
```

```
hello!
hello!
hello!
>>
```

native! repeat Red-by-example

`repeat` is the same as `loop`, but it has an index that gets incremented each loop

```
Red[]

repeat i 3 [print i]
```

```
1
2
3
>>
```

native! forall Red-by-example MyCode4fun

Executes a block as it moves forward in a series.

```
Red[]

a: ["china" "japan" "korea" "usa"]
forall a [print a]
```

```
china japan korea usa
japan korea usa
korea usa
```

```
usa
>>
```

native! foreach [Red-by-example] [MyCode4fun]

Executes a block for each element of the series.

```
Red[]

a: ["china" "japan" "korea" "usa"]
foreach i a [print i]
```

```
china
japan
korea
usa
>>
```

native! while [Red-by-example] [MyCode4fun]

Executes a block while a condition is true.

```
Red[]

i: 1
while [i < 5] [
      print i
      i: i + 1
]
```

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

native! until [Red-by-example] [MyCode4fun]

Evaluates a block until the block returns a `true` value.

```
Red[]

i: 4
until [
      print i
      i: i - 1
```

```
    i < 0 ;  <= condition
]
```

```
4
3
2
1
0
>>
```

native! **break** Red-by-example MyCode4fun

Forces an exit from the loop.

/return

Forces the exit and sends a given value, like a code or a message, as a return value.

native! **forever** Red-by-example MyCode4fun

Creates a loop that never ends.

< Previous topic Next topic >

Branching

native! if Red-by-example MyCode4fun

Executes a block if a test is `true`.

if <test> [block **]**

```
>> if 10 > 4 [print "large"]
large
```

Remember from the Datatypes chapter that everything that is not `false` , `off` or `no` is considered `true`:

```
>> if "house" [print "It's true!"]
It's true!

>> if 0 [print "It's true!"]
It's true!

>> if [] [print "It's true!"]
It's true!

>> if [false] [print "It's true!"]   ;bizarre!
It's true!
```

native! unless Red-by-example MyCode4fun

Same as `if not`. Executes block only if a test is `false`.

unless <test> [block (if test `false`) **]**

```
>> unless 10 > 4 [print "large"]
== none

>> unless 4 > 10 [print "large"]
large
```

native! either [Red-by-example] [MyCode4fun]

A new name for the classic if-else. Executes the first block if the test is `true` or executes the second block if the test is `false`.

either <test> [`true` block**] [**`false` block**]**

```
>> either 10 > 4 [print "bigger"] [print "smaller"]
bigger

>> either 4 > 10 [print "bigger"] [print "smaller"]
smaller
```

native! switch [Red-by-example] [MyCode4fun]

Executes the block correspondent to a given value.

```
Red[]

switch 20 [
      10 [print "ten"]
      20 [print "twenty"]
      30 [print "thirty"]
]
```

```
twenty
```

/default

If the program does not find a match, executes a *default* block.

```
Red[]

switch/default 15 [
      10 [print "ten"]
      20 [print "twenty"]
      30 [print "thirty"]
][ print "none of them"]      ;default block
```

```
none of them
```

native! case [Red-by-example] [MyCode4fun]

Makes a series of tests, executing the block corresponding to the **first** `true` test.

```
Red[]

case [
      10 > 20 [print "not ok!"]
      20 > 10 [print "this is it!"]
      30 > 10 [print "also ok!"]
]
```

```
this is it!
```

/all

Executes **all** the `true` tests.

```
Red[]

case/all [
      10 > 20 [print "not ok!"]
      20 > 10 [print "this is it!"]
      30 > 10 [print "also ok!"]
]
```

```
this is it!
also ok!
```

native! catch & throw [Red-by-example]

Catch and throw may be used to create complex control structures. In its simplest form, `catch` receives a **return** from one of many throws:

```
Red[]

a: 10
print catch [
      if a < 10 [throw "too small"]
      if a = 10 [throw "just right"]
      if a > 10 [throw "too big"]
]
```

```
just right
```

catch/name

catches a named throw. Really deserves an example, hope to make one soon...

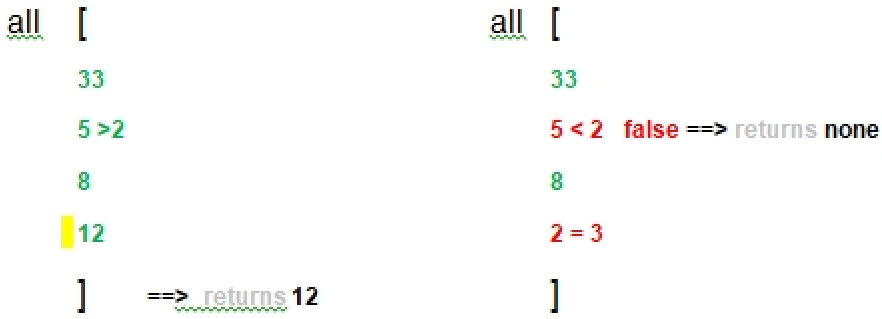
throw/name

throws a named catch.

Boolean branching

native! all Red-by-example MyCode4fun

Evaluates all expressions in a block. If one evaluation returns `false`, it returns `none`, otherwise returns the result of the last evaluation.




```
>> john: "boy"
== "boy"

>> alice: "girl"
== "girl"

>> all [john = "boy" alice = "girl" 10 + 3]   ;all true, the last
evaluation is returned.
== 13

>> all [john = "boy" alice = "boy" 10 + 3]    ; alice = "boy" is
false!
== none

>> if all [john = "boy" alice = "girl"] [print "It' all true"]
It' all true
```

native! any Red-by-example MyCode4fun

Evaluates each expression in a block in and returns the first resulting value that is not `false`. If all resulting values are `false` it returns `none`.

```
any [
    3 = 5
    5 < 2
    8     ==> returns 8
    12
]
```

```
any [
    3 = 5
    5 < 2
    9 = 3
    2 = 3
]   ==> returns none
```

```
>> john: "boy"
== "boy"

>> alice: "girl"
== "girl"

>> any [john = "girl"  alice = "girl"  10 + 3]  ;alice = "girl" is not
false: return it!
== true

>> any [john = "girl"  10 + 3  5 > 2]                 ; 10 + 3 is not
false: return it!
== 13

>> if any [john = "girl" alice = "girl"] [print "Something is true
here"]
Something is true here
```

< Previous topic   Next topic >

String and text manipulation

Note: in the examples, some output lines of the console were removed for clarity.

function! split Red-by-example MyCode4fun

Returns a block (a series) containing the pieces of a string that are separated by a delimiter. Does not change original block. The delimiter is given as an argument. `split` is particularly useful to the parse dialect and to analyze and manipulate text files.

```
>> s: "My house is a very funny house"
>> split s " "                                          ;every space is
a delimiter.
== ["My" "house" "is" "a" "very" "funny" "" "" "" ""
"house"] ;result is a series with 11 elements.

>> s: "My house ; is a very ; funny house"
>> split s ";"                                          ;split happens
at the semi-colons.
== ["My house " " is a very " " funny house"]       ;result is a
series with 3 elements.
```

removing characters: action! trim Red-by-example MyCode4fun

The word `trim` with no refinements removes white space (tabs and spaces) from the beginning and end of a string! (it also removes `none` from a block! or object!). The value of the argument is altered. It has a refinement to remove specific characters. It returns the trimmed series **and changes the original series**.

Refinements:

/head - Removes only from the head.
/tail - Removes only from the tail.
/auto - Auto indents lines relative to first line.
/lines - Removes all line breaks and extra spaces.
/all - Removes all whitespace (but not line breaks).
/with - Same as /all, but removes characters in a 'with' argument we supply. It must be one of: char! string! or integer!

```
>> e: " spaces before and after "
>> trim e
== "spaces before and after"
```

trim leading spaces:

```
>> e: " spaces before and after "
>> trim/head e
== "spaces before and after "
```

trim trailing spaces:

```
>> e: " spaces before and after "
>> trim/tail e
== " spaces before and after"
```

trim specific characters:

```
>> d: "our house in the middle of our street"
>> trim/with d " "
== "ourhouseinthemiddleofourstreet"
```

```
>> a: "house"
>> trim/with a "u"
== "hose"
```

the opposite of trim: function! pad Red-by-example

`pad` expands the string to a given size by adding spaces. The default is to add spaces to the right, but with the refinement `/left` , spaces are added to the beginning of the string. **Changes the original string**, beware.

```
>> a: "House"
>> pad a 10
== "House "
```

```
>> pad/left a 20
== "               House "
```

text concatenation: function! rejoin Red-by-example MyCode4fun

```
>> a: "house" b: " " c: "entrance"
>> rejoin [a b c]
== "house entrance"
```

or, using `append` - this changes the original series

```
>> append a c
== "house entrance"
```

```
>> a: "house" b: " " c: "entrance"

>> append a c
== "houseentrance"

>> append a append b c
== "houseentrance entrance"       ; "a" was changed to
"houseentrance" in the last manipulation
```

turning a series into text: action! form Red-by-example MyCode4fun

`form` **returns** a series as a string, removing brackets and adding spaces between elements. `form` was briefly seen in the Accessing and formating data chapter.

```
>> a: ["my" "house" 23 47 4 + 8 ["a" "bee" "cee"]]
>> form a
== "my house 23 47 4 + 8 a bee cee"
```

/part

The refinement `/part` limits the number of characters of the created string.

```
>> a: ["my" "house" 23 47 4 + 8 ["a" "bee" "cee"]]
```

```
>> form/part a 8
== "my house"
```

string length: action! length? Red-by-example MyCode4fun

```
>> f: "my house"
>> length? f
== 8
```

left part of string:

using **copy/part** :

```
>> s: "nasty thing"
>> b: copy/part s 5
== "nasty"
```

right part of string:

using `at` :

```
>> s: "nasty thing"
>> at tail s -5
== "thing"
```

using `remove/part` - this changes the original series, beware!

```
>> s: "nasty thing"
>> remove/part s 6
== "thing"
```

middle part of string:

using `copy/part` and `at`:

```
>> a: "abcdefghijkl"
>> copy/part at a 4 3
== "def"
```

insert strings:

at the beginning, using `insert`:

```
>> s: "house"
>> insert s "beautiful "

>> s
== "beautiful house"
```

at the end, using `append`:

```
>> s: "beautiful"
>> append s " day"
== "beautiful day"
```

in the middle, using `insert` `at`:

```
>> s: "nasty thing"
>> insert at s 7 "little "

>> s
== "nasty little thing"
```

native! lowercase Red-by-example MyCode4fun

Changes the original string, beware.

```
>> a: "mY HoUse"
>> lowercase a
== "my house"
```

/part

```
>> a: "mY HoUse"
>> lowercase/part a 2
```

```
== "my HoUse"
```

native! uppercase Red-by-example MyCode4fun

Changes the original string, beware.

```
>> a: "mY HoUse"
>> uppercase a
== "MY HOUSE"
```

/part

```
>> a: "mY HoUse"
>> uppercase/part a 2
== "MY HoUse"
```

< Previous topic Next topic >

Printing special characters

These were taken from Rebol's documentation, but I have tested most of them in Red and they seem to work:

Control characters:

Character	Definition
#"^(null)" or #"^@"	null (zero)
#"^(line)", or #"^/"	new line
#"^(tab)" or #"^-"	horizontal tab
#"^(page)"	new page (and page eject)
#"^(esc)"	escape
#"^(back)"	backspace
#"^(del)"	delete
#"^^"	caret character
#"^""	quotation mark
#"(0)" to #"(FFFF)"	hex forms of characters

Special characters for within strings:

Character	Function
^"	prints a " (quote)
^}	inserts a } (closing brace)
^^	inserts a ^ (caret)
^/	starts a new line
^(line)	starts a new line
^-	inserts a tab
^(tab)	inserts a tab
^(page)	starts a new page (?)
^(letter)	inserts control-letter (A-Z)
^(back)	erases one character back
^(null)	inserts a null character
^(esc)	inserts an escape character
^(XX)	inserts an ASCII character by hexadecimal (XX) number

< Previous topic Next topic >

Time and timing

native! wait [Red-by-example] [MyCode4fun]

Stops the execution for the number of seconds given as argument.

- Note: as of November 2017, the GUI Console does not work well with `wait`.

native! now [Red-by-example] [MyCode4fun]

Returns date and time:

```
>> now
== 12-Dec-2017/19:24:41-02:00
```

Refinements

/time - Returns time only. time!

```
>> now/time
== 21:42:41
```

/precise - High precision time. date!

```
>> now/precise
== 2-Apr-2018/21:49:04.647-03:00
```

```
>> a: now/time/precise
== 22:05:46.805       ;a is a time!

>> a/hour
== 22                 ;hour is an integer!

>> a/minute
== 5                  ;minute is an integer!

>> a/second
== 46.805             ;second is a float!
```

This script creates a delay of 300 miliseconds (0.3 seconds):

```
Red []
thismoment: now/time/precise
print thismoment
while [now/time/precise < (thismoment + 00:00:00.300)][]
print now/time/precise
```

```
21:51:58.173
21:51:58.473
```

/year - Returns year only. integer!

```
>> now/year
== 2018
```

/month - Returns month only. integer!

```
>> now/month
== 4
```

/day - Returns day of the month only. integer!

```
>> now/day
== 2
```

/zone - Returns time zone offset from UCT (GMT) only. time!

```
>> now/zone
== -3:00:00
```

/date - Returns date only. date!

```
>> now/date
== 2-Apr-2018
```

/weekday - Returns day of the week as integer! (Monday is day 1).

```
>> now/weekday
== 1
```

/yearday - Returns day of the year (Julian). integer!

```
>> now/yearday
== 92
```

/utc - Universal time (no zone). date!

```
>> now/utc
== 3-Apr-2018/0:53:50
```

VID DLS rate Red-by-example MyCode4fun

Timing may also be achieved with VID dialect (GUI) using the `rate` facet.

< Previous topic Next topic >

Error handling

---

function! attempt Red-by-example MyCode4fun

Evaluates a block and returns the result or `none` if an error occur.

```
>> attempt [a: 10 b: 9]     ;first lets try with no errors...
== 9

>> a
== 10                       ;... no problems here!

>> attempt [a: 10 nosyntax]       ;nosyntax has no value: ERROR!
== none

>> attempt [divide 100 0]
== none
```

native! try Red-by-example MyCode4fun

Tries to evaluate a block. Returns the value of the block, but if an `error!` occurs, the block is abandoned, and an
error value is returned.
To identify a block that generates an error without actually having the error output printed, we use the function `error?`.

You may ask why not use `attempt` instead of `error?` & `try`. I think the answer is that the `error?` & `try` combination returns `true` and `false`, instead of `none` or an evaluation. This is useful when used inside other structures.

```
>> error? [nosyntax]
== false                      ;nosyntax has no value and it generates an
error,
                              ;but only if evaluated. In itself, is not a
error! datatype.

>> try [nosyntax]
*** Script Error: nosyntax has no value
```

```
*** Where: try
*** Stack:            ; just "try" does not work, you get an
error!!

>> error? try [nosyntax]
== true               ;OK!

>> error? try [divide 100 0]
== true
```

native! catch and native! throw Red-by-example

These are used to handle errors, but I could not figure how. Does not seem to be a beginner's issue.

< Previous topic   Next topic >

Files

Path, directories and files

Path names

File paths are written with a percent sign (%) followed by a sequence of directory names that are each separated by a forward slash (/). In Windows, Red makes all the conversions from backslashes to forward slashes, you don't have to worry.

Just remembering:

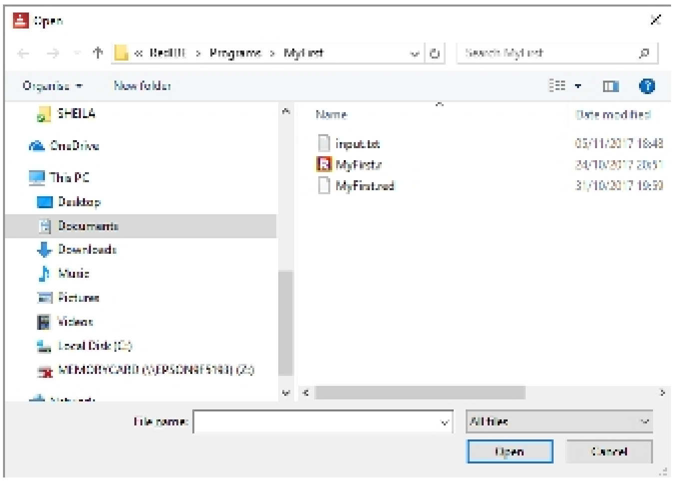
- `/` is the root of the current drive;
- `./` is the current directory;
- `../` is the parent of the current directory;
- file paths that do not begin with a forward slash (/) are relative paths;
- to refer to Window's often used "C" drive you should use: `%/C/docs/file.txt`
- absolute paths should be avoided to ensure machine-independent scripts;

A graphic file selector:

function! **request-file** Red-by-example MyCode4fun

`request-file` opens a graphic file selector and returns the full file path as a `file!`

```
>> request-file
```



```
== %/C/Users/André/Documents/RED/myFirstFile.txt
```

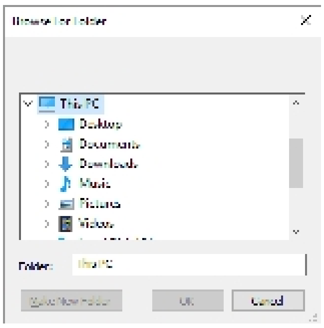
Refinements

/title - window title. Example: `request-file/title "My file is:"`
/file - Default file name or directory. Example: `request-file/file %"MyFile.txt"`
/filter -Supply a block of filters consisting of pairs of filter names, and the actual filters. Example: `request-file/filter ["executables" "*.exe" "text files" "*.txt"]`
/save - File save mode. Example with filters: `request-file/save/filter ["executables" "*.exe" "text files" "*.txt"]`
/multi - Allows multiple file selection, returned as a block.

A graphic directory selector:

function! **request-dir** Red-by-example MyCode4fun

`request-dir` opens a graphic directory selector and returns the full file path as a `file!`



```
== %/C/Users/André/Documents/RED/
```

Refinements

/title => Window title.
/dir => Set starting directory.
/filter => TBD: Block of filters (filter-name filter).
/keep => Keep previous directory path.
/multi => TBD: Allows multiple file selection, returned as a block.

Deleting a file:

action! **delete** Red-by-example MyCode4fun

Deletes a file and **returns** `true` if successful, `false` otherwise.

```
>> delete %file.txt
== true
```

Getting the size of a file:

native! **size?** Red-by-example

Returns the number of bytes a file has, or `none` if file does not exist.

```
>> size? %myFirstFile.txt
== 37
```

Other directory and path functions:

cd or change-dir - Changes the current directory.

dir, ls or list-dir - Lists the contents of a given directory. If no argument is given, lists the current directory.

dir? - Returns `true` if the supplied name is a valid file `path!`,
otherwise returns `false`.

dirize - Turns its argument into a valid directory.
The argument can be of `file!` `string!` `url!`.
Effectively dirize only appends a trailing / if needed.

exists? - Returns `true` if its argument is an existing `path!`
or `false` otherwise.

file? - Returns true if its argument is a `file!`.

get-current-dir - Returns the current directory the program is using.

get-path? - Returns `true` if its argument is a `get-path!`

path? - Returns `true` if its argument is a `path!`

split-path - Splits a `file!` or `url!` path. Returns a block containing path and target.

suffix? - Returns the sufix of a file. e.g: exe, txt

what-dir - Returns the current directory path as a `file!` value.

to-red-file - Converts a local file system path to Red's standard machine independent path format.

to-local-file - Converts standard, system independent Red file paths to the file format used by the local operating system.

clean-path - Cleans-up '.' and '..' in a path and returns the cleaned path.

red-complete-file

red-complete-path

set-current-dir

< Previous topic   Next topic >

Writing to files

Writing to a text file:

action! **write** Red-by-example MyCode4fun

Writes to a file, creating it if it doesn't exist.

```
>> write %myFirstFile.txt "Once upon a time..."
```

Appending a text file:

/append

If you just `write` again to the file created above, it will be overwritten. If you want to add more text to it (append it):

```
>> write/append %myFirstFile.txt "there was a house."
```

Your file now has "Once upon a time...there was a house" in it.

Writing a series to a file making each element a line:

/lines

```
>> write/lines %mySecondFile.txt ["First line;" "Second line;"
"Third line."]
```

Appending full lines:

```
>> write/append/lines %mySecondFile.txt ["Fourth line;" "Fifth
line;" "Sixth line."]
```

Your file now looks like this:

```
First line;
Second line;
Third line.
Fourth line;
Fifth line;
Sixth line.
```

Notice that you could have written `write/lines/append`. The order of the refinements makes no difference.

Replacing characters in a file:

To replace characters in a text file, starting at n+1 position, use `write/seek %<file> <n>` :

```
>> write/seek %myFirstFile.txt "NEW TEXT" 5
```

Now the first file has: "Once NEW TEXTime...there was a house."

Write refinements:

/binary => Preserves contents exactly.

/lines => Write each value in a block as a separate line.

/info =>

/append => Write data at end of file.

/part => Partial write a given number of units.

/seek => Write at a specific position.

/allow => Specifies protection attributes.

/as => Write with the specified encoding, default is 'UTF-8.

function! **save** Red-by-example MyCode4fun

Saves a value, block, or other data to a file, URL, binary, or string.

Difference between write and save:

```
>> write %myFourthFile.txt [11 22 "three" "four" "five"]
```

Your file now has: `[11 22 "three" "four" "five"]`

```
>> save %myFourthFile.txt [11 22 "three" "four" "five"]
```

Your file now has `11 22 "three" "four" "five"`

An important use of `save` is to simplify the saving of Red scripts that can be interpreted using the action `do` :

```
>> save %myProgram.r [Red[] print "hello"]
>> do %myProgram.r
hello
```

`do`, `load` and `save` are better understood if you think of Red's console as the screen of some old computer from the 80's running some variation of basic language. You can `load` your program, `save` it, or `do` (execute) it.

< Previous topic Next topic >

Reading files

Reading files as text:

action! **read** Red-by-example MyCode4fun

```
>> a: read %mySecondFile.txt
== {First line;^/Second line;^/Third line.^/Fourth line;^/Fifth li
```

Now the word (variable) "a" has the entire content of the file:

```
>> print a
First line;
Second line;
Third line.
Fourth line;
Fifth line;
Sixth line.
```

Reading files as series where every line is an element:

Notice that, so far, the word "a" above is just text with newlines. If you want to read the file as a `series!` having each line as an element, you should use `read/lines`:

```
>> a: read/lines %mySecondFile.txt
== ["First line;" "Second line;" "Third line." "Fourth line;"...

>> print pick a 2
Second line;
```

Read refinements:

- **/part** => Partial read a given number of units (source relative).
- **/seek** => Read from a specific position (source relative).
- **/binary** => Preserves contents exactly.
- **/lines** => Convert to block of strings.
- **/info** =>
- **/as** => Read with the specified encoding, default is 'UTF-8.

function! load [Red-by-example](#) [MyCode4fun](#)

Reading files as a series where every word (separated by space) is an element:

In this case, you should use `load` instead of `read`:

```
>> a: load %mySecondFile.txt
== [First line Second line Third line.
          Fourth line Fifth...

>> print pick a 2
line
```

Reading and writing binary files:

To read or write a binary file such as an image or a sound, you should use the `/binary` refinement. The following code loads a bitmap image to variable a and saves that image with another name:

```
>> a: read/binary %heart.bmp
== #{
424D66070000000000003600000028000001E00000014000000010...
>> write/binary %newheart.bmp a
```

Load refinements:

/header => TBD.
/all => Load all values, returns a block. TBD: Don't evaluate Red header.
/trap => Load all values, returns [[values] position error].
/next => Load the next value only, updates source series word.
/part =>
/into => Put results in out block, instead of creating a new block.
/as => Specify the type of data; use NONE to load as code.

< Previous topic Next topic >

Functions

Functions must be declared before they are used and so must be written on top of your program. However, this is not required if a function is called from within another function.

native! func Red-by-example MyCode4fun

Variables inside a function created with `func` are **global**. They are the seen by the entire program.

A function is created with `func` as follows:

<function name>: func [<argument1> <argument2> ... <argument n>] [<actions performed on arguments>]

```
Red []
mysum: func [a b] [a + b]
print mysum 3 4
```

```
7
```

Demonstrating that variables are **global**:

```
Red []
mysum: func [a b] [
      mynumber: a + b
      print mynumber
]
mynumber: 20
mysum 3 4
print mynumber
```

```
7
7
```

native! function Red-by-example MyCode4fun

`function` makes its variables **local**, i.e. it hides (shades) the variables inside it from the rest of the program.

Same program as above, only using `function` instead of `func`:

```
Red []
mysum: function [a b] [
      mynumber: a + b
      print mynumber
]
mynumber: 20
mysum 3 4
print mynumber
```

Different results:

```
7
20
```

Forcing variables to be global with /extern refinement:

```
Red []
myfunc: function [/extern a b] [
      a: 22
      b: 33
]
a: 7
b: 9
myfunc
print a
print b
```

```
22
33
```

Defining the argument type:

You can force your arguments to be of a certain `datatype`:

```
Red []
mysum: function [a [integer!] b[integer!]] [print a + b]
print mysum 3.2 4 ; oops! 3.2 is a float!
```

```
*** Script Error: mysum does not allow float! for its a argument
*** Where: mysum
*** Stack: mysum
```

You may allow multiple `datatypes`:

```
Red []
mysum: function [a [integer! float!] b[integer!]] [print a + b]
print mysum 3.2 4
```

```
7.2
```

Or use an upper class of `datatypes`:

```
Red []
mysum: function [a [number!] b[number!]] [print a + b]
```

```
print mysum 3.2 4
```

```
7.2
```

Documenting your functions

A description of your function may be included by placing a string inside the argument block before the arguments. Also, you may also add explanations about your arguments as a string after the restriction block. These descriptions and explanations will show when you ask for help on your own function.

```
Red [ ]

sum: func [
      "Adds two integers, floats or pairs"
      a [integer! float! pair!] "Fisrt number"
      b [integer!] "Next Number - must be integer"
][
      a + b
]

print "This is my function's help:"
print ? sum
```

```
This is my function's help:
USAGE:
     SUM a b

DESCRIPTION:
     Adds two integers, floats or pairs.
     SUM is a function! value.

ARGUMENTS:
     a            [integer! float! pair!] "Fisrt number."
     b            [integer!] "Next Number - must be integer."


>> sum 5 8,4
*** Script Error: sum does not allow float! for its b argument
*** Where: sum
*** Stack: sum

>> sum 2x3 4
== 6x7
```

Returning values from functions: native! return Red-by-example MyCode4fun

The **return** value of a function is either the last value evaluated by the function or one explicitly determined by the word `return`:

Last evaluation example:

```
Red []
```

```
myfunc: function [] [
      8 + 9
      3 + 3
      print "got here"   ; this executes
      10 + 5             ; this is returned
]
print myfunc
```

```
got here
15
```

`return` example:

```
Red []
myfunc: function [] [
      8 + 9
      return 3 + 3                 ; this is returned
      print "never got here"   ; NOT executed
      10 + 5
]
print myfunc
```

```
6
```

Creating your own refinements:

You can create refinements to you functions, like the native refinements of Red: <myfunction>/<myrefinement>. The refinements are boolean values that are checked by the function:

```
Red []
myfunc: function [a /up   b /down   c] [
      if up [print a + b]
      if down [print a - c]
]
myfunc/up 10 3
myfunc/down 10 3
```

```
13
7
```

Note that arguments are **not** mandatory for refinements.

A more complete example:

```
Red [ ]

sum: func [
      "Adds two integers, floats or pairs"
      a [integer! float! pair!] "Fisrt number"
      b [integer!] "Next Number - must be integer"
      /average "Average instead of add"
][
      either average [a + b / 2] [a + b]
]
```

```
print "This is my function help:"
print ? sum
print
print "Using add with 10 and 16:"
prin "sum = " print sum 10 16
prin "sum/average = " print sum/average 10 16
```

```
This is my function help:
USAGE:
     SUM a b

DESCRIPTION:
     Adds two integers, floats or pairs.
     SUM is a function! value.

ARGUMENTS:
     a             [integer! float! pair!] "Fisrt number."
     b             [integer!] "Next Number - must be integer."

REFINEMENTS:
     /average      => Average instead of add.


Using add with 10 and 16:

sum = 26
sum/average = 13
```

Assigning functions to words (variables)

To assign a function to a variable (a word) you must precede the function with a colon: <word>: :<function>

```
>> mysum: func [a b] [a + b]
== func [a b][a + b]

>> a: :mysum
== func [a b][a + b]

>> a 3 9
== 12
```

native! does [Red-by-example](#) [MyCode4fun](#)

If your function just do something with **no arguments** and **no local variables**, create it with the word `does` :

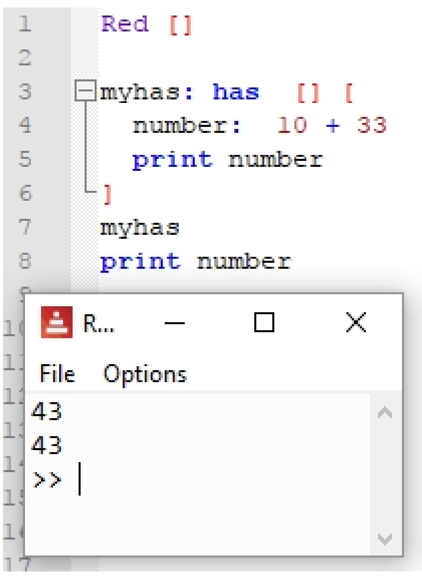
```
Red []
greeting: does [
      print "Hello"
```

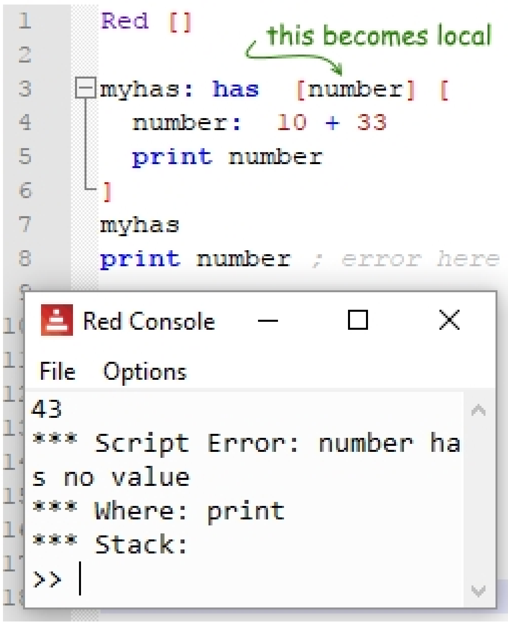
```
        print "Stranger"
]

greeting
```

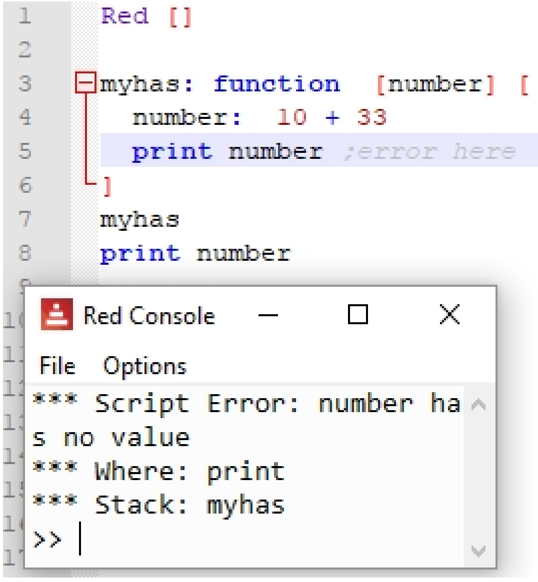
```
Hello
Stranger
```

native! **has** [Red-by-example] [MyCode4fun]

If your routine uses no external arguments but has local variables, use the word `has`. `has` turns the argument into a local variable. Compare the three programs below. The first uses `has` with no argument, hence "number"is a global variable. The second gives "number" as argument, making it local. And the third shows that a function with argument **need** that argument to be sent by the calling event.







native! exit [Red-by-example](#) [MyCode4fun](#)

Exits a function without returning any values.

[< Previous topic](#) [Next topic >](#)

Objects

An object is a container that groups data and/or functions, usually (always?) assigned to a word (variable) . To access an object's attribute in Red, we use a slash (/) as a separator. This is unusual as most languages use a dot, but once you get used to it, it seems more intuitive as it is similar to a path.

Creating an object:

action! make object! , function! context and function! object Red-by-example

You may use `make object!` , `object` or `context` to create an object. They are the same command. `object` and `context` are just shortcuts to `make object!`.

```
Red []
myobject: object [
       x: 10
       y: 20
       f: function [a b] [a + b]
       name: none
       tel: none
]
myobject/name: "Dimitri"
myobject/tel: #3333-3333
print myobject/x
print myobject/y
print myobject/f 3 5
print myobject/name
print myobject/tel
```

```
10
20
8
Dimitri
3333-3333
```

Evaluation is done only when creating an `object!` (constructor code). Notice that the `print` command in the code below is not executed when the object is accessed:

```
>> myobject: object [print "hello" a: 1 b: 2]
hello
== make object! [
    a: 1
```

```
    b: 2
]

>> myobject/a
== 1
```

Self reference:

When an object must do a reference to itself, we use a special keyword named `self` :

```
Red []
myobject: object [
      x: 10
      y: 20
      f: function [a b] [a + b]
      autoanalisys: does [print self]
]

myobject/autoanalisys
```

```
x: 10
y: 20
f: func [a b][a + b]
autoanalisys: func [][print self]
```

Cloning an object:

Simply assigning an object to another creates a "link" to the same data. If the original changes, the second also changes:

```
>> a: object [x: 10]        ;lines of the console deleted for the
sake of clarity.
>> b: a                     ;lines of the console deleted for the
sake of clarity.
>> a/x: 20
== 20

>> b/x
== 20                       ;changed too!
```

To make a true copy of an object, we use the word `copy`:

```
>> a: object [x: 10]        ;lines of the console deleted for the
sake of clarity.
>> b: copy a                ;lines of the console deleted for the
sake of clarity.
>> a/x: 20
```

```
== 20

>> b/x
== 10                        ;NO change! b is a true copy.
```

Prototyping (Inheritance)

Any object can serve as a prototype for making new objects. If we want to create a new object that inherits the first object , we use: `make` **<original object> <new specifications>**:

```
Red []
a: object [x: 3]
b: make a [y: 12]
print b
```

```
x: 3
y: 12
```

Another example:

```
Red []
myobject: object [
      name: none
      tel: none
]
myobject/name: "Dimitri"
myobject/tel: #3333-3333

myextended-object: make myobject [
      gender: "male"
      zip_code: 666
]

myextended-object/name: "Igor"
myextended-object/tel: #9996-9669

prin myobject/name prin "  tel:" print myobject/tel
prin myextended-object/name prin "  tel:" prin myextended-object/tel
prin "  gender:" prin myextended-object/gender prin "  zip:"
print myextended-object/zip_code
```

```
Dimitri  tel:3333-3333
Igor  tel:9996-9669  gender:male  zip:666
```

find and select - for objects

`find` simply checks if the field exists, returning `true` or `none` .

`select` does the same checking, but if the field exists, returns its value.

```
Red []
```

```
obj: object [a: 44]
print find obj 'a
print select obj 'a
print find obj 'x
print select obj 'something
```

```
true
44
none
none
```

Notice that both look for the **word** (indicated by the ' symbol preceding it), not the variable itself. The variable would be accessed by a simple path like `obj/a`.

Note on extending objects:

Documentation says the built-in function `extend` should be able to add new items not only to `map!`, but also to `object!` However, this seems not to have been implemented yet.

< Previous topic Next topic >

Reactive programming

Reactive programming in Red's documentation

Reactive programming creates an internal mechanism that automatically updates things when a special kind of object is changed. No need to call functions or subroutines do do that. You change object A, and B is automatically changed too.

Reactor: is the object that, when changed, triggers the changes. Created by `make reactor!` .

Reactive expression: changes when the reactor changes. Created by `is` .

action! make reactor! and op! is Red's documentation on reactor! Red's documentation on is

Very basic example of using reactive programming:

```
Red[]

a: make reactor! [x: ""]       ;reactor object - triggers changes when
changed
b: is [a/x]                    ;reactive expression - changes when 'a'
changes

forever [
      a/x: ask "?"             ;here we input a value for 'x' field of
'a'
      print b                  ;here we print 'b' and... surprise! it
changed!
]
```

```
?house
house
?fly
fly
?bee
bee
```

A reactor can update itself:

```
Red[]

a:  make reactor! [x: 1 y: 2 total: is [x + y]]

forever [
      a/x: to integer! ask "?"
      print a/total
```

```
]
```

```
?33
35
?45
47
```

Be careful not create an endless loop. That happens if a change triggers a change in itself.

deep-reactor! Red's documentation

Just like `copy` has the refinement `/deep` to reach nested values (blocks within the main block), so does `reactor!`.

This program is supposed to repeat what you type on the console, but **it does not work**:

```
Red[]

a: make reactor! [z: [x: ""]]
b: object [w: is [a/z/x]]
b/w: "no change"

forever [
      a/z/x: ask "?"
      print b/w
]
```

```
?house
no change
?blue
no change
```

However, if you change to `deep-reactor!`:

```
Red[]

a: make deep-reactor! [z: [x: ""]]
b: object [w: is [a/z/x]]
b/w: "no change"

forever [
      a/z/x: ask "?"
      print b/w
]
```

```
?house
house
?blue
blue
```

function! react

Red's documentation

This is the built-in function used for creating reactive GUIs. Please look at GUI/Advanced topics.

Copied-and-pasted from the documentation:

function! clear-reactions

Removes all defined reactions, unconditionally.

function! react?

Checks if an object’s field is a reactive source . If it is, the first reaction found where that object’s field is present as a source, will be returned, otherwise `none` is returned. `/target` refinement checks if the field is a target instead of a source, and will return the first reaction found targeting that field or `none` if none matches.

/target => Check if it's a target instead of a source.

function! dump-reactions

Outputs a list of registered reactions for debug purposes.

< Previous topic Next topic >

OS interface

native! call Red Wiki Red-by-example MyCode4fun

Executes a shell command. In most cases, is the same as writing to the command prompt (CLI), but there are a few quirks.

The following code opens Windows Explorer:

```
>> call "explorer.exe"
== 11272              ; this is the number of the process opened.
```

This also works:

```
>> str: "explorer.exe"
== "explorer.exe"

>> call str
== 11916
```

However, the following code creates the process, but does not open Notepad on screen:

```
>> call "notepad.exe"
== 4180
```

If you want a behavior more similar to typing a command on the shell, you must use the refinement `/shell`:

```
>> call/shell "notepad.exe"       ;opens notepad on screen
== 6524
```

Generate a beep (tone, duration). Must have Powershell installed.

```
>> call "powershell [console]::beep(1000,500)"
== 1088
```

Other refinements:

/wait

Runs command and waits until the command you executed is finished to continue. Be careful: If you use /wait on a command that you can't finish (like `call "notepad.exe"` above), Red will wait... and wait.. indefinetly.

/input - we provide a string! a file! or a binary!, which will be redirected to stdin.

I don't understand this one. Seems as the same as simply `call` , as we provide string or a file anyway.

/output

We provide a a string! a file! or a binary! which will receive the redirected stdout from the command. Note that the output is appended.

The following code will create a text file with the shell output for "dir" (a list of files and folders from current path):

```
>> call/output "dir" %mycall.txt
== 0
```

This will create a (long) string with the results from "dir":

```
>> a: ""
== ""

>> call/output "dir" a
== 0

>> a
== { Volume in drive C has no label.^/ Volume Serial Number is BC5
;...
```

/show

Force the display of system's shell window (Windows only). Your script will run with windows command prompt open.

```
>> call/shell/show "notepad.exe"
== 12372
```

I believe this will have some use in the future, when Red allows using the `/console` option from the GUI console. Maybe.

/console

Runs command with I/O redirected to console (CLI console only at present, does not work with Red's normal GUI console).

Open Red on system console, as explained here, then, using the `/console` refinement on `call`, you the `cmd` output on the same console as Red:

```
C:\Users\André\Documents\RedIDE>red-063.exe --cli
--== Red 0.6.3 ==--
Type HELP for starting information.

>> call/console "echo hello world"
hello world
== 0
>>
```

native! write-clipboard & read-clipboard

Writes to and reads from the OS clipboard:

```
>> write-clipboard "You could paste this somewhere you find useful"
== true

>> print read-clipboard
You could paste this somewhere you find useful
```

< Previous topic Next topic >

I/O

As of october 2018, Red only has as simple I/O. That includes access to files and HTTP (HTTPS?).

< Previous topic Next topic >

I/O - HTTP

I have created a few files on helpin.red server to make tests with HTTP I/O:

http://helpin.red/samples/samplescript1.txt - a simple loop without Red's header (`repeat i 3 [prin "hello " print i]`).

http://helpin.red/samples/samplescript2.txt - a simple loop with Red's header. (`Red[] repeat i 3 [prin "hello " print i]`)

http://helpin.red/samples/samplehtml1.html - a sample html page

```
>> print read http://helpin.red/samples/samplescript1.txt
repeat i 3 [prin "hello " print i]

>> print read http://helpin.red/samples/samplescript2.txt
Red[] repeat i 3 [prin "hello " print i]
```

From a red script or using the console, you may execute code from a remote server:

```
>>  do read http://helpin.red/samples/samplescript1.txt    ;without header
hello 1
hello 2
hello 3
```

If the code in the remote server has the Red header, you may execute it directly, without the `read` statement:

```
>> do http://helpin.red/samples/samplescript2.txt    ;with Red [] header
hello 1
hello 2
hello 3
```

You may load data or code, including functions and objects:

```
>> a: load http://helpin.red/samples/samplescript1.txt
== [repeat i 3 [prin "hello " print i]]
>> do a
hello 1
hello 2
hello 3
```

HTML files may also be accessed for processing. Take a look at the example using the parse dialect.

```
>> print read http://helpin.red/samples/samplehtml1.html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>testHtmlPage</title>
</head>
<body>
...
</html>
```

Rebolek's red-tools has some HTTP tools that you may find interesting.

To be continued...

< Previous topic Next topic >

GUI - Overview

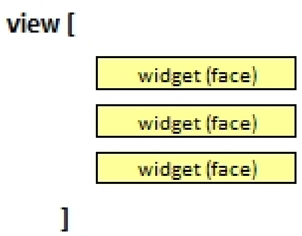
Very good information also in red-by-example. and in the Red documentation.

The following chapters will describe each of Red's *View Graphic Engine* & *VID dialect* elements (**faces, facets, container settings, layout commands** and **view refinements**) in detail, but I find that an overview of how Red creates GUIs makes it a lot simpler to understand how these elements relate to each other.

Notice that you may create GUIs using Red's positioning commands, like `at`, for each of its graphical elements (faces), but it also has a very clever GUI-creating method based on simple sequences and a few specific commands. This method is considered the *default* in this chapters.

Simple start:

Red creates GUIs by describing them in a view block. This description is very straightforward and in its simplest form would be:

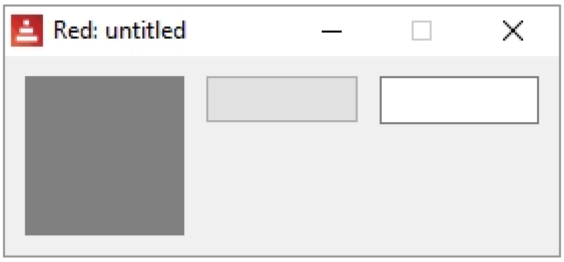


If you are going to compile your script, you must add "needs: view" in the Red header. If you run your scripts from the GUI console, the View module is already present.

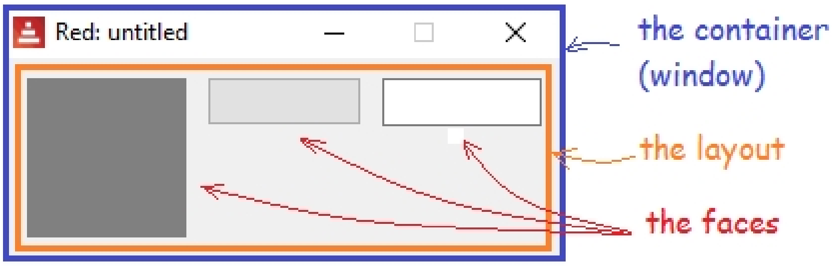
An example code of that:

```
Red [needs: view]  ; "needs: view" is needed if the script is going to be compiled

view[
      base
      button
      field
]
```

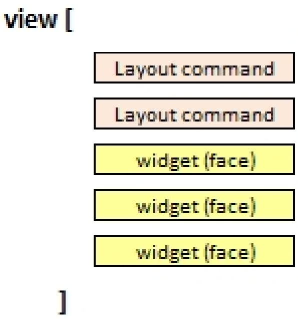
And the resulting GUI:



Red documentation calls things like buttons and fields "**faces**" (called "widgets" in some other languages). These **faces** are set on a **layout** inside a **container** (window)



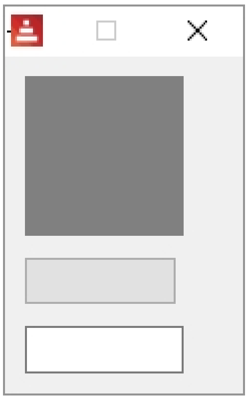
There are built-in functions (**layout commands**) that define how **faces** are displayed on this **layout**. These commands should be written before the faces they alter:



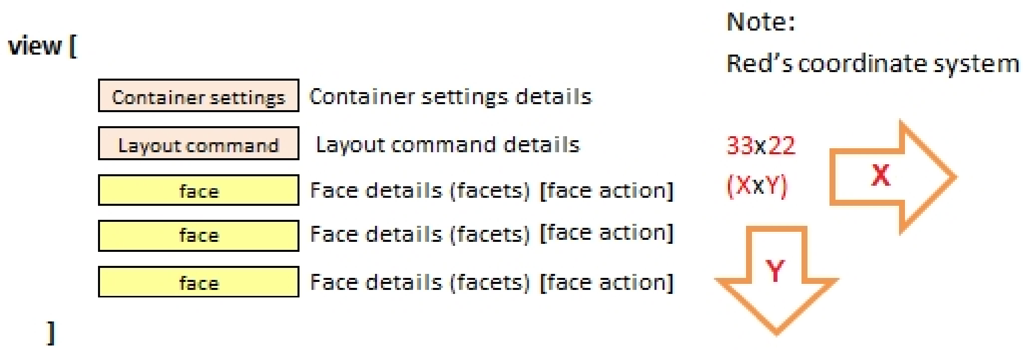
In the following example, `below` (a **layout command**) tells Red to arrange the **faces** below each other, instead of the default `across` of the first example:

```
Red [needs: view] ; "needs: view" is needed if the script is going to be
compiled

view[
        below ; layout command
        base  ; face (widget)
        button      ; face (widget)
        field ; face (widget)
]
```

The resulting GUI:



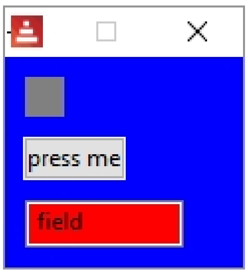
There is also the **container settings**, which describe how the window itself should look like. And both the **container settings** and the **layout commands** may allow further detailing, like its size, color etc. **Faces**not only allow this detailing (called **facets** in Red's jargon) but also may allow a block of commands to be performed by the **face** (called "**action facet**") in an event, e.g. the click of a button.



Exemple code:

```
Red [needs: view]

view[
	backdrop blue		;container setting
	below			; layout command
	base 20x20		; face and facet
	button 50x20 "press me" [quit]		; face, facets  and action facet
	field red "field"		; face and facets
]
```

And the resulting GUI:

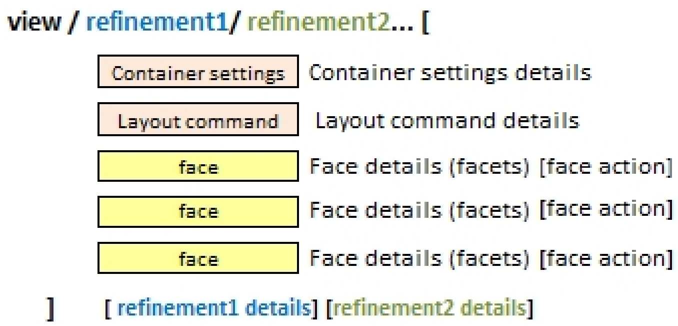


Red understands what to do with each **facet** simply by its `datatype!`. So if it sees a `pair!` it knows it's the size of the face, if it sees a `string!` it knows it's the text to be displayed. An odd consequence of that is that...

```
button 50x20 "press me" [quit]
button "press me" [quit] 50x20
button [quit] 50x20 "press me"
```

... are all the same, i.e. they result in the same GUI.

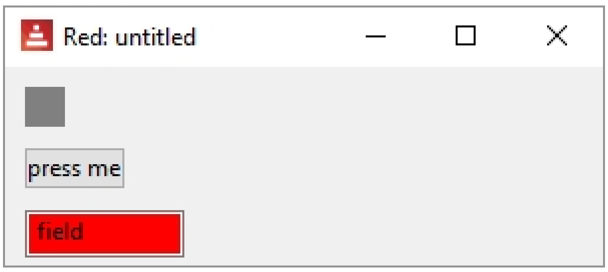
The `view` built-in function (command) allows refinements that will change the window itself (not the layout inside it). The refinements are described in blocks coded after the main view block, and should be coded in the same order that they were declared in the `view` command:



In the following code, **flags** tells Red that the window is of the **modal** type and it's resizable, while the option's refinement block makes the window show on the top left of the screen (50 pixels down, 50 pixels left):

```
Red [needs: view]

view/flags/options[
      size 300x100       ;container setting
      below              ; layout command
      base 20x20         ; face and facet
      button 50x20 "press me" [quit]      ; face, facets  and actor
      field red "field"        ; face and facets
]['modal 'resize] [offset: 50x50] ; flags and options
```

The resulting GUI:



< Previous topic Next topic >

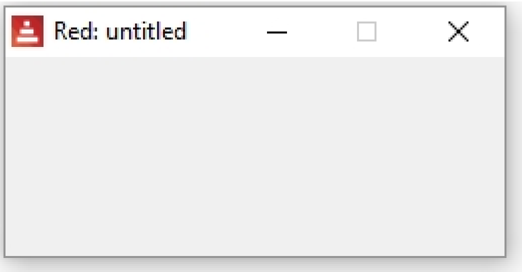
GUI - Container settings

These define the characteristics of the window that will contain your GUI elements.

VID DLS size

Sets the size of the window in pixels.

```
Red [needs: 'view]

view [
    size 250x100]
```



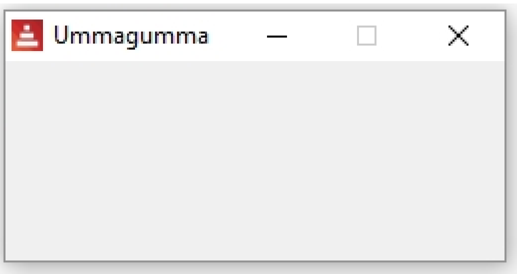
If you don't set a size, Red does it automatically.

As an interesting note, unless the window is big enough to show part of the title, you can't move (drag) it.

VID DLS title

Sets the title at top of the window.

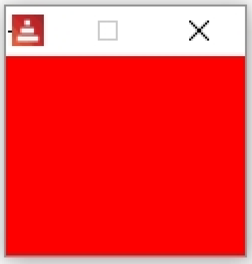
```
Red [needs: 'view]

view [
    title "Ummagumma"
    size 250x100]
```



VID DLS backdrop

Sets the background color of the window

```
Red [needs: 'view]

view [
    backdrop red]
```



actors

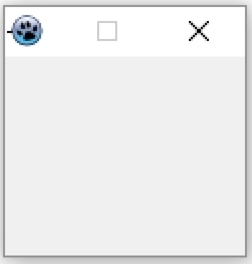
- See the specific chapter.

Setting an icon

This only works if you compile the code! Does NOT work on interpreted code.

Not a container setting, but I think it fits here. If you want to set an icon to your window that is not the Red default, add `icon: <path-to-icon>` after the `needs: 'view` in the Red initial block:

```
Red [needs: 'view
     icon: %project1.ico]

view []
```



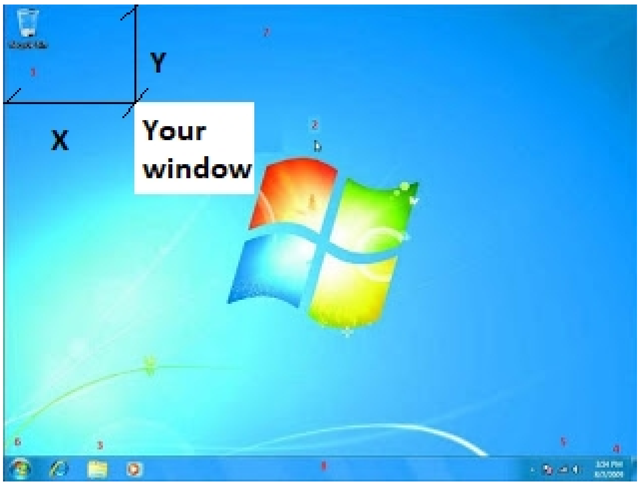
Refinements

Containers (windows) allow the refinements **options, flags, no-wait** and **tight.** The refinements options and flags are defined in blocks after the `view` main block.

/options

In the **options** refinement you can determine your window's offset and size (size seems to be definable in both ways, as a container setting or an option).

- **Offset** determines where your window will show, measured from the top left of your screen.



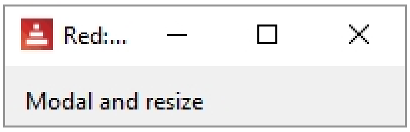
/flags

- **modal** - modal window. Demands attention, disables all other windows until you close it.

Note: if you create a window that is modal **and** no-title/no-border, it is pretty hard to get rid of it, I had to use Task Manager.

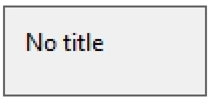
- **resize** - the window can be resized.

```
Red [needs: 'view]
View/flags [ size 200x30 text "Modal and resize" ] [modal resize]
```



- **no-title** - results in a rectangular frame with no title or buttons.

```
Red [needs: 'view]
View/flags [ text "No title" ] [no-title]
```

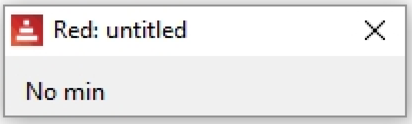


- **no-border** - results in a rectangular frame with no title or buttons and no border.

```
Red [needs: 'view]
View/flags [ text "No border" ] [no-border]
```

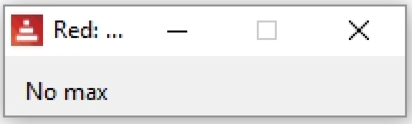
- **no-min** - only the close button is shown on window's top.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No min" ] [no-min]
```



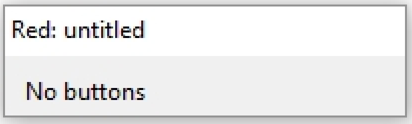
- **no-max** - the maximize button is shown as inactive.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No max" ] [no-max]
```



- **no-buttons** - no window's buttons (maximize, minimize, close) are shown.

```
Red [needs: 'view]
View/flags [ size 200x30 text "No buttons" ] [no-buttons]
```



- **popup** - Windows only - makes the window a popup. It has a special styling (close button only) and allows other windows to stay active. Closes if you change focus to other windows.

/no-wait

From the documentation: "View: Render on screen a window from a face tree or a block of VID code. Enters an event loop **unless /no-wait refinement is used**.

That is, if you don't use `no-wait`, View will create a face and stay there waiting for events. If you use no-wait, Red will execute the View block (show the GUI) and keep going forward in the script.

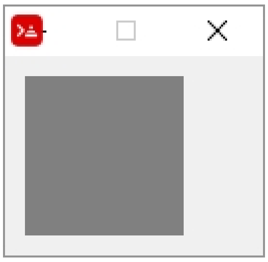
```
Red [needs: view]

view/no-wait [button "Quit" [quit]]

print {This text would not have been printed
if you have removed the "no-wait" refinement.
That is because the interpreter would stay in
the View block waiting for events}
```

/tight

Zero offset and origin.

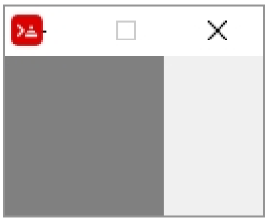
Default (without `/tight`):

```
Red [needs: view]
view[base]
```



With `/tight`:

```
Red [needs: view]
view/tight[base]
```

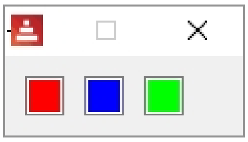


< Previous topic Next topic >

GUI - Layout commands

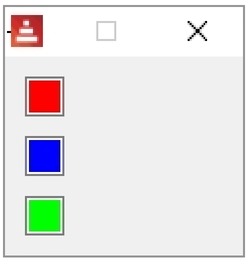
VID DLS across

```
Red [needs: view] ; "needs: view" is needed if the script is going to be compiled

view [
      across
      area 20x20 red
      area 20x20 blue
      area 20x20 green
]
```



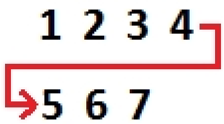
VID DLS below

```
Red [needs: view]

view [
      below
      area 20x20 red
      area 20x20 blue
      area 20x20 green
]
```

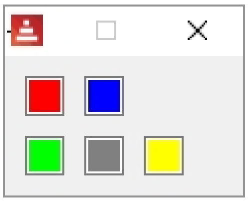


VID DLS return

`return` while in *across* mode:

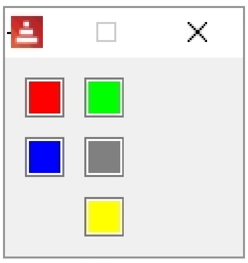



```
Red [needs: view]; "needs: view" is needed if the script is going to be
compiled

view [
      across
      area 20x20 red
      area 20x20 blue
      return
      area 20x20 green
      area 20x20 gray
      area 20x20 yellow
]
```



`return` **while in *below* mode:**

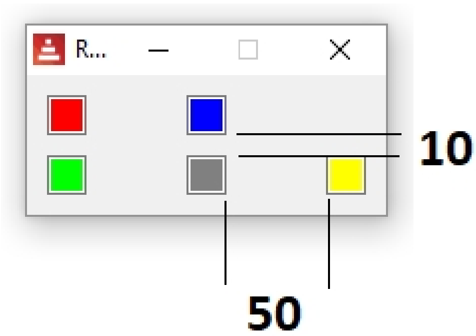

```
Red [needs: view]

view [
      below
      area 20x20 red
      area 20x20 blue
      return
      area 20x20 green
      area 20x20 gray
      area 20x20 yellow
]
```



VID DLS space

Sets the new spacing offset which will be used for placement of following faces.

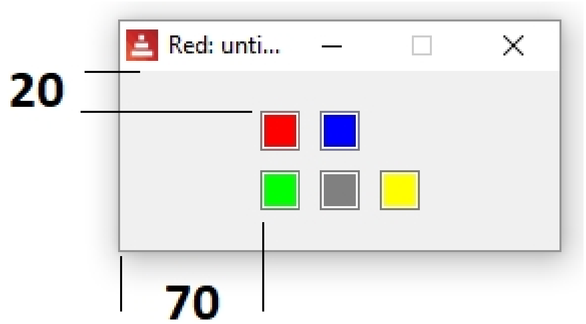
```
Red [needs: view]

view [
	across
	space 50x10
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
	area 20x20 yellow
]
```



VID DLS origin

Sets the offset of the first face from the upper left corner of the window's panel.

```
Red [needs: view]

view [
	across
	origin 70x20
	area 20x20 red
	area 20x20 blue
	return
	area 20x20 green
	area 20x20 gray
	area 20x20 yellow
]
```



VID DLS at

Places the next face at an absolute position. This positioning mode only affects the next following face, and does not change the layout flow position. So, the following faces, after the next one, will be placed again in the continuity of the previous ones in the layout flow.

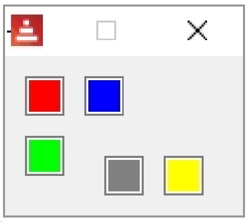
```
Red [needs: view]

view [
      across
      area 20x20 red
      area 20x20 blue
      return
      area 20x20 green
      at 2x5
      area 20x20 gray
      area 20x20 yellow
]
```

VID DLS pad

Modifies the layout current position by a relative offset. All the following faces on the same row (or column) are affected.

```
Red [needs: view]

view [
      across
      area 20x20 red
      area 20x20 blue
      return
      area 20x20 green
      pad 10x10
```

```
        area 20x20 gray
        area 20x20 yellow
]
```



native! do

This is the same `do` from the [Running code] chapter. In this case, it is used to run regular code inside your view.

You **can** do this:

```
Red [needs: 'view]
a: 33 + 12
print a          ;prints on console
view [
    text "hello"
]
```

But this will give you an **error**:

```
Red [needs: 'view]
view [
    text "hello"
    a: 33 + 12          ;ERROR!!!
    print a
]
```

Inside the view, you must code:

```
Red [needs: 'view]
view [
      text "hello"
      do [a: 33 + 12 print a]       ;OK!
]
```

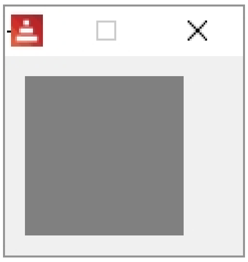
< Previous topic   Next topic >

GUI - Faces

VID DLS base

Most basic face. It may be used to create other faces. By default, it will only display a gray background.

```
Red [needs: view]

view [
      base
]
```



box and image

Strictly speaking, these are not faces, but styles of the `base` face. box is a `base` with a default transparent color and image is a `base` that expects and `image!` option, if none is provided, an empty image with white background is provided.

Note: the default sizes for a `base` and `box` is 80x80, but for an `image`, is 100x100.

```
Red [needs: view]

view [
      base
      box
      image
      image %smallballoon.jpeg
]
```

facets:

When Red interprets the code and finds a **face**, it looks for one or more of the following datatypes after it. Each has a meaning that will change the appearance of the face displayed. Their use will be made more clear in the examples of faces given ahead.

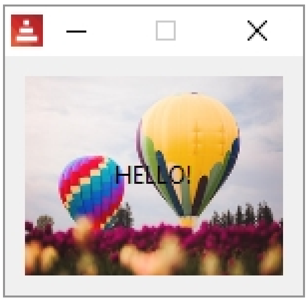
From Red's documentation:

Datatype	Purpose
integer!	Specifies the width of the face.
pair!	Specifies the width and height of the face.
tuple!	Specifies the color of the face's background.
issue!	Specifies the color of the face's background using hex notation (#rgb, #rrggbb, #rrggbbaa).
string!	Specifies the text to be displayed by the face.
percent!	Sets the `data` facet (useful for `progress` and `slider` types).
logic!	Sets the `data` facet (useful for `check` and `radio` types).
image!	Sets the image to be displayed as face's background.
url!	Loads the resource pointed to by the URL.
block!	Sets the action for the default event of the face.
get-word!	Uses an existing function as actor.

A list of facets copied from the documentation is given at the end of this chapter.

So, using **facets** with the `base` **face**:

```
Red [needs: view]

view [
      base "HELLO!" 130x100 %balloon.jpeg       ;balloon.jpeg is an
image saved on the same...
]                                               ;...directory as you Red
executable.
```



text face and text facet

There is a face named text and the text facet.

About the facet: text facets can be set in most faces and it can be formatted both in style and in position on the face. The following code...

```
Red [needs: view]

view [
      button "hello"
      button "bold" bold
      button "underline" underline
      button "strike" strike
      return
      button "top" 70x70 top
      button "middle" 70x70 middle ;vertical
      button "bottom" 70x70 bottom
      return
      button "left" 70x70 left
      button "center" 70x70 center ;horizontal
      button "right" 70x70 right
      return
      button "mix1" 70x70 top left
      button "mix2" 70x70 top center
      button "mix3" 70x70 top right
      return
      button "No" 70x70 right bold ; does not work!
]
```

... generates:

VID DLS text

The event that triggers the default actor is a *click* (see action facets)

```
Red [needs: view]

view [
      text "Hello"
]
```

Although **h1, h2, h3, h4** and **h5** may not be proper **faces** (they are styles), I think I should describe them here as they are text faces with different font sizes and are quite handy if you are working with text:

```
Red [needs view]

view [
      below
      h1 "Hello"
      h2 "Hello"
      h3 "Hello"
```

```
        h4 "Hello"
        h5 "Hello"
]
```

the font object

Maybe you already tried to set a color to your text and noticed that just adding, say, `blue` after the `text` face makes the background blue, but not the text. To format the font used to display strings on faces, there is this thing the documentation calls "font object". Think of it just as a set of commands to format the font. You write them after you declared your face, along with other facets.

font-name <Valid font name installed on the OS>

font-size <Font size in points>

font-color <Font color in R.G.B or R.G.B.A format, or its name>

You can also add `bold` `italic` `underline` or `strike`.

So:

```
Red [needs: view]

view [
        text "hello" font-name "algerian" font-size 18 font-color red bold
        text "hello" font-name "algerian" font-size 18 font-color blue
        text "hello" font-name "broadway" font-size 15 font-color green
strike
        text "hello" font-name "arial" font-size 12 font-color cyan
underline
]
```

VID DLS button

The event that triggers the default actor is a *click*.

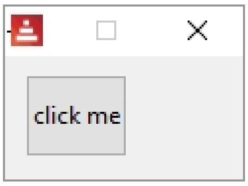
```
Red [needs: view]

view [
      button
]
```

action facets

Most faces allow an **action facet**, that is a block of commands that is triggered by an event. That event may be a mouse click (called "down" in Red), or something else, like pressing pressing *enter* or making a selection.

For buttons the **action facet** trigger is "down" event (mouse click) and in the following example it triggers the `quit` command that exits the program.`[quit]` would be the action facet (Should I call it the **default actor**?, you can set you own actors as described here).

```
Red [needs: view]

view [
      button 50x40 "click me" [quit]
]
```



colors

If you run the program below...

```
Red [needs: view]
view [
      base 30x30 aqua  text "aqua"       base 30x30 beige text "beige"

      base 30x30 black text "black"     base 30x30 blue   text "blue"

      return
      base 30x30 brick text "brick"      base 30x30 brown text "brown"

      base 30x30 coal text "coal"       base 30x30 coffee   text
```

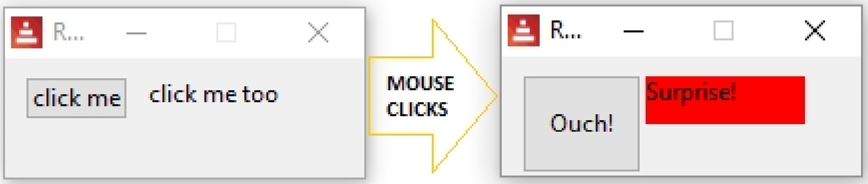
```
"coffee"
        return
        base 30x30 crimson text "crimson"  base 30x30 cyan text "cyan"

        base 30x30 forest text "forest"   base 30x30 gold text "gold"

        return
        base 30x30 gray text "gray"        base 30x30 green text "green"

        base 30x30 ivory text "ivory"     base 30x30 khaki text "khaki"

        return
        base 30x30 leaf text "leaf"        base 30x30 linen text "linen"

        base 30x30 magenta text "magenta" base 30x30 maroon  text "maroon"

        return
        base 30x30 mint text "mint"        base 30x30 navy text  "navy"

        base 30x30 oldrab text "oldrab"   base 30x30 olive text "olive"

        return
        base 30x30 orange text "orange"    base 30x30 papaya text "papaya"

        base 30x30 pewter text "pewter"   base 30x30 pink text "pink"

        return
        base 30x30 purple text "purple"    base 30x30 reblue text "reblue"

        base 30x30 rebolor text "rebolor"  base 30x30 red text "red"
]
```

...you get:

faces are objects

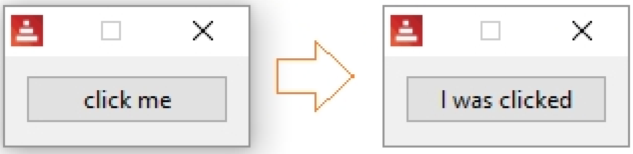
Each face is a clone of the `face!` template object and you can change their attributes (the facets) during runtime:

```
Red [needs: 'view]

view [
    size 180x60
     b: button 50x20 "click me" [b/text: "Ouch!"  b/size: 60x50]
     t: text "click me too"  [t/color: red t/text: "Surprise!"]
    ]
```



Inside the **action facet**, you can refer to a face's attribute using `face/<attribute>`,so:

```
Red [needs: 'view]
view [
    button 100 "click me" [face/text: "I was clicked"]
    ]
```



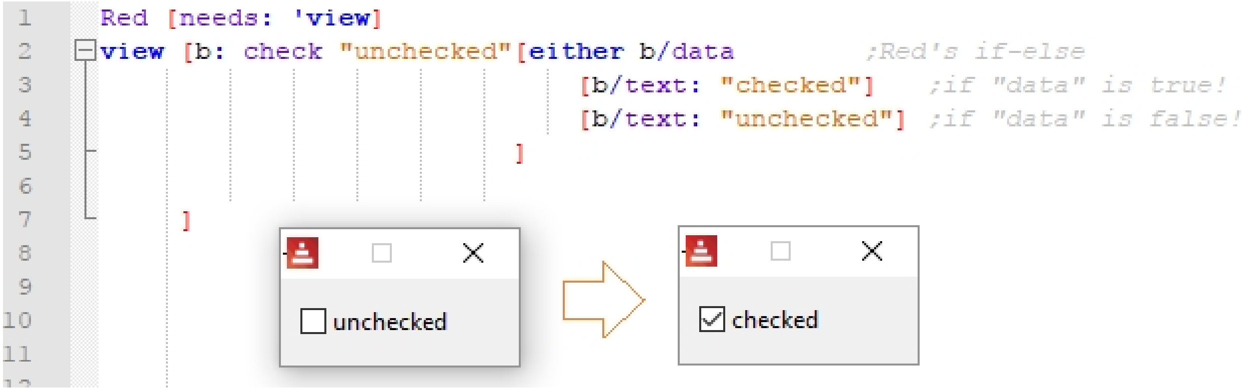
Run the script below and click the button to have an idea of the complexity of a face like a button:

```
Red [needs: view] view [b: button [print b]]
```

VID DLS check

```
Red [needs: view]
view [
        check
]
```

The event that triggers the action facet is a *change.* The current state is in the attribute */data (*`true` *or* `false`*)*



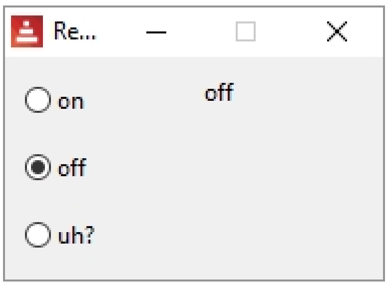
By the way, that is not proper coding style, just seems more didactic. Take a look at Red's [Coding Style Guide](#).

VID DLS radio

The event that triggers the action facet is a *change.* The current state is in the attribute */data*

This type represents a radio button, with an optional label text, displayed on left or right side. Only one radio button per pane is allowed to be checked.

```
Red [needs: view]
view [
      r1: radio "on" [t/text: "on"]
      t: text "none"
      return
      below
      r2: radio "off" [t/text: "off"]
      r3: radio "uh?" [t/text: "uh?"]
]
```



VID DLS field

To input text data.

The events that triggers the action facet is *enter.* The current state (the text inside the field) is in the attribute */data. _This works both ways: if you change* /data, *the text displayed in* *the field is changed. Trying to change* /data_ with code inside the view block but outside the action facet gives you an error.

```
Red [needs: view]
view [
      field
]
```

This example prints your input **on the console** when you press enter:

```
Red [needs: view]
view [
      f: field [print f/text]
]
```

`field` allows a `no-border` facet*:

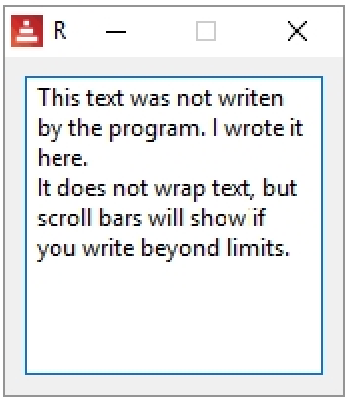
```
Red [needs: view]
view [
      f: field no-border
]
```

*Just so you know, in Red's documentation they call `no-border` a "flag", not a facet.

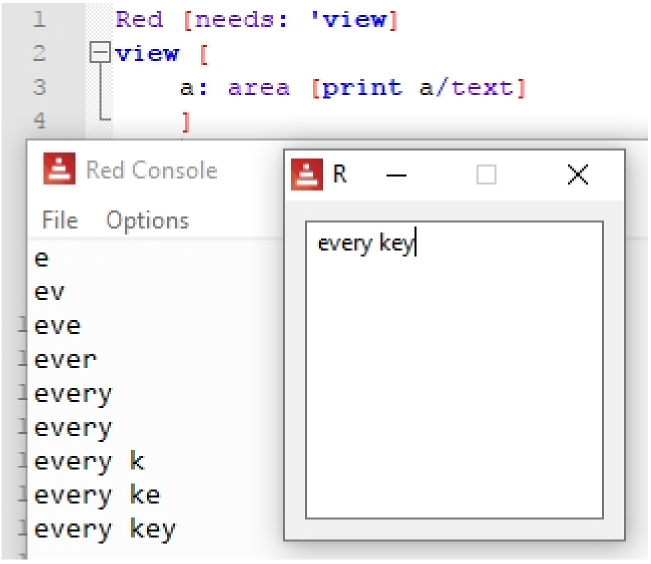
VID DLS **area**

The event that triggers the action facet is a *change.* The text inside `area` is in the attribute */text.* You may change the text assigning strings to */text*.

```
Red [needs: view]
view [
      area
]
```



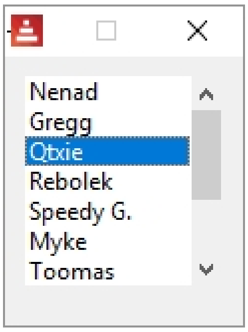
Since any change is a triggering event, every keystroke inside the `area` executes the action facet:



VID DLS text-list

The event that triggers the action facet is a *selection.* The strings to be listed are in the attribute `/data`. The index of the selected data is in the attribute `/selected`

```
Red [needs: view]
view [
      tl: text-list 100x100 data[
      "Nenad" "Gregg" "Qtxie" "Rebolek"
      "Speedy G." "Myke" "Toomas"
      "Alan" "Nick" "Peter" "Carl"
      ]
      [print tl/selected]
]
```




```
3
```

To use the string selected, the code snippet could be:

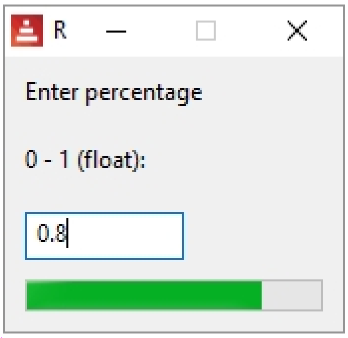
```
pick face/data face/selected
```

This would be the same as : `pick ["Nenad" "Greg" "Qtxie" "Rebolek" (...)] 3`

VID DLS progress

I don't think it allows an action facet, it's just a display. The current state is set in the attribute `/data`, as a `percent!` or a `float!` between 0 and 1.

```
Red [needs: view]
view [
      below
      text "Enter percentage"
      text "0 - 1 (float):"
      field [p/data: to percent! face/data]
      p: progress
]
```



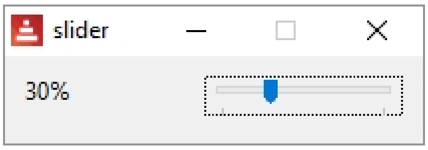
VID DLS slider

The event that triggers the action facet is a *change.* The current percentage is in the attribute `/data` *, as a* `percent!` *datatype*.

```
Red [needs: view]
```

```
view [
	title "slider"
	t: text "Percentage"
	slider 100x20 data 10% [t/text: to string! face/data]
]
```

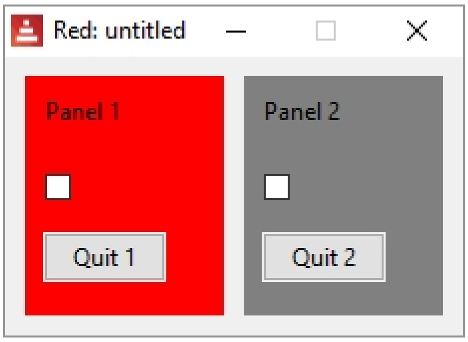
Move the slider's cursor to see the percentage data:



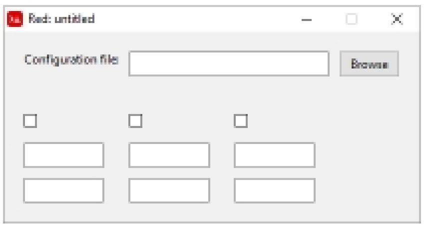
VID DLS panel

Creates a new area where you can display faces using the same syntax explained so far. I think the example below is self-explanatory. Does not seem to allow an action facet.

```
Red [needs: view]
view [
	panel red [size 100x120 below text red "Panel 1" check button
"Quit 1" [quit]]
	panel gray [size 100x120 below text gray "Panel 2" check button
"Quit 2" [quit]]
]
```



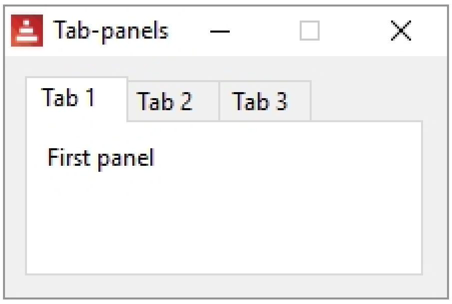
An important use for `panel` is to create nicely formated GUIs without using too many `at` commands. For example, to create the layout below, you could use two `panel`s, one for the upper part and another for the lower part:



VID DLS tab-panel

Creates a set of panels where only one can be seen at a given time, selected by a tab. Does not seem to allow an action facet. Data is at: `<tab-panel>/data` - Block of tabs names (string values).
`<tab-panel>/pane` - List of panels corresponding to tabs list (block!).
`<tab-panel>/selected` - Index of selected panel or `none` value (integer!) (read/write). i.e. the panel that has the focus, 1 for the first, 2 for the second and so on.

```
Red [needs: view]
view [
      Title "Tab-panels"
      tab-panel 200x100 [
            "Tab 1 " [text "First panel"]
            "Tab 2 " [text "Second panel"]
            "Tab 3 " [text "Third panel"]
      ]
]
```



And each panel allows a set of faces:

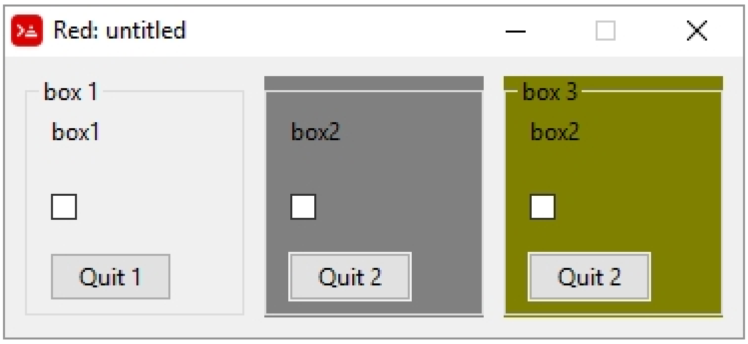
```
Red [needs: view]
view [
      Title "Tab-panels"
      tab-panel 110x140 [
            "Tab 1 " [
            below
            text font-color blue "First panel"
            button "quit" [quit]
            check "check to quit" [quit]
            ]
            "Tab 2 " [text "Second panel"]
      ]
]
```

VID DLS group-box

From documentation: A group-box is a container for other faces, with a visible frame around it. *This is a temporary style which will be removed once Red has the support for* `edge` *facet.*

Seems to me it it's just a panel with a border. I noticed it gives strange results when you give it a color:

```
Red [needs: view]
view [
      group-box "box 1" [size 110x120 below text "box1" check button
"Quit 1" [quit]]
      group-box gray [size 110x120 below text "box2" check button "Quit
2" [quit]]
      group-box "box 3" olive [size 110x120 below text "box2" check
button "Quit 2" [quit]]
]
```



VID DLS drop-down

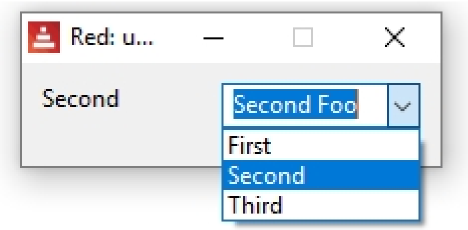
The event that triggers the action facet is *enter*.

From the documentation: "This type represents a vertical list of text strings, displayed in a foldable frame. A vertical scrollbar appears automatically if the content does not fit the frame. The `data` facet accepts arbitrary values, but only string values will be added to the list and displayed. Extra values of non-string datatype can be used to create associative arrays, using strings as keys. The `selected` facet is a 1-based integer index indicating the position of the selected string in the list, and not in the `data` facet."

You can type text in the text-box. The content of the text-box will be in the attribute `/text`. It will show when you press "enter"

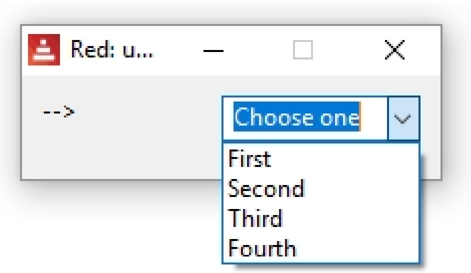
```
Red [needs: view]

view [
       t: text "-->"
       drop-down "Choose one" data [
               "First"
               "Second"
               "Third"
```

```
        ] [ t/text: pick face/data face/selected ]
]         ;must press enter to change text
```



Here is an example using events:

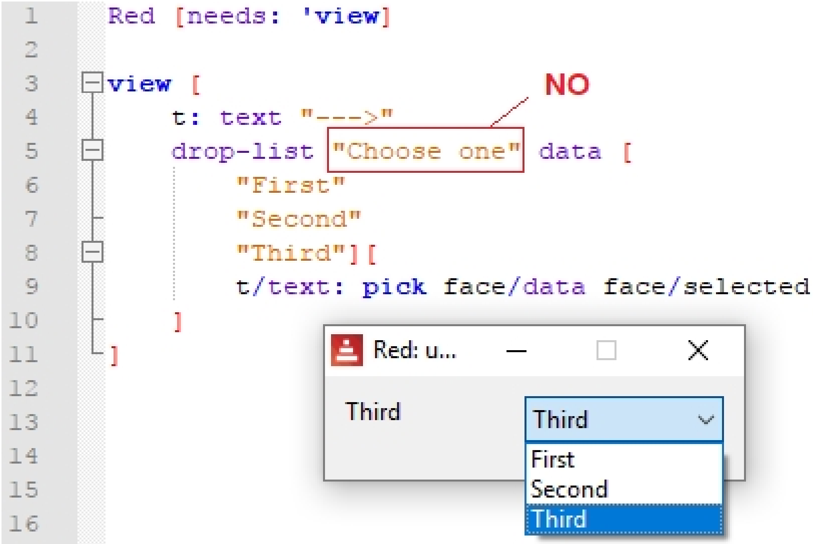
```
Red [needs: view]
view [
      t: text "-->"
      drop-down "Choose one" data ["First" "Second" "Third" "Fourth"]
      on-change [ t/text: pick face/data face/selected ]
]
```



VID DLS drop-list

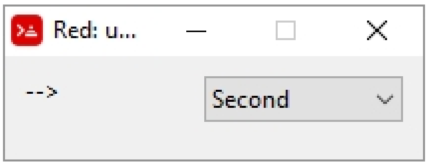
The event that triggers the action facet is *change.*

Similar to drop-down, but you cannot write in the text box and it does not show a default text.



You can, however, give it a default selection by specifying e.g. `select 2` :

```
Red [needs: view]
view [
      t: text "-->"
      drop-list "Choose one" select 2 data [
            "First"
            "Second"
            "Third"
      ] [ t/text: pick face/data face/selected ]
]
```



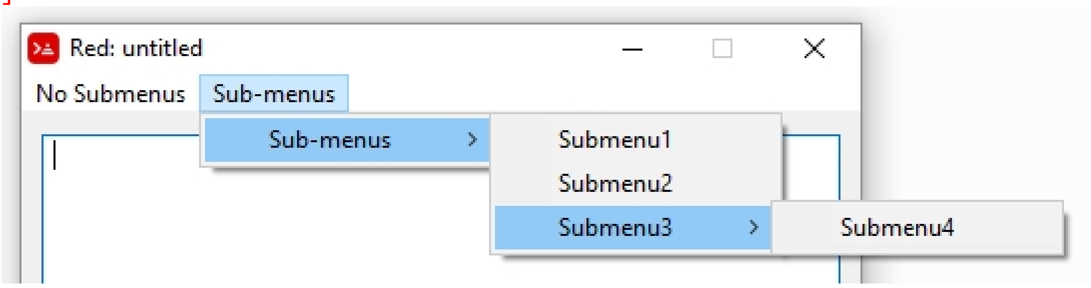
VID DLS menus

`menu` is a `facet`, but I believe that who is learning Red wants to know "what are the *widgets* available for Red", and `menu` looks and feels like a *widget* to me. Since throughout helpin.red I mention that Red's *widgets* are called "faces", I think it deserves an entry as one, even though technically it may be something else.

It's very poorly documented. Toomas Vooglaid kindly provided a few examples of the use of menus. The first is a rewriting of an example taken from Nick Antonaccio's [Short Red Code Examples](#) (I suggest you take a look at that excellent webpage), but using only VID:

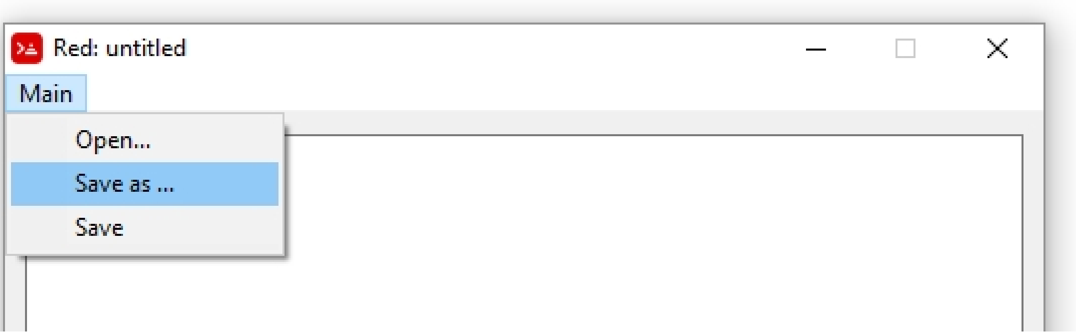
```
Red [needs: view]
view/options [area 400x400] [
    menu: [
        "No Submenus" [
            "Print" prnt
            ---
            "Quit" kwit
        ]
```

```
        "Sub-menus" [
            "Sub-menus" [
                "Submenu1" s1
                "Submenu2" s2
                "Submenu3" [
                    "Submenu4" s4
                ]
            ]
        ]
    ]
    actors: make object! [
        on-menu: func [face [object!] event [event!]][
            if event/picked = 'kwit [unview/all]
            if event/picked = 'prnt [print "print menu selected"]
            if event/picked = 's4 [print "submenu4 selected"]
        ]
    ]
]
```



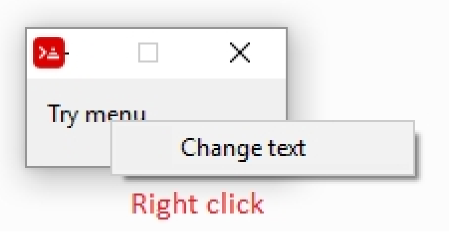
The second example is a simple framework of a text editor using menus:

```
Red [title: "Menus" needs: 'view]

view/options [editor: area 500x300][
   menu: ["Main" ["Open..." open "Save as ..." save-as "Save" save]]
   actors: object [on-menu: func [face event /local new-name][switch
event/picked [
      open [if new-name: request-file [editor/text: read editor/extra:
new-name set-focus editor]]
      save-as [if new-name: request-file/save [write editor/extra: new-
name editor/text]]
      save [write editor/extra editor/text]
]]]]
```



The third example makes a menu appear when you right-click on text:

```
Red [needs: view]
view [text "Try menu" with [
    menu: ["Change text" change]
    actors: object [on-menu: func [f e][
        switch e/picked [change [
            view/flags [text "Please enter new text:" field [
                f/text: face/text unview
            ]][modal]
]]]]]]
```



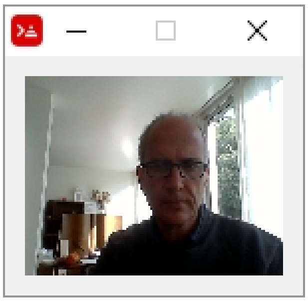
This last example can be rewritten using `on-menu` event:

```
Red [needs: view]
view [
    text "Try menu"
    with [menu: ["Change text" change]]
    on-menu [
        f: face
        if event/picked = 'change [
            view/flags [
                text "Please enter new text:"
                field [f/text: face/text unview]
            ][modal]
        ]
    ]
]
```

VID DLS camera

Displays a camera stream.

```
Red []
view [
    cam: camera 130x100 select 1
]
```



This script saves a snapshot of the camera stream as as .jpeg image:

```
Red []
count: 0
snapshot: does [
    load rejoin [mold '% 'picture count: count + 1 '.jpeg]
]
view [
    cam: camera 120x100 select 1
    button "Save picture" [save/as snapshot to-image cam 'jpeg]
]
```

Facets according to Red's documentation:

Facet	Datatype	Mandatory?	Applicability	Description
type	word!	yes	all	Type of graphic component
offset	pair!	yes	all	Offset position from parent top-left origin.
size	pair!	yes	all	Size of the face.
text	string!	no	all	Label text displayed in the face.
image	image!	no	some	Image displayed in the face background.
color	tuple!	no	some	Background color of the face in R.G.B or R.G.B.A format.
menu	block!	no	all	Menu bar or contextual menu.
data	any-type!	no	all	Content data of the face.
enabled?	logic!	yes	all	Enable or disable input events on the face.
visible?	logic!	yes	all	Display or hide the face.
selected	integer!	no	some	For lists types, index of currently selected element.
flags	block!, word!	no	some	List of special keywords altering the display or behavior of the face.
options	block!	no	some	Extra face properties in a [name: value] format.
parent	object!	no	all	Back-reference to parent

				face (if any).
pane	block!	no	some	List of child face(s) displayed inside the face.
state	block	no	all	Internal face state info*(used by Viewengine only)*.
rate	integer!, time!	no	all	Face's timer. An integer sets a frequency, a time sets a duration, none stops it.
edge	object!	no	all	*(reserved for future use)*
para	object!	no	all	Para object reference for text positioning.
font	object!	no	all	Font object reference for setting text facet's font properties.
actors	object!	no	all	User-provided events handlers.
extra	any-type!	no	all	Optional user data attached to the face (free usage).
draw	block!	no	all	List of Draw commands to be drawn on the face.

< Previous topic

Next topic >

GUI - Events and actors

Events:

Mouse clicking, mouse hovering, key pressing etc., are events that you may want to associate with code. We saw on last chapter that there is something called **action facet** that executes code triggered by a default event. You can add more blocks of code associated with events by following this layout:

```
view [
      face  facet facet [action facet]
            on-event [action]
            on-event [action]
      face2 facet facet [action facet]
            on-event [action]
            on-event [action]
   ]
```

There is an extensive list of possible **events** in the documentation. I copied it at the end of this chapter for reference.

Each face accepts a set of events, i.e. not all events apply to all faces.

I made a short set of examples. I see no point in giving examples of each existing event, since the logic is the same:

down - left mouse button pressed;

over - mouse cursor passing over a face;

```
Red [needs: view]
view [
      t: area 40x40 blue
      on-down [quit]
      on-over [either t/color = red [t/color: blue][t/color: red]]
]
```

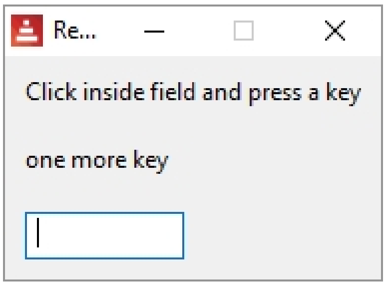
wheel - mouse wheel being turned;

```
Red [needs: view]

list: ["first" "second" "third" "fourth"]
view [
      t: text "Place your cursor over here and roll the wheel"
            on-wheel [
                  t/text: first list
                  list: next list
                  if tail? list [list: head list]
            ]
```

```
]
```

key-down - a key has been pressed;

```
Red [needs: view]

list: ["key" "another key" "one more key"]
view [
	below
	text "Click inside field and press a key"
	t: text 100
	a: field
		on-key-down [
			t/text: first list
			list: next list
			if tail? list [list: head list]
		]
]
```



time - the delay set by face's `rate` facet expired.

The following example "blinks" a text at a 1 second rate (see **rate** in chapter [GUI-Advanced topics](#)):

```
Red [needs: view]

view [
	t: text "Now you see..." rate 1
		on-time [either t/text = "" [t/text: "Now you see..."]
[t/text: ""]]
]
```

close - this is a window event: the window was closed. Very useful to include code to be executed when the user quits (closes the window).

```
Red [needs: view]

view [
	on-close [print "bye!"]
	button [print "click"]
]
```

Actors

Actors is the name of the **event handling functions** in Red. That is, the code inside the block after `on-<event>` . So why not call them just **event handlers** like most other language do? I think is because they are an object inside the face as you can see if you run this code below and click on the **area face**:

```
Red [Needs: view]
view [
      t: area 40x40 blue on-down [print t] ;click to quit
      on-over [either t/color = red [t/color: blue][t/color: red]]
]
```

You will see in the console, nearly at the end of the print, an object with the **actors** described:

```
(...)
edge: none
para: none
font: none
actors: make object! [
    on-down: func [face [object!] event [event! none!]][print t]
    on-over: func [face [object!] event [event! none!]][either t/color =
red [t/color: blue] [t/color: red]]
]
extra: none
draw: none
(...)
```

on-create actor:

In addition to the GUI events, it is possible to define an `on-create` actor which will be called when the face is shown for the first time, just before system resources are allocated for it. Unlike other actors, `on-create` has only one argument, `face`.

Full list of event names:

Name	Input type	Cause
down	mouse	Left mouse button pressed.
up	mouse	Left mouse button released.
mid-down	mouse	Middle mouse button pressed.
mid-up	mouse	Middle mouse button released.
alt-down	mouse	Right mouse button pressed.
alt-up	mouse	Right mouse button released.
aux-down	mouse	Auxiliary mouse button pressed.
aux-up	mouse	Auxiliary mouse button released.

drag-start	mouse	A face dragging starts.
drag	mouse	A face is being dragged.
drop	mouse	A dragged face has been dropped.
click	mouse	Left mouse click (button widgets only).
dbl-click	mouse	Left mouse double-click.
over	mouse	Mouse cursor passing over a face. This event is produced once when the mouse enters the face and once when it exits. If `flags` facet contains **all-over** flag, then all intermediary events are produced too.
move	mouse	A window has moved.
resize	mouse	A window has been resized.
moving	mouse	A window is being moved.
resizing	mouse	A window is being resized.
wheel	mouse	The mouse wheel is being moved.
zoom	touch	A zooming gesture (pinching) has been recognized.
pan	touch	A panning gesture (sweeping) has been recognized.
rotate	touch	A rotating gesture has been recognized.
two-tap	touch	A double tapping gesture has been recognized.
press-tap	touch	A press-and-tap gesture has been recognized.
key-down	keyboard	A key is pressed down.
key	keyboard	A character was input or a special key has been pressed (except control; shift and menu keys).
key-up	keyboard	A pressed key is released.
enter	keyboard	Enter key is pressed down.
focus	any	A face just got the focus.
unfocus	any	A face just lost the focus.
select	any	A selection is made in a face with multiple choices.
change	any	A change occurred in a face accepting user inputs (text input or selection in a list).
menu	any	A menu entry is picked.
close	any	A window is closing.

time	timer	The delay set by face's `rate` facet expired.

Notes:

- touch events are not available for Windows XP.+
- One or more `moving` events always precedes a `move` one.
- One or more `resizing` events always precedes a `resize` one.

< Previous topic Next topic >

GUI - Event!, mouse position and key pressed

Every time an `event!` happens on a face, you may get information about it from `event/`<see list below>.

Mouse position:

So, in the stripped-down example below, we print the event type and the mouse coordinates when the event happens, in this case, a mouse `down` (click) event:

```
Red [needs: view]

view [
    base 100x100
    on-down [
            print event/type
            print event/offset
      ]
]
```

```
down
39x57
down
86x43
```

Key pressed:

Interestingly, in the example above, you only get `none!` if you try to print event/key, but in the example below, using on-key as event, you get not only the key pressed, but also the mouse coordinates. In fact, you get mouse coordinates from wherever the mouse is on the screen when the key is pressed, referenced to the upper left corner of the face.

```
Red [needs: view]

view [
    area 100x100
    on-key [
            print event/type
            print event/offset
            print event/key
      ]
```

```
]
```

```
key
-59x84
r
key
-36x59
s
key
-116x79
o
```

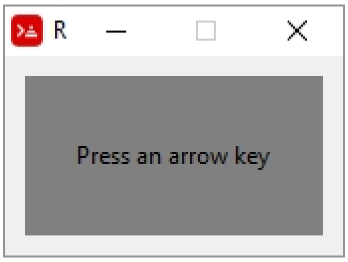
Note that, in the example above, if we change `area` for `base`, we get no results on the console. However, this code works:

```
Red [needs: view]
view [base focus on-key [probe event/key]]
```

Here, `focus` seems to make the difference. Note that `probe` outputs a char!

Another example:

```
Red [needs: view]
view [canvas: base 150x80 "Press an arrow key" focus
      draw[]
      on-key  [
       switch event/key
            [
            up [canvas/text: "move up"]
            down [canvas/text: "move down"]
            left [canvas/text:  "move left"]
            right [canvas/text:  "move right"]
            ]
      ]
]
```



Here is a list of events taken from [Red's official documentation](#):

Field	Returned value
`type`	Event type (`word!`).

`face`	Face object where the event occurred (`object!`).
`window`	Window face where the event occured (`object!`).
`offset`	Offset of mouse cursor relative to the face object when the event occurred (`pair!`). For gestures events, returns the center point coordinates.
`key`	Key pressed (`char!` `word!`).
`picked`	New item selected in a face (`integer!` `percent!`). For a mouse `down`event on a `text-list`, it returns the item index underneath the mouse or `none`. For `wheel` event, it returns the number of rotation steps. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. For `menu` event, it returns the corresponding menu ID (`word!`). For zooming gesture, it returns a percent value representing the relative increase/decrease. For other gestures, its value is system-dependent for now (Windows: `ullArguments`, field from GESTUREINFO).
`flags`	Returns a list of one or more flags (see list below) (`block!`).
`away?`	Returns `true` if the mouse cursor exits the face boundaries (`logic!`). Applies only if `over` event is active.
`down?`	Returns `true` if the mouse left button was pressed (`logic!`).
`mid-down?`	Returns `true` if the mouse middle button was pressed (`logic!`).
`alt-down?`	Returns `true` if the mouse right button was pressed (`logic!`).
`ctrl?`	Returns `true` if the CTRL key was pressed (`logic!`).
`shift?`	Returns `true` if the SHIFT key was pressed (`logic!`).

< Previous topic

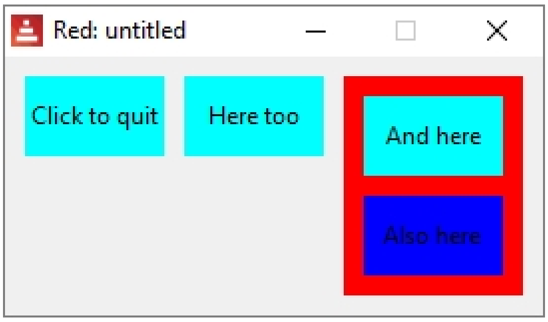
Next topic >

GUI - Advanced topics

VID DLS style

`style` is used to create your own custom faces.

```
Red [Needs: view]

view [
      style myface: base 70x40 cyan [quit]

      myface "Click to quit"
      myface "Here too"
      panel red 90x110 [
            below
            myface "And here"
            myface "Also here" blue
      ]
]
```



function! view and function! unview

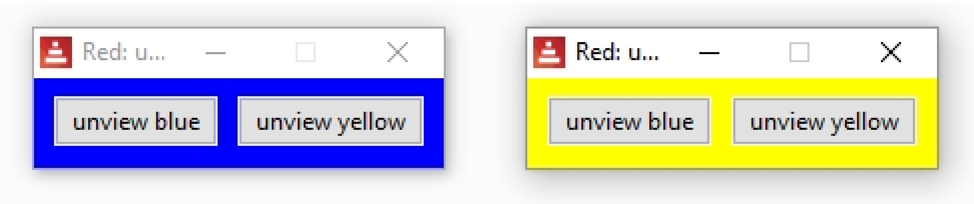
Multiple windows on the screen

`view` can also be used to show windows with faces (a face tree) that were created in another part of the code. `unview`, of course, closes the view. The following code creates two identical but independent (different face trees) windows in different parts of the screen:

```
Red [needs: view]
my-view: [button {click to "unview"} [unview]]

print "something"  ;do something else
print "biding my time" ;do something else

view/options/no-wait my-view [offset: 30x100]
view/options/no-wait my-view [offset: 400x100]
```

`unview` allows the refinement `/only` to act only on a given window:

```
Red [needs: view]

v1: view/options/no-wait [
    backdrop blue
    button "unview blue"[unview/only v1]
    button "unview yellow" [unview/only v2]
][   ;options:
    offset: 30x100
]
v2: view/options/no-wait [
    backdrop yellow
    button "unview blue"[unview/only v1]
    button "unview yellow" [unview/only v2]
][   ;options:
    offset: 400x100
]
```



Refinements for `view`:

/tight => Zero offset and origin.
/options =>
/flags =>
/no-wait => Return immediately - do not wait.

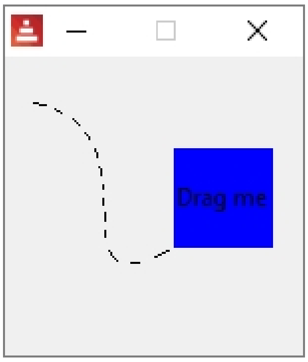
Refinements for `unview`:

/all => Close all views.
/only => Close a given view.

VID DLS loose

`loose` is a facet that allows the face to be dragged (moved around) by the mouse.

```
Red [needs: view]

view [
      size 150x150
      base blue 50x50 "Drag me" loose
]
```

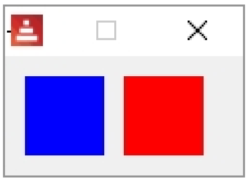


VID DLS all-over

The `on-over` event normally happens when the mouse cursor "enters" or "leaves" the face. When you set the `all-over` facet, every event that happens when the cursor is on the face, like movements or clicks, generates an `on-over` event.

In the following example the left square changes colors only when the mouse cursor "enters" or "leaves it" (over or not over), but the square on the right changes colors with every little movement of the cursor over it, or with mouse left button clicks:

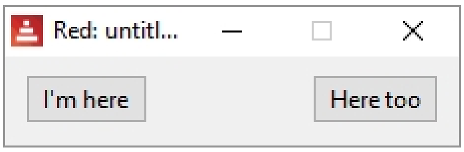
```
Red [needs: view]

view [
	a: base 40x40 blue
		on-over [either a/color = red [a/color: blue][a/color: red]]
	b: base 40x40 blue all-over
		on-over [either b/color = red [b/color: blue][b/color: red]]
]
```



VID DLS hidden

Makes the face invisible by default. One possible use is to create a hidden face with a `rate`, so you may have the timing without the need of showing a face.

```
Red [needs: view]

view [
	button "I'm here"
	button "I'm not" hidden
	button "Here too"
]
```



VID DLS disabled

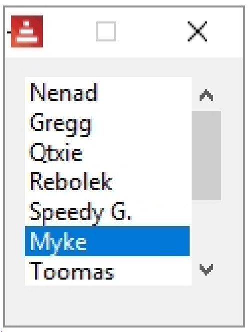
Disables the face by default (the face will not process any event until it is enabled).

```
Red [needs: view]

view [
      button "I should quit, but I don't" disabled [quit]
      button "Quit" [quit]
]
```

VID DLS select

Sets the `selected` facet of the current face. Used mostly for lists to indicate which item is pre-selected.

```
Red [needs: view]
view [
      tl: text-list 100x100 data  [
      "Nenad" "Gregg" "Qtxie" "Rebolek"
      "Speedy G." "Myke" "Toomas"
      "Alan" "Nick" "Peter" "Carl"
      ] select 6
      [print tl/selected]
]
```

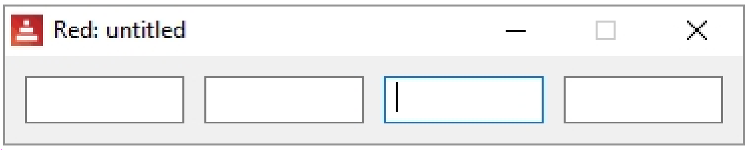


VID DLS focus

Gives the focus to the current face when the window is displayed for the first time. Only one face can have the focus. If several `focus` options are used on different faces, only the last one will get the focus.

```
Red [needs: view]
view [
```

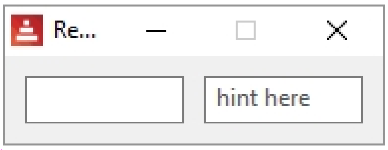
```
    field
    field
    field focus
    field
]
```



VID DLS hint

Provides a hint message inside `field` faces, when the field's content is empty. That text disappears when any new content is provided (user action or setting the `face/text` facet).

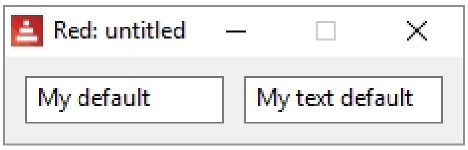
```
Red [needs: view]
view [
    field
    field hint "hint here"
]
```



VID DLS default

Defines a default value for `data` facet when the conversion of `text` facet returns `none`. Currently only works for `text` and `field` face types.

```
Red [needs: view]
view [
      a: field 100 default "My default"
      b: field 100 "My text default"
      do [
            print a/text
            print a/data      ; "data" was defined by "default" facet
            print b/text
            print b/data      ; this will give you an error, as "data"
was not defined yet
      ]
]
```



```
My default
My default
My text default
*** Script Error: My has no value
*** Where: print
*** Stack: view layout do-safe
```

VID DLS with

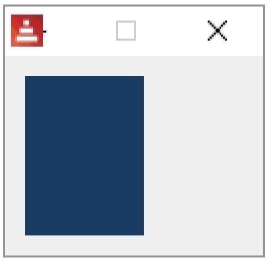
Suppose you want to create a face whose facets' values are evaluated as you create it. You can't use evaluation in your face "arguments", so you set them with `with` .

This does not work:

```
Red [needs: view]
a: 2
b: 3
view [
      base a * 30x40 b * 8.20.33
]
```

This works:

```
Red [needs: view]
a: 2
b: 3
view [
      base with [
            size: a * 30x40
            color: b * 8.20.33
      ]
]
```



VID DLS rate

`rate` is a facet that has a timer. When the timer "ticks" an `on-time` event is generated. Notice that the rate argument is an `integer!` it means "times per second" , so a `rate` of 20 is **faster** than a `rate` of 5. You may provide a `time!` argument to set a time for `rate`.

This code makes a text blink:

```
Red [needs: view]
```

```
view [
        t: text "" rate 2
        on-time [either t/text = "" [t/text: "Blink"] [t/text: ""]]
]
```

This code makes a crude animation where a blue base crosses the window:

```
Red [Needs: 'View]

view[
    size 150x150
    b: base 40x40 blue "I move" rate 20
        on-time [b/offset: b/offset + 1x1]
]
```

Slower rates:

For periods longer thant 1 second, use a `time!` argument for `rate`:

```
Red [Needs: view]

view[
    t: text "" rate 0:0:3
        on-time [either t/text = "" [t/text: "Blink" print now/time]
[t/text: "" print now/time]]
]
```

function! react

`react` is a facet that links the behavior of one face to the data of another face.

The classic example:

```
Red [Needs: view]

view[
    progress 100x20 20% react [face/data: s/data]
      s: slider 100x20 20%
]
```

The progress bar face reacts to the sliding of the slide face:

/link => Link objects together using a reactive relation.

/unlink => Removes an existing reactive relation.

/later => Run the reaction on next change instead of now.

/with => Specifies an optional face object (internal use).

function! layout

`layout` is used to create custom views without displaying them. You assign your layout to a word, and then, to show or close it, you use `view` or `unview`. With `layout` you can have GUI windows "ready" for specific tasks.

However, it seem it uses the same face tree for both instances, so you cannot create two independent windows like we did above.

The code bellow, for example, will display one window, and only show the other when you close the first.

```
Red [needs: view]

my-view: layout [button {click to "unview"} [unview]]

print "something"  ;do something else
print "biding my time" ;do something else

view/options my-view [offset: 30x100]
view/options my-view [offset: 400x100]
```

Get the size of screen:

```
>> print system/view/screens/1/size
1366x768
```

Check the chapter about system.

Create a full-screen view:

The following script creates a full-screen view:

```
Red [needs: view]

view [size system/view/screens/1/size]
```

system/view/auto-sync?:

From the documentation:

> "The View engine has two different modes for updating the display after changes are done to the face tree:

- Realtime updating: any change to a face is immediately rendered on screen.
- Deferred updating: all changes to a face are not propagated on screen, until `show` is called on the face, or on the parent face."

What this means is that, in the following script, if you uncomment the second line (`on` is default), clicking on "Hello" will not change it to "Good bye" until you click on "Show".

```
Red [needs: view]

{if you uncomment the next line
you will have to click on "Show" after
clicking on "Hello" to turn it into "Good bye"}

;system/view/auto-sync?: off

view [
a: button "Hello" [a/text: "Good bye"]
button "Show" [show a]
]
```

Debugging View:

You may use `system/view/debug?: yes` to see on the console what is happening to your view. Try it. Remember to pass the mouse cursor over the view and do some clicking there:

```
Red []
system/view/debug?: yes
view [button "hello"]
```

< Previous topic Next topic >

GUI - Rich text

Red wiki on rich-text

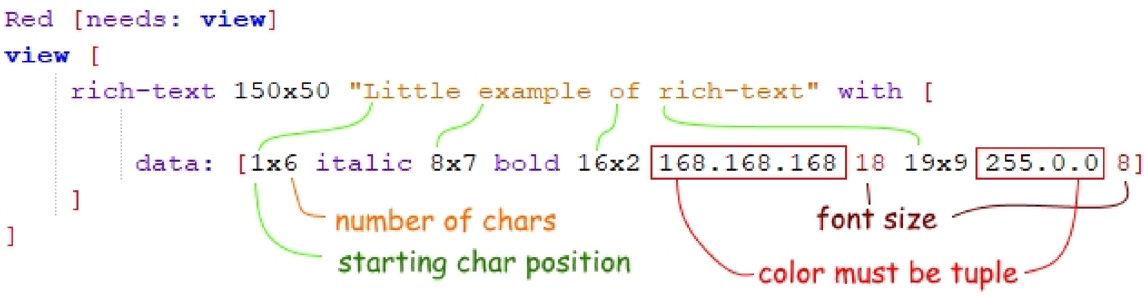
VID DLS rich-text

`rich-text` is a face that can display text in italic, bold, color and with different font sizes. I believe there are two ways of passing the parameters to a rich-text:

First method, using with :

```
Red[needs: view]
view [
    rich-text 150x50 "Little example of rich-text" with [
        data: [1x6 italic 8x7 bold 16x2 168.168.168 18 19x9 255.0.0 8]
    ]
]
```

Explaining first method:



If you don't want to use tuples for colors, you could change the data line to:

```
data: reduce [1x6 'italic 8x7 'bold 16x2 gray 18 19x9 red 8]
```

Second method, using function! **rtd-layout**

`rtd-layout` returns a rich-text face from a RTD source code. I believe it is simpler, and allows you to use rich-text from external sources, but you should read the draw chapter first, and remember to use `compose/deep` in `view`. `compose` evaluates things in parentheses, and it is used to "bring" outside Red code into the `view` dialect block, and must have the `/deep` refinement because the parentheses are nested inside brackets.

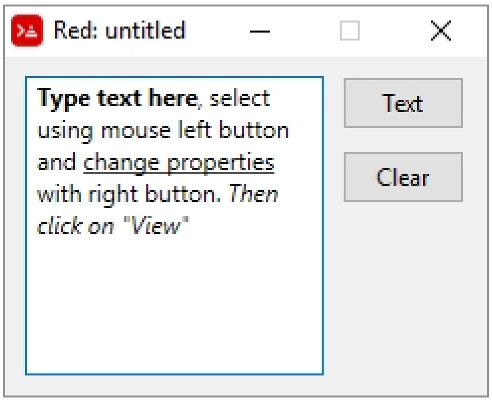
```
Red[needs: view]

myrtf: rtd-layout [i "This " /i b "uses " /b red font 14 "rtd-
layout" /font]

view compose/deep [
    rich-text 200x50 draw [text 0x0 (myrtf)]
    rich-text 200x50 draw [text 20x10 (myrtf)] ;the pair! locates the
text
]
```

Please take a look at Toomas Vooglaid's [rich-text examples page](). With his kind permission, I added a few below. Toomas also has an excellent [gist]() with a variety of Red examples on many topics.

```
Red [
  Author: "Toomas Vooglaid"
]

view [rich-text 200x50 "Little example of rich-text" with [
    data: [1x6 italic 8x10 bold 16x2 168.168.168 19x9 255.0.0 18]]
]
rb: rtd-layout [i "And " /i b "another " /b red font 14 "example" /font]
view compose/deep [rich-text 200x50 draw [text 0x0 (rb)]]
```

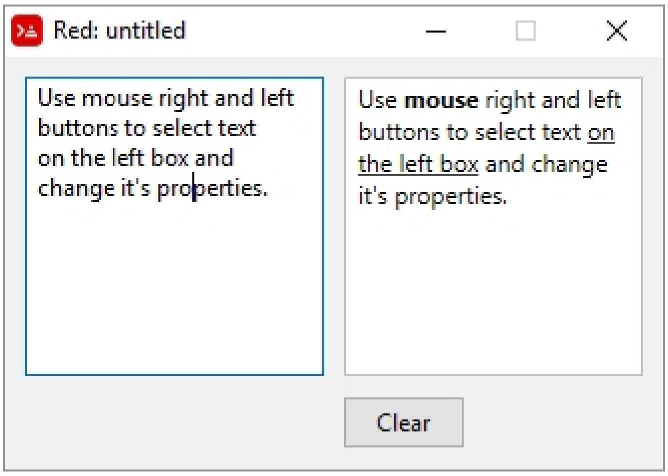

```
Red [
      Purpose: {Relatively simple rich-text demo}
      Help: {Enter text. Select some text, choose formatting from
contextual menu (alt-click).
            Press "View" to see formatting, "Text" to return to text
editing, "Clear" to clear                      formatting.}
]
count-nl: func [face /local text n x][
      n: 0 x: face/selected/x
      text: copy face/text
```

```
        while [all [
                text: find/tail text #"^/"
                x >= index? text
        ]][
                n: n + 1
        ] n
]
view compose [
        src: area wrap with [
                menu: ["Italic" italic "Bold" bold "Underline" underline]
        ]
        on-menu [
                nls: count-nl face
                append rt/data reduce [
                        as-pair face/selected/x - nls face/selected/y -
face/selected/x + 1 event/picked
                ]
        ]
        at 16x12 rt: rich-text hidden with [
                data: copy []
                size: src/size - 7x3
                line-spacing: 15
        ]
        below
        button "View" [
                if show-rt: face/text = "View" [rt/text: copy src/text]
                face/text: pick ["Text" "View"] rt/visible?: show-rt
        ]
        button "Clear" [clear rt/data]
]
```




```
Red [
        Purpose: {Relatively simple rich-text demo}
        Help: {Select some text in first box, choose formatting from
context-menu (alt-click).
                "Clear" clears formatting.}
]
count-nl: func [face /local text n x][
        n: 0 x: face/selected/x
        text: copy face/text
        while [all [
                text: find/tail text #"^/"
                x >= index? text
        ]][
```

```
            n: n + 1
        ] n
]
view compose [
        below src: area wrap with [
            menu: ["Italic" italic "Bold" bold "Underline" underline]
        ]
        on-menu [
            nls: count-nl face
            append rt/data reduce [
                as-pair face/selected/x - nls face/selected/y -
face/selected/x + 1 event/picked
            ]
        ]
        on-key [rt/text: face/text rt/data: rt/data]
        return
        pnl: panel white with [
            size: src/size
            draw: compose [pen silver box 0x0 (size - 1)]
            pane: layout/only compose [
                at 7x3 rt: rich-text with [
                    size: src/size - 10x6 data: copy []
                ]
            ]
        ]
        button "Clear" [clear rt/data]
]
```



< Previous topic

Next topic >

GUI - Create views programmatically

VID is the graphical dialect of Red. All the GUI commands (`base`, `across`, `style`, etc) are VID code.

FACE TREE - the object! of a graphical view. `view` and `show`.can only display this object!

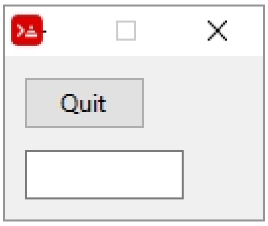
LAYOUT transforms any block containing VID code into a face tree.

VIEW transforms (if needed) a block of VID code into a face tree and display it as a GUI.

SHOW displays a face tree. It can display a `layout` (or even a `view`), but it cannot display a GUI out of a block of VID code. Inside a VID block, it updates a face, however, on Red, unlike Rebol, that update is automatic unless you change settings on `system/view/auto-sync?`, as explained here.

So, the argument for `view` is just a block of VID code and you can change it:

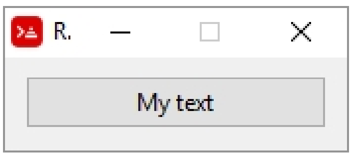
```
Red[needs: view]
board: []
append board [below button "Quit" [quit] field ]
view board
```



Using external variables as facets for a view

The built-in function `compose` evaluates things inside parentheses and you may "pass" parameters to `view` using it:

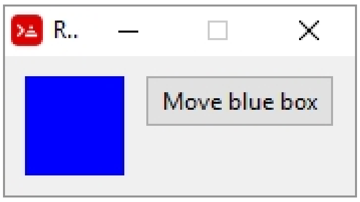
```
Red [needs: view]
txt: "My text"
size: 150
view compose [ button (txt) (size)]
```



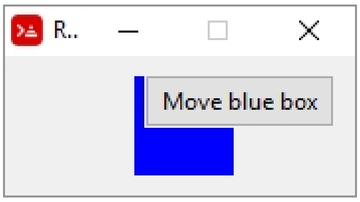
Changing a GUI from the GUI itself

If the GUI is created from a block with `compose` and then rendered by `view`, any change in the values in the block is reflected on the GUI "on the fly":

```
Red[needs: view]

board: compose [
      a: box blue 50x50
      button "Move blue box" [a/offset: (a/offset: a/offset + 5x0)]
]   ; every click increases position of blue box

view board
```



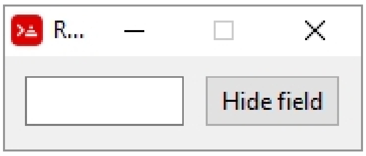
few clicks →



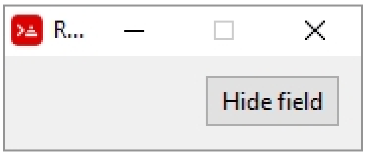
Hiding/showing faces

Faces have the attribute `visible?` that can be changed from `true` (default) to `false` to hide a face. In the following script, click the button to toggle on and off the visibility of the `field`:

```
Red [needs: view]
view [
    f: field
    button "Hide field" [f/visible?: not f/visible?]
]
```

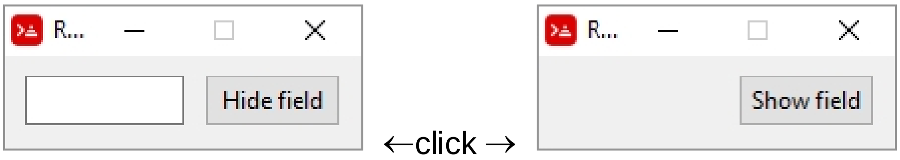


click →



An elegant example (by Toomas Vooglaid):

```
Red[needs: view]

view [
   f: field
   button "Hide field" [
      face/text: pick [
         "Hide field" "Show field"
      ] f/visible?: not f/visible?
   ]
]
```



[< Previous topic]

[Next topic >]

Parse

Very good information also in red-by-example. and in the links in Parse links chapter.

Parse is a "dialect" of Red (a DSL - domain specific language to be precise), that is, a mini-language embedded inside Red. The Red interpreter you download comes with a few of these languages: VID, for GUI creation, DRAW for graphics and PARSE.

Parse should be studied as a small programming language.

native! parse

In a very basic level, `parse` picks each element of the input and submits it to the corresponding rule in the rules block. It returns `true` if all rules are matched or `false`, if one fails to match its corresponding rule.

A most basic example would be to simply check if each element in the input block is equal to the corresponding rule in the rules' block:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block
print parse a ["fox" "dog" "owl" "rat" "elk" "cat"]
```

```
true
```

For the sake of clarity in the description of parse, lets rewrite the example above with a different format:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block

print parse a [   ;here the rules begin:
      "fox"  ; rule 1 matches element 1 => success
      "dog"  ; rule 2 matches element 2 => success
      "owl"  ; rule 3 matches element 3 => success
      "rat"  ; rule 4 matches element 4 => success
      "elk"  ; rule 5 matches element 5 => success
      "cat"  ; rule 6 matches element 6 => success
]
; since all matches are success, the result is "true"
```

```
true
```

The match may be done with datatypes:

```
Red[]
```

```
a:  [33 18.2 #"c" "rat"]      ; input block

print parse a [
      integer!
      float!
      char!
      string!
]
```

```
true
```

Regular code may be inserted in the rules' block by enclosing it in parenthesis:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk"]       ; input block

print parse a [
      "fox"
      "dog"
      "owl"
      (loop 3 [print "just regular code here!"])
      "rat"
      "elk"
]
```

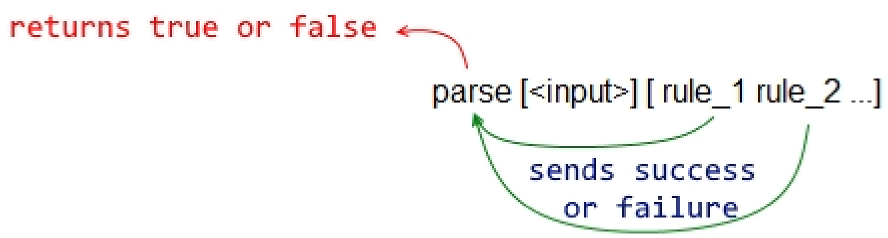
```
just regular code here!
just regular code here!
just regular code here!
true
```

Parse Refinements:

/case =>
/part =>
/trace =>

Important clarification:

`parse` command returns `true` or `false`, but the matching itself passes to `parse` **success** or **failure**. Have that in mind to avoid confusion.



< Previous topic Next topic >

Debugging Parse

Parse dialect is powerful, but it's also hard to visualize and notoriously difficult to debug. Before you proceed to the more advanced features of parse, I suggest you learn how to debug your code. There are two ways that I'm aware of: using the `parse-trace` function and printing information along the evaluation.

function! parse-trace

Parses the input, but also prints (traces) every step of the process.

```
Red[]
a:  ["fox" "owl" "rat"]
print parse-trace a ["fox" "owl" "rat"]
```

```
 -->
   match: ["fox" "owl" "rat"]
   input: ["fox" "owl" "rat"]
   ==> matched
   match: ["owl" "rat"]
   input: ["owl" "rat"]
   ==> matched
   match: ["rat"]
   input: ["rat"]
   ==> matched
return: true
true
```

```
Red[]
a:  ["fox" "owl" "rat"]
print parse-trace a [["fox" | "cow"] "owl" "rat"]
```

```
 -->
   match: [["fox" | "cow"] "owl" "rat"]
   input: ["fox" "owl" "rat"]
   -->
     match: ["fox" | "cow"]
     input: ["fox" "owl" "rat"]
     ==> matched
     match: [| "cow"]
     input: ["owl" "rat"]
   <--
   match: ["owl" "rat"]
   input: ["owl" "rat"]
   ==> matched
   match: ["rat"]
```

```
   input: ["rat"]
   ==> matched
return: true
true
```

print statements:

Put `print` statements in strategic locations to inform the status of the evaluation:

```
Red[]
a:  ["fox" "owl" "rat"]
print parse a ["fox" (print "reached fox")
               "owl" (print "reached owl")
               "rat" (print "reached the end")
              ]
```

```
reached fox
reached owl
reached the end
true
```

< Previous topic Next topic >

Parse - Matching

PARSE skip

Skips (jumps) one element:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"] ; input block

print parse a [   ;here the rules begin:
      "fox"  ; rule 1 matches element 1 => true
      skip   ; just skips this element
      "owl"  ; rule 3 matches element 3 => true
      "rat"  ; rule 4 matches element 4 => true
      "elk"  ; rule 5 matches element 5 => true
      "cat"  ; rule 6 matches element 6 => true
]
```

```
true
```

Another example, noting that strings are series of characters, and are a common input block:

```
Red []
a: "XYZhello"
print parse a [skip skip skip "hello"]
```

```
true
```

Or, more elegantly (check repetition):

```
Red []
a: "XYZhello"
print parse a [3 skip "hello"]
```

```
true
```

PARSE to and PARSE thru

Skips elements until if finds a match. `thru` sets the input is set past the match, `to` sets it before the match.

The next two examples illustrate well the use of `to` and `thru`. They use strings (series of `char!`) as input blocks.

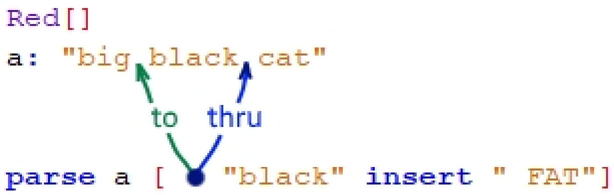
```
Red[]
a: "big black cat"
parse a [ to "black" insert "FAT "]
print a
```

```
big FAT black cat
```

```
Red[]
a: "big black cat"
parse a [ thru "black" insert " FAT"]
print a
```

```
big black FAT cat
```

So:



Example of `to`:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

print parse a [    ;here the rules begin:
      "fox"        ; rule 1 matches element 1 => true
      to "elk"     ; skips all elements until...
                   ; ...it finds a match, but..
      "elk"        ; ... it also checks if the match fits the rule
      "cat"        ; rules for the elements...
      "bat"        ; ... following the match
]
```

```
true
```

Example of `thru`:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat" "bat"] ; input block

print parse a [    ;here the rules begin:
      "fox"        ; rule 1 matches element 1 => true
      thru "elk"   ; skips all elements until...
                   ; ...it finds a match
      "cat"        ; rules for the elements...
      "bat"        ; ... following the match
]
```

```
true
```

PARSE end

Returns `true` if all input items have been checked by parse.

```
Red[]

a:  [33 18.2 #"c" "rat"] ; input block

print parse a [
      integer!
      float!
      char!
      string!
      end
]
```

```
true
```

However, the most common use of `end` is as a reference for `to` and `thru` keywords, to skip all inputs and bring the parse to the end of the input block.

```
Red[]

a:  [33 18.2 #"c" "rat"]
print parse a [to end]  ; just skips to the end, after "rat"
```

```
true
```

PARSE ahead

Checks if the next element (ahead) matches a rule.

```
Red[]

a:  ["fox" "dog" "owl" "rat"] ; input block

print parse a [
      "fox"
      "dog"
      ahead "owl" ;checks if the next item matches the rule
      "owl"
      "rat"
]
```

```
true
```

PARSE none

Always returns sucess. It is a catch-all rule

```
Red[]

a:  ["fox" "dog" "owl" "rat"]  ; input block

print parse a [
      "fox"
      "dog"
      none  ; does nothing, but actions can be inserted here
      "owl"
      "rat"
]
```

```
true
```

PARSE opt

If it finds a match, it returns sucess, and parse follows to the next input. If the input does not match the `opt` rule, parse skips (ignores) this `opt` rule and checks the same input with the next rule.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]  ; input block

print parse a [   ;here the rules begin:
      "fox"   ; rule 1 matches element 1 => success
      "dog"   ; rule 2 matches element 2 => success
      opt "owl"   ; rule 3 matches element 3 => success
      "rat"   ; rule 4 matches element 4 => success
]
print parse a [   ;here the rules begin:
      "fox"   ; rule 1 matches element 1 => success
      "dog"   ; rule 2 matches element 2 => success
      opt "BAT"   ; no "BAT" here in input, so parse just skip this
rule...
      "owl"   ; ...and parse continues here with the next input.
      "rat"   ; rule 4 matches element 4 => success
]
```

```
true
true
```

Another example:

```
Red []
a:  ["Mrs" "Robinson"]
print parse a [opt "Mrs" "Robinson"] ;TRUE

a:  ["Robinson"]
print parse a [opt "Mrs" "Robinson"] ;TRUE, the "Mrs" is OPTional

a:  ["Miss" "Robinson"]
print parse a [opt "Mrs" "Robinson"] ; FALSE, "Mrs" is optional, but
```

```
"Miss" is wrong!
```

Another example:

```
a: [ "elk" "cat" "owl"]

parse a [ opt [ "fig" ] "elk" "cat" "owl" ]  ; never or at least once
true

parse a [ opt [ "elk" "cat" ] "owl" ]      ; never or at least once


true

parse a [ opt [ "elk" "owl" ]  "cat" ]     ; never or at least once


false *
```

* If the entry does not match the **opt** rule , the parse skips this rule and checks the same entry by the following rule.

One more example for **opt** :

```
hd: "mountaintrack"       ; string
parse hd [ opt "mountain" "track"]    ; == true
parse hd [ opt "mountain" "rights"]    ; == false
```

PARSE not

The official definition of the `not` rule is that it "invert the result of the sub-rule". To me, it seems as a rule that excludes a possible match from the next rule.
It does not "consume" input.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]

print parse a [
      "fox"
      "dog"
      not "owl" ;does not consume input
      skip ;anything here, except "owl" - fails!
      "rat"
]
print parse a [
      "fox"
      "dog"
      not "COW" ; does not "consume" input
      skip ;anything here, except "COW" - success!
      "rat"
]
```

```
false
true
```

PARSE quote

Matches the argument exactly as it is except for `paren!`

This gives an error:

```
>> parse [x] [x]
*** Script Error: PARSE - invalid rule or usage of rule: x
*** Where: parse
*** Stack:
```

But this works:

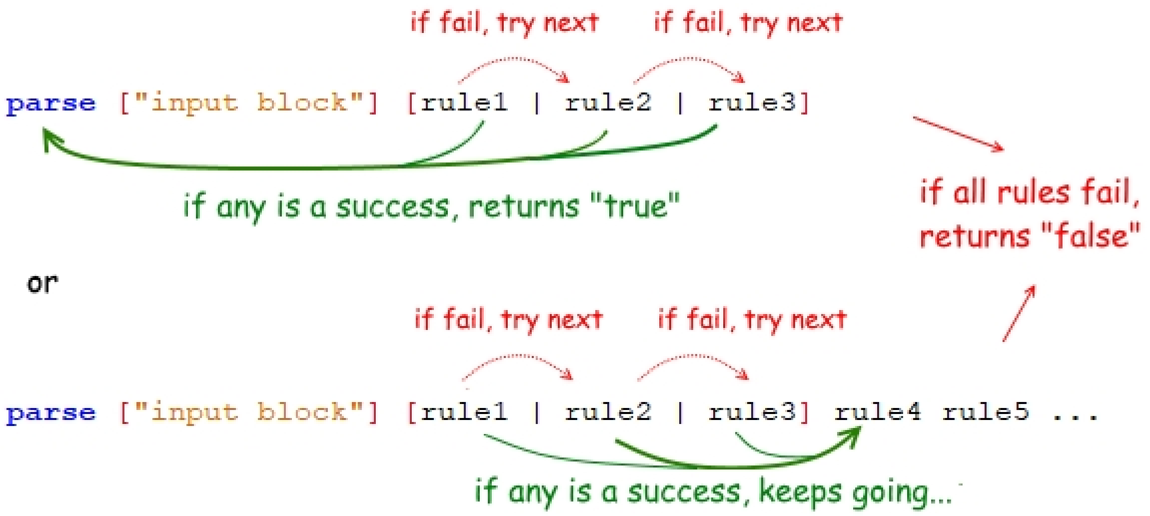
```
>> parse [x] [quote x]
== true

>> parse ['x] [quote 'x]
== true

>> parse [[x]] [quote [x]]
== true
```

< Previous topic Next topic >

Parse - Ordered Choices

Rules accept a "ordered choice" operator, represented by "|":

If a block of rules separated by "|" is found by parse, it will try each rule, from left to right until it finds a match, returning success and going to the next rule after the block. If none of them is a match, of course, it fails and the parsing is stopped returning `false`.



This is similar to a logic "or" operator, but order matters.

Example1:

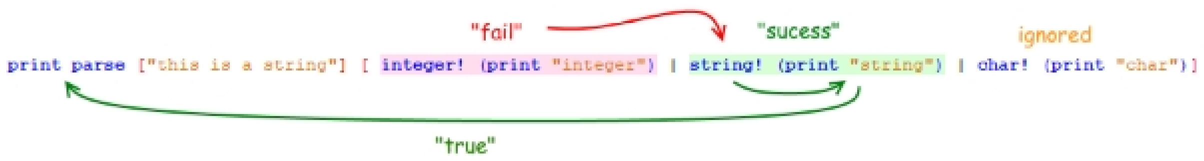
```
Red[]

a: ["fox" "rat" "elk"]
b: ["fox" "owl" "elk"]

print parse a [
      "fox"
      ["rat" | "owl"]         ;notice enclosing brackets
      "elk"
]
print parse b [
      "fox"
      ["rat" | "owl"| "cat" | "whatever"]
      "elk"
]
```

```
true
true
```

Example2:

```
Red[]
print parse ["this is a string"] [ integer! (print "integer") | string!
(print "string") | char! (print "char")]
```

```
string
true
```



Example3:

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [integer! (print "I") | string! (print "S") | time! (print
 "T")]
```

```
S
false
```

Repeating the script with `parse-trace` instead of `print parse` (color highlights, newlines, bold font and comments added by edition):

```
 -->
   match: [integer! (print "I") | string! (print "S") | time
   input: ["string" 3 #"A"]
   ==> not matched

   match: [string! (print "S") | time! (print "T")]
   input: ["string" 3 #"A"]
   ==> matched
                              ;keeps going to execute commands in
parenthesis
   match: [(print "S") | time! (print "T")]
   input: [3 #"A"]
S
   match: [| time! (print "T")]
   input: [3 #"A"]
return: false                 ;too much input and not enough rules ->
false
```

To obtain `true`, we may add more rules to the successful ordered choice...

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [integer! (print "I") | string! (print "S") integer! char!
| integer! (print "T")]
```

```
S
true
```

... or enclose the ordered choices in brackets and add rules to the main rule block:

```
Red[]
a: ["string" 3 #"A"] ; that is a string!, an integer! and a char!
print parse a [[integer! (print "I") | string! (print "S") | time!
(print "T")] integer! char!]
```

```
S
true
```

[< Previous topic] [Next topic >]

Parse - Repetition and Matching loops

Keywords: some, any, opt, while.

Rule rule can be optional or repeated in a different way.

Keyword or Value	Description
3 <rule>	repeat the rule 3 times
1 3 <rule>	repeat rule 1 to 3 times
0 3 <rule>	repeat the rule 0 to 3 times
some	repeat its rule(s) while (and if) it gets a `true` (match) from the rule. Returns `false` if it doesn't get at least one match (makes the parse `false`).
any	repeat its rule(s) until it gets a `false` (no match) from the rule. Always returns `true` to the parse expression.
while	see text below.

Known Repetition Number - Examples

```
>> parse "fogfogfog" [3 "fog"]; determined exactly
== true
```

```
>> parse "fogfogfog" [0 5 "fog"]; determined by range
== true
```

Script examples for exact repetitions:

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
    4 skip  ; see command skip at Parse/Matching
      "elk"
      "cat"
]
```

```
true
```

```
Red[]

a:  ["rat" "rat" "rat" "rat" "elk" "cat"]

print parse a [
    4 "rat"
      "elk"
      "cat"
]
```

```
true
```

Or a range:

```
Red[]

a:  ["rat" "rat" "elk" "cat"]

print parse a [
    0 4 "rat"     ; will return success if there is from zero up to four
"rat"
      "elk"
      "cat"
]
```

```
true
```

Matching Loops:

PARSE some, PARSE any

Again:

`some` **-** repeat its rule(s) while (and if) it gets a true (match) from the rule. Returns `false` if it doesn't get at least one match (makes the parse false).

`any` **-** repeat its rule(s) until it gets a false (no match) from the rule. Always returns true to the parse expression

Both return success for as long as they find matches in the input, the difference is that `some` requires at least one occurrence of the input (match), while `any` will return success even with no match.

```
Red[]

a:  ["fox" "dog" "fox" "dog" "fox" "dog" "elk" "cat"]

print parse a [
      some ["fox" "dog"]
```

```
        "elk"
        "cat"
]

print parse a [
        any ["fox" "dog"]
        "elk"
        "cat"
]
```

```
true
true
```

```
Red[]

a:  ["elk" "cat"]

print parse a [
        some ["fox" "dog"]
        "elk"
        "cat"
]

print parse a [
        any ["fox" "dog"]
        "elk"
        "cat"
]
```

```
false
true
```

Example that shows the "loop" behavior more clearly:

```
Red []
txt: {In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?}

print parse txt [some [thru "blue" (print "I found blue!")] to end]
```

```
I found blue!
I found blue!
I found blue!
I found blue!
true
>>
```

Explaining the example:

```
[some
      [thru "blue" (print "I found blue!")] ; this rule will be repeated while
if finds a match
to end]
```

- first loop:

```
In a one-story blue house, there was a blue person,
```

```
a blue cat - everything was blue! What color were the stairs?
```
-> found a match, so repeat `[thru "blue" (print "I found blue!")]`

- second loop:
```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```
-> found a match, so repeat `[thru "blue" (print "I found blue!")]`

- third loop:
```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```
-> found a match, so repeat `[thru "blue" (print "I found blue!")]`

- fourth loop:
```
In a one-story blue house, there was a blue person,
a blue cat - everything was blue! What color were the stairs?
```
-> found a match, so repeat `[thru "blue" (print "I found blue!")]`

-> NO match, so exits `some` loop and goes for the next rule: `to end`, which is a match, because it simply goes to the end.

Since all rules found a match (`some` found more than one), parse returns `true`.

PARSE while

Definitely not for beginners, as kindly explained by Vladimir Vasilyev (@9214) from gitter:

"

```
>> parse x: [a 1 a 1][while [ahead ['a change quote 1 2] | 'a quote 2]]
== true

>> x
== [a 2 a 2]

>> parse x: [a 1 a 1][any [ahead ['a change quote 1 2] | 'a quote 2]]
== false

>> x
== [a 2 a 1]
```

The main difference between `while` and `any` is that the former continues parsing even if index did not advance after successful match, while the latter fails as soon as index remained at the same position, even if match was successful.

That's why I went with `ahead` - it's a look-ahead rule, that matches "in advance", but keeps index where it is. In the example above, `ahead ['a change quote 1 2]` will match successfully, and 1 after a will be changed to 2, *but the input position will not advance, because ahead looks ahead, while standing where it is now.* Outcomes are:

- With `while`, first `ahead ...` changes 1 to 2 without advancing the input, but since `while` doesn't care about that, it goes to the next iteration, on which top-level rule will fail and backtrack (an alternate after `|`) to `'a quote 2`, which will match (because we've just changed `a 1` to `a 2` and advance the input, thus leading us to the `end` marker and successful parsing of the whole series.

- With `any`, however, first `ahead ...` changes `1` to `2`, does not advance the input, and `any`, because it's picky about input advancing, bails out without going to the second iteration.

The use-case for `while` is a tricky one. In my experience, I used it for context-sensitive parsing (that is, you first look behind and ahead, determining the *context* of a token, and only then decide what to do; "looking behind and ahead *requires matching various rules* while standing where you are, at current position*) and also in situations where input needs to be modified during parsing (example above), or if parsing depends on some outside state. It's also proved to be useful for deep-first traversal of tree-like structures - situation is the same, you're tinkering with node, matching some rules successfully, but the position should not advance if you've matched something, otherwise you'll loose the track of the current node.

That is, `while` is anything but newbie-friendly. I'd noted in your tutorials that you shouldn't worry about it if you're a newcomer, and that it is useful in advanced situations, where you need more tight control over parsing."

< Previous topic Next topic >

Parse - Storing input

PARSE set and PARSE copy

Both get the input of the **next** parse rule, if successful. The difference happens when you have a subexpression (see examples below). The `set` operation sets the given variable to the first matched value, while the `copy` operation copies the whole part of the input matched by the subexpression.

```
Red[]

a: ["fox" "rat" "elk"]

 parse a [
      "fox"
      set b       ;ready to assign if next rule is successful. Could use
copy instead.
      "rat"       ;success here, so "rat" => b
      "elk"
]
print b
```

```
rat
```

```
Red[]
block: [7 9]
print parse block [set value integer! integer!]
print value
```

```
true
7
```

```
Red[]
block: [6 3]
print parse block [integer! copy value integer!]
print value
```

```
true
3
```

Explaining the code:

```
Red[]
block: [6 3]
print parse block [integer! copy value integer!
print value
```

```
true
3
```

Showing the difference between `copy` and `set`:

`Set` gets only the first match of a subexpression:

```
Red[]
a: ["cat" "dog" "bat" "owl"]
parse a ["cat" set b any string!]
print b
```

```
dog
```

`Copy` gets all the matches of a subexpression:

```
Red[]
a: ["cat" "dog" "bat" "owl"]
parse a ["cat" copy b any string!]
print b
```

```
dog bat owl
```

PARSE **collect** and PARSE **keep**

If you have a collect block inside your rules' block, `parse` will no longer return a logical `true` or `false`, instead it will return a block with all the successes that preceded by the built-in function (command) `keep` .

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block

 print parse a [
      collect[
      keep "fox"  ; success, WILL be kept
      "dog"
      "owl"
      keep "rat"  ; success, WILL be kept
      keep "cow"  ; FAIL! will NOT be kept
      "cat"
            ]
]
```

```
fox rat
```

PARSE collect set

parse will return a logical true or false, and insert all the successes preceded by the word keep in a new block.

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block

 print parse a [
     collect set b [                           ; creates b to store
keeps
     keep "fox"  ; success, WILL be kept
     "dog"
     "owl"
     keep "rat"  ; success, WILL be kept
     keep "cow"  ; FAIL! will NOT be kept
     "cat"
            ]
]

print b
```

```
false
fox rat
```

PARSE collect into

parse will return a logical true or false, and insert all the successes preceded by the word keep in a block you previously created. It seems to append results to the block.

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]       ; input block
b:""                                           ; must create block
first

 print parse a [
     collect into b [
     keep "fox"  ; success, WILL be kept
     "dog"
     "owl"
     keep "rat"  ; success, WILL be kept
     keep "cow"  ; FAIL! will NOT be kept
     "cat"
            ]
]

print b
```

```
false
foxrat
```

Collecting the input using set-word syntax

During `parse` processing, you may assign what is left of the input to a word (variable):

```
Red[]

a:  ["fox" "dog" "owl" "rat" "elk" "cat"]

print parse a [
      "fox"
      "dog"
      b:
]
probe b
```

```
false
["owl" "rat" "elk" "cat"]
```

```
Red []
txt: "They are one person, they are two together"
parse txt [thru "person, " b:]
print b
```

```
they are two together
```

< Previous topic   Next topic >

Parse - Modifying input

PARSE insert

Inserts a value in the input block at the current input position.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
       "fox"
       "dog"
       insert 33
       "owl"
       "rat"
]
print a
```

```
true
fox dog 33 owl rat
```

Another example using a string:

```
Red[]
a: "My big eyes"
parse a [ thru "big" insert " brown"]
print a
```

```
My big brown eyes
```

PARSE remove

Removes the matched input from the input block.

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
       "fox"
       remove "dog"
       remove "owl"
       "rat"
]
print a
```

```
true
fox rat
```

Another example, using strings:

```
Red[]
a: "My big eyes"
parse a [ to "big" remove "big "]
print a
```

```
My eyes
```

PARSE change

Changes a matched input:

```
Red[]

a:  ["fox" "dog" "owl" "rat"]
 print parse a [
      "fox"
      "dog"
      change "owl" "COW"
      "owl"
      "rat"
]
print a
```

```
false
fox dog COW rat
```

< Previous topic   Next topic >

Parse - Control flow

PARSE if

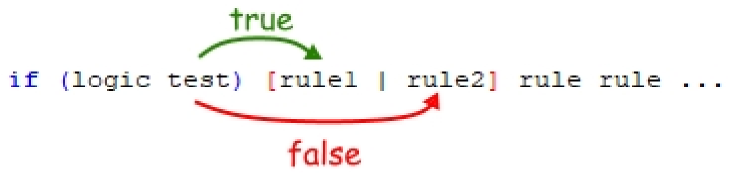
`if` tests the result of a logic expression within parenthesis. It is usually followed by a `rule1 | rule 2` .

If there is no ordered choice (`rule1 | rule 2`) after the `if`, and the result of the logic expression is `false` or `none` the parsing is halted, returning `false`.

```
Red[]
block: [6 3 7]
print parse block [integer! integer! if (1 = 1) integer!] ;(1 = 1) is true, so it goes on
print parse block [integer! integer! if (1 = 2) integer!] ;(1 = 2) is false, so it halts, returning false
```

```
true
false
```

With ordered choices: If the result of this logic expression is `true`, the parsing loop uses `rule1`, if it's `false` or `none`, it uses `rule2` for the next parsing match attempt.




```
Red[]
block: [6 3 7]
print parse block [integer! integer! if (1 = 1) [integer! | string!]] ; 7 is an integer! -> true
print parse block [integer! integer! if (1 = 2) [integer! | string!]] ; 7 is not a string! false
```

```
true
false
```

Another simple example:

```
Red[]
block: [1 2]
print parse block [set value integer! if (value = 1) to end]
block: [2 2]
print parse block [set value integer! if (value = 1) to end]
```

```
true
false
```

PARSE then

Regardless of failure or success of what follows, skip the next alternate rule. That is, when a `then` is encountered, the next alternate rule is disabled.

I couldn't find good examples and can't think of any use for that.

PARSE into

Switch input to matched series (string or block) and parse it with rule.
Could not find good examples.

PARSE fail

Force current rule to fail and backtrack.
Could not find good examples. I believe it is related mostly, if not completely, related to matching loops (any, some and while).

PARSE break

Break out of a matching loop, returning success.
Could not find good examples. I believe it is related mostly, if not completely, related to matching loops (any, some and while), specifically to offer a way to avoid endless loops.

PARSE reject

Break out of a matching loop, returning failure.
Could not find good examples. I believe it is related mostly to matching loops (any, some and while)

< Previous topic Next topic >

Parse usage - Validate inputs

Validating alphanumeric entries:

Before we proceed, I should warn you that the datatyping of Red may cause some trouble to programming. For example, a single-digit number in Red may be an `integer!`, a `string!`, a `char!`, or something else. So if you have some inexplicable bugs in your script, make sure your debugging checks the datatypes being parsed.

Here is a script that prompts the user to enter 4 single digit numbers and check if the entry is OK until the entry is "q":

```
Red []
entry: ""
while [entry <> "q"] [
      entry: ask "Enter four digits in the 1-8 range: "
      either (parse entry [some ["1" | "2" | "3" | "4" | "5" | "6" | "7"
| "8"]]) and ((length? entry) = 4) [
            print "OK"]
            [
            print "Not OK!"
            ]
]
```

That works, but `["1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"]` may be substituted for `charset ["12345678"]`:

```
Red []
entry: ""
validchar: charset ["12345678"]
while [entry <> "q"] [
      entry: ask "Enter four digits in the 1-8 range: "
      either (parse entry [some validchar]) and ((length? entry) = 4) [
            print "OK"]
            [
            print "Not OK!"
            ]
]
```

Since parse checks character by character, `charset ["12345678"]` may also be written as `charset [#"1" - #"8"]`! Red understands that that is a sequence of characters. So, for example, your program may be made to accept any numeric and lower case ASCII characters by using `charset [#"0" - #"9" #"a" - #"z"].`

Crude phone number validator (from Rebol/Core manual) - Rules referring to rules:

```
Red []
digits: charset "0123456789"
area-code: ["(" 3 digits ")"]
phone-num: [3 digits "-" 4 digits]

print parse "(707)467-8000" [[area-code | none] phone-num]
```

```
true
```

Crude email validator (from Red blog):

```
Red []

digit:   charset "0123456789"
letters: charset [#"a" - #"z" #"A" - #"Z"]
special: charset "-"
chars:   union union letters special digit
word:    [some chars]
host:    [word]
domain:  [word some [dot word]]
email:   [host "@" domain]

print parse "john@doe.com" email
print parse "n00b@lost.island.org" email
print parse "h4x0r-l33t@domain.net" email
```

```
true
true
true
```

Validating math expressions in string form (from Rebol/Core manual):

Notice that this example uses recursing rules (a rule that refer to itself).

```
Red []

expr:    [term ["+" | "-"] expr | term]
term:    [factor ["*" | "/"] term | factor]
factor:  [primary "**" factor | primary]
primary: [some digit | "(" expr ")"]
digit:   charset "0123456789"

print parse "1+2*(3-2)/4" expr        ; will return true
print parse "1-(3/)+2" expr           ; will return false
```

```
true
false
```

< Previous topic Next topic >

Parse usage - Extract data

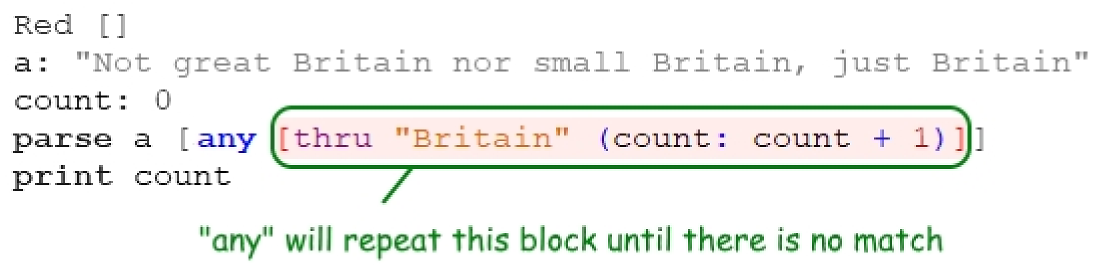
Counting words on text :

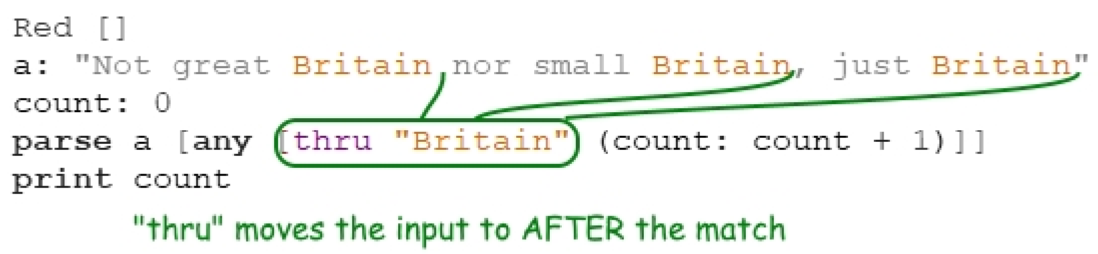
```
Red []
a: "Not great Britain nor small Britain, just Britain"
count: 0
parse a [any [thru "Britain" (count: count + 1)]]
print count
```

```
3
```

Explaining the code:

As long as `thru "Britain"` finds a "Britain", `any` will repeat the rule





Notice that if you used `to` instead of `thru`, the input would be moved to BEFORE the match, creating an **endless loop**, since "Britain" would be a match over and over again.

Extracting the middle part of a text :

To extract the remaining part of a text from a given point, you may use `word:` , as explained in the Storing Input chapter. To extract text between two parse matchings, you may use `copy` :

```
Red []
txt: "They are one person, they are two together"
parse txt [thru "person, " copy b to " two"]
print b
```

```
they are
```

Extract data from the Internet:

This is a very basic example. I have created an html page at helpin.red: http://helpin.red/samples/samplehtml1.html. The html is very simple and you can see it by typing `print read http://helpin.red/samples/samplehtml1.html` at the console. Since I know the html, I can extract some information with the code below:

```
Red []
txt: read http://helpin.red/samples/samplehtml1.html
parse txt [
      thru "today"
      2 thru ">"
      copy weather1 to "<"
      thru "tomorrow"
      2 thru ">"
      copy weather2 to "<"
      thru "week"
      2 thru ">"
      copy weather3 to "<"
]
print {Acording to helpin.red website weather will be: }
print [] ; just adding an empty line
print ["Today:     "  weather1]
print ["Tomorrow:  "  weather2]
print ["Next week: "  #"^(tab)"  weather3] ; just showing the use of tab
```

```
Acording to helpin.red website weather will be:

Today:      sunny
Tomorrow:   horrible
Next week:        really really horrible
```

I will show how the parsing works for extracting the weather of "today" to the "weather1" variable:

```
thru "today" ; skips all text until after a "today" string.
```

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
   <tr>
     <td style="color: black;">weather today:</td>
     <td style="color: black;">sunny</td>
   </tr>
   <tr>
```

```
2 thru ">" ;this skips text until (after) the character ">". Does it 2 times!
```

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
   <tr>
     <td style="color: black;">weather today:</td>        ; 1
     <td style="color: black;">sunny</td>                 ; 2
   </tr>
   <tr>
```

```
copy weather1 to "<" ; this copies to "weather1" all that it finds until
(before) a "<".
```

```
border="1" cellpadding="2" cellspacing="2">
 <tbody>
   <tr>
     <td style="color: black;">weather today:</td>
     <td style="color: black;">sunny</td>                  ; ==>
weather1
   </tr>
   <tr>
```

[< Previous topic]   [Next topic >]

Parse usage - Manipulating text

Inserting words in text:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [any [to "Britain" insert "blue " skip]]
print a
```

```
Not great blue Britain nor small blue Britain, just blue Britain
```

Notice that `skip` was added to the rule to prevent an endless loop: `to` takes the input to before the match, so "Britain" would be matched over and over again if we dont skip it.

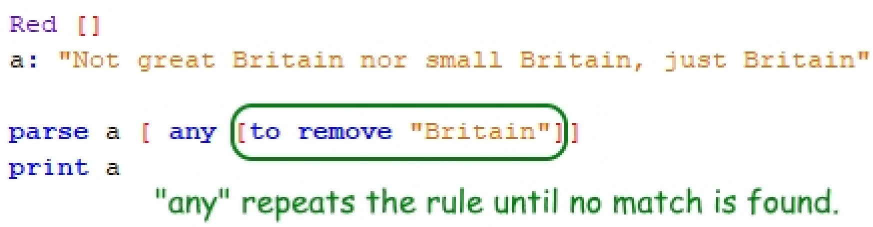
Removing words from text:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [ any [to remove "Britain"]] ;seems to work the same as [to
"Britain" remove "Britain"]
print a
```

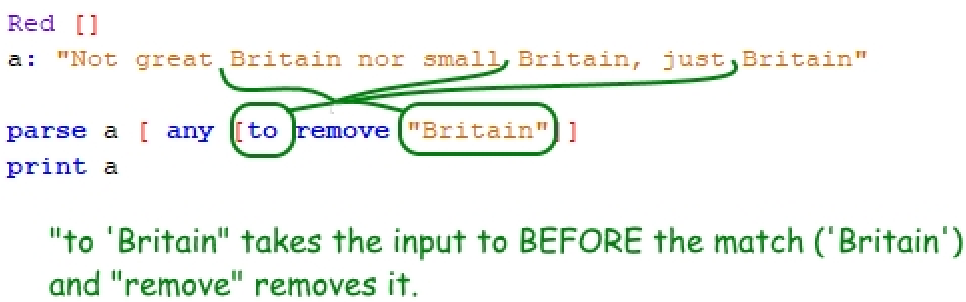
```
Not great  nor small , just
```

Explaining the code:

First:



Then:



Changing words from text:

```
Red []
a: "Not great Britain nor small Britain, just Britain"
parse a [ any [to "Britain" change "Britain" "Australia"]] ;[to change
"Britain" "Australia"] also seems to work!
print a
```

```
Not great Australia nor small Australia, just Australia
```

< Previous topic

Next topic >

Links to pages that may help you to learn how to use parse:

Red specific links:

http://www.red-by-example.org/parse.html - Maybe the best resource available.

http://www.red-lang.org/2013/11/041-introducing-parse.html

http://www.michaelsydenham.com/reds-parse-dialect/

https://github.com/red/red/issues/3478 - Not what you expect, but informative anyway. Discusses issues of parse.

The following links refer to Parse in Rebol :

http://video.respectech.com - with interactive editor.

http://www.rebol.com/docs/core23/rebolcore-15.html

http://www.codeconscious.com/rebol/parse-tutorial.html

http://www.codeconscious.com/rebol/r2-to-r3-parse.html

http://www.rebol.com/r3/docs/concepts/parsing-summary.html - very informative.

http://www.rebol.com/r3/docs/functions/parse.html

http://blog.hostilefork.com/why-rebol-red-parse-cool/

https://en.wikibooks.org/wiki/Rebol_Programming/Language_Features/Parse/Parse_expressions

http://rebol2.blogspot.com/2012/05/text-extraction-with-parse.html

https://github.com/revault/rebol-wiki/wiki/Parse-Project

http://www.colellachiara.com/soft/Misc/parse-rep.html - proposals for improvements of parse

< Previous topic Next topic >

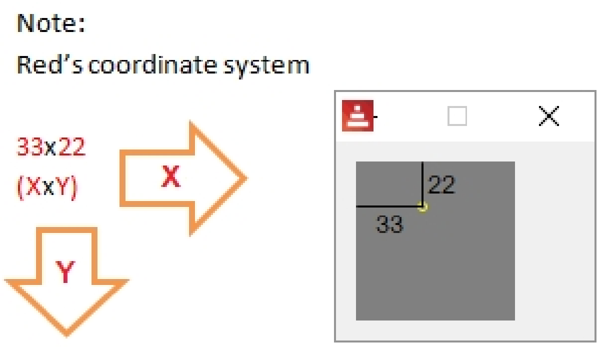
Draw

Very good information also in red-by-example. and in the Red documentation.

Draw is used to create 2D graphics. Like PARSE and VID, Draw is a DSL, that is, a dialect of Red, a language within a language.

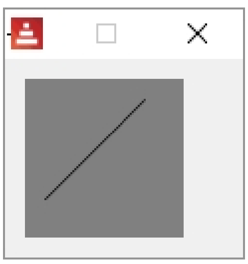
To use `draw`, you must also use VID, so every script that uses `draw` must have a `view` block, and within the `view` block, one must have a `base` face to draw on. The following examples show all the basic shapes of draw.

Remembering:

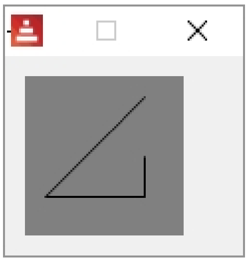


DRAW line

```
Red [needs: view]
view [
	base draw [line 60x10 10x60]
]
```



```
Red [needs: view]
view [
	base draw [line 60x10 10x60 60x60 60x40]
]
```



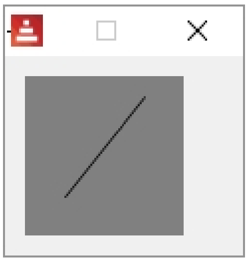
The importance of native! compose for DRAW

Suppose you want to perform evaluations on DRAW arguments, like:

```
Red [needs: view]
view [
      base draw [line 60x10 (2 * 10x30)]
]
```

This is a very common situation, but Red will give you an **error** because **DRAW does not evaluate expressions**.
So you need to use `compose`, most commonly with the refinement `/deep` , to achieve that.

```
Red [needs: view]
view compose/deep [
      base draw [line 60x10 (2 * 10x30)]
]
```



DRAW is part of the face object!

Open the Red GUI console ant type `view/no-wait [a: base draw [line 60x10 10x60]]` .
Then type `? a`. You will see a lot of data about the object a, among them you will see:

```
>> view/no-wait [a: base draw [line 60x10 10x60]]
== make object! [
    type: 'window
    offset: 636x360
    size: 130x100
    text: "Red: untitled"
 ...
>> ? a
A is an object! with the following words and values:
     type                word!          base

<...>
```

```
<...>
     draw               block!        length: 3  [line 60x10 10x60]
     on-change*         function!     [word old new /local srs same-pane?
f saved]
     on-deep-change*    function!     [owner word target action new index
part]
```

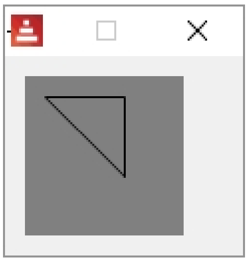
So you may access the `draw` block using path!:

```
>> a/draw
== [line 60x10 10x60]
```

This is very important for animation - programmatic drawing.

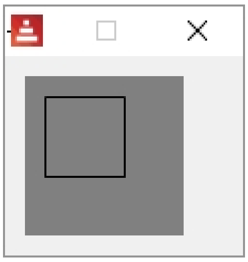
DRAW triangle

```
Red [needs: view]
view [
      base draw [triangle 10x10 50x50 50x10]
]
```



DRAW box

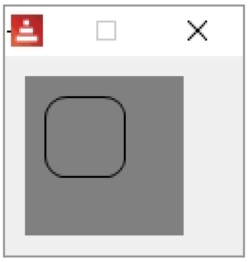
```
Red [needs: view]
view [
      base draw [box  10x10     50x50]
;                    top left  bottom-right
]
```



with a rounded corner:

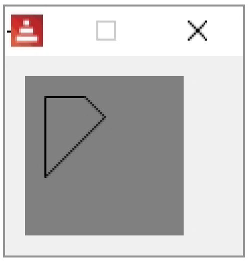
```
Red [needs: view]
view [
      base draw [box     10x10        50x50      10]
```

```
    ;                  top left    bottom-right     corner-radius
]
```



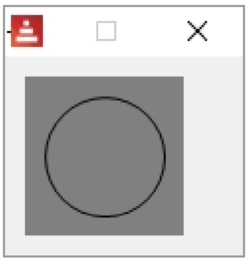
DRAW polygon

```
Red [needs: view]
view [
      base draw [polygon 10x10 30x10 40x20 30x30 10x50]
      ; it closes the polygon automatically
]
```



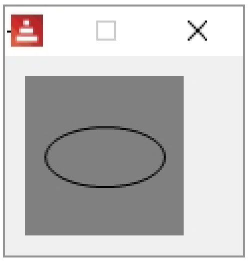
DRAW circle

```
Red [needs: view]
view [
      base draw [circle  40x40     30]
                 ;      center    radius
]
```



ellipse mode:

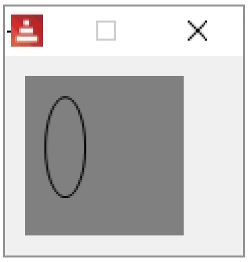
```
Red [needs: view]
view [
      base draw [circle  40x40     30        15 ]
                 ;     center   radius-x  radius-y
]
```



DRAW ellipse

The ellipse is drawn within an imaginary rectangle. The arguments are the box top-left point and the other corner's point

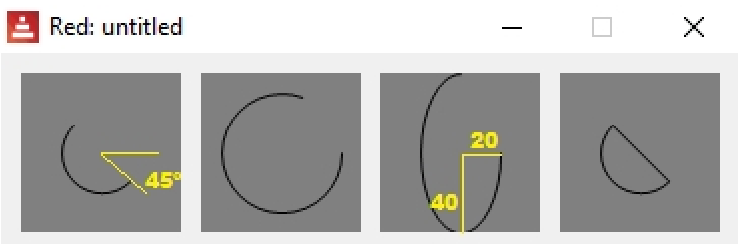
```
Red [needs: view]
view [
      base draw [ellipse 10x10 20x50]
]
```



DRAW arc

Draws the arc of a circle from the provided center (pair!) and radius (also a pair!) values. The arc is defined by two angles values in degrees. An optional `closed` keyword can be used to draw a closed arc using two lines coming from the center point.

```
Red [needs: view]
view [
      base draw [arc 40x40       20x20            45             180]
              ;      center   radius-x/radius-y   start angle   finish angle
      base draw [arc 40x40 30x30 0 290]
      base draw [arc 40x40 20x40 0 270]
      base draw [arc 40x40 20x20 45 180 closed]
]
```



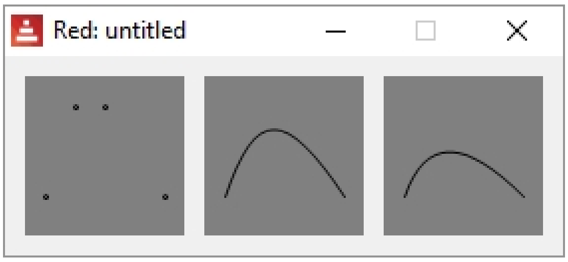
DRAW curve

Draws a Bezier curve from 3 or 4 points:

- 3 points: 2 end points, 1 control point.
- 4 points: 2 end points, 2 control points.

The 4 points option allow more complex curves to be created.

```
Red [needs: view]
view [
      ;first we just show 4 points:
      base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle
70x60 1]
      ;then the curves:
      ;4 points-  startpoint controlpoint1 controlpoint2 endpoint:
      base draw [curve 10x60 25x15 40x15 70x60]
      ;3 points- startpoint controlpoint endpoint:
      base draw [curve 10x60 25x15       70x60]
]
```

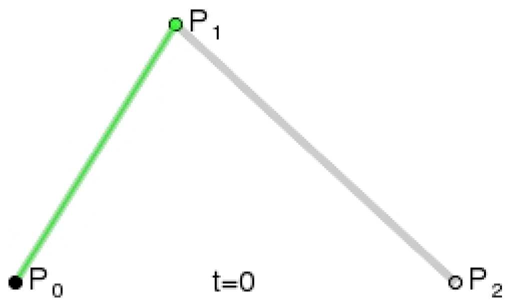


Bezier curves

Bezier curves have a start point, an endpoint and one or two control points. If it has one control point its a quadratic Bezier, if it has two control points its a cubic Bezier.

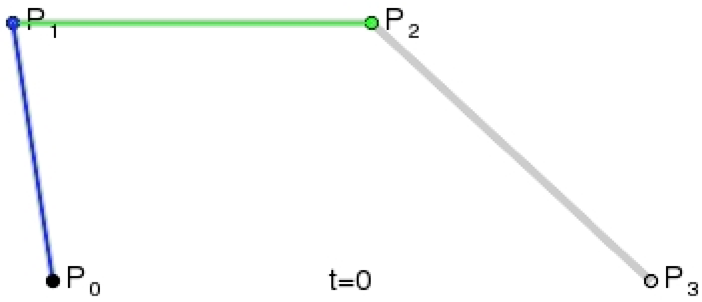
The following animated gifs were made by Phil Tregoning and released to public domain (thank you) at Wikimedia Commons. If you can't see the animation, check the page on Wikipedia about Bezier curves :

Quadratic Bezier:



You should also check out this great interactive demonstration.

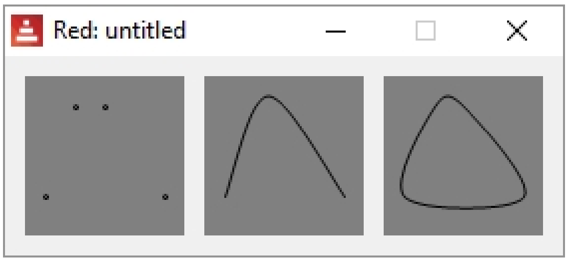
Cubic Bezier:



DRAW spline

Constructs a curve that follows a sequence of points.

```
Red [needs: view]
view [
      ;first we just show 4 points:
      base draw [circle 10x60 1 circle 25x15 1 circle 40x15 1 circle
70x60 1]
      ;then the splines:
      base draw [spline 10x60 25x15 40x15 70x60]
      base draw [spline 10x60 25x15 40x15 70x60 closed]
]
```

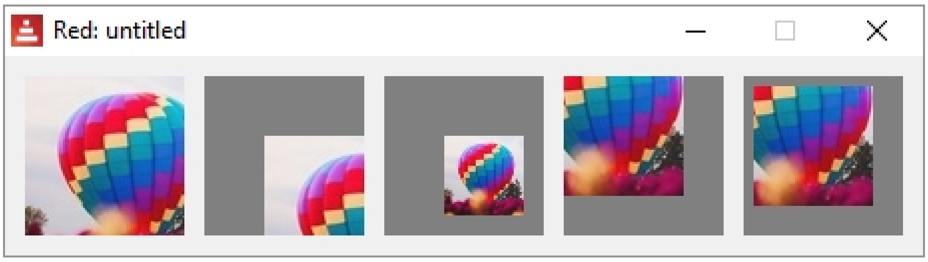


DRAW image

Paints an image using the provided information for position and width.

```
Red [needs: view]
; image command expects a image! not a file!
; so you must first load the file
picture: load %smallballoon.jpeg
view [
      base draw [image picture]
      base draw [image picture 30x30]
      base draw [image picture 30x30 70x70]
      base draw [image picture crop 30x30 60x60]
      base draw [image picture 5x5 crop 30x30 60x60]
```

```
]
```

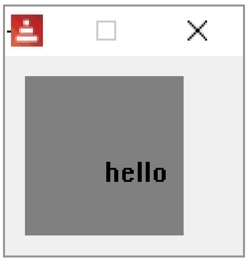


There is also a `color` command (key color to be made transparent) and a `border` command, but I couldn't make those work yet.

```
;base draw [image picture 30x30 70x30 30x70 70x70]
;base draw [image picture 30x30 70x70 red]
;base draw [image picture 30x30 70x70 blue border]
```

DRAW text

```
Red [needs: view]
view [
	base draw [text 40x40 "hello"]
]
```



DRAW font

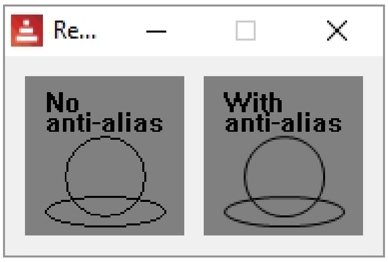
?

DRAW anti-alias

Anti-aliasing gives nicer visual rendering, but degrades performance. It can be set on (default) or off.

```
Red [needs: view]
view [
	base draw [
		anti-alias off
		text 10x5 "No"
		text 10x15 "anti-alias"
		circle 40x50 20
		ellipse 10x60 60x15
```

```
        ]
        base draw [
            anti-alias on ; this is the default
            text 10x5 "With"
            text 10x15 "anti-alias"
            circle 40x50 20
            ellipse 10x60 60x15
        ]
]
```



DRAW shape

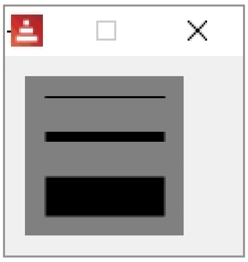
See the [Shape sub-dialect page](#).

[< Previous topic](#) [Next topic >](#)

DRAW - Line properties

DRAW line-width

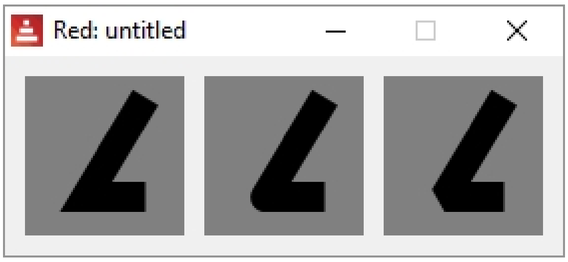
```
Red [needs: view]
view [
        base draw [
                line-width 1
                line 10x10 70x10
                line-width 5
                line 10x30 70x30
                line-width 20
                line 10x60 70x60
        ]
]
```



DRAW line-join

May be `miter`, `round` , `bevel` or miter-bevel*. `miter` is default

```
Red [needs: view]
view [
        base draw [
                line-width 15
                line-join miter
                line 60x10 30x60 60x60
        ]
        base draw [
                line-width 15
                line-join round
                line 60x10 30x60 60x60
        ]
        base draw [
                line-width 15
                line-join bevel
                line 60x10 30x60 60x60
        ]
]
```

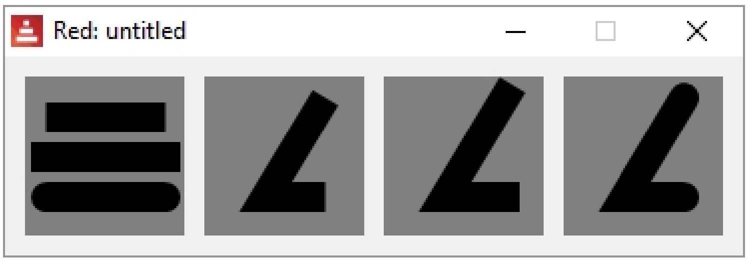


* I could not make the miter-bevel option work.

DRAW line-cap

Defines the line ending's cap mode. May be `flat` (default) `square` or `round`.

```
Red [needs: view]
view [
	base draw [
		line-width 15
		line-cap flat ;default
		line 10x20 70x20
		line-cap square
		line 10x40 70x40
		line-cap round
		line 10x60 70x60
	]
	base draw [
		line-width 15
		line-cap flat ;default
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-cap square
		line 60x10 30x60 60x60
	]
	base draw [
		line-width 15
		line-cap round
		line 60x10 30x60 60x60
	]
]
```

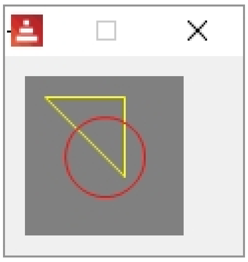


< Previous topic Next topic >

DRAW - Color, gradients and patterns

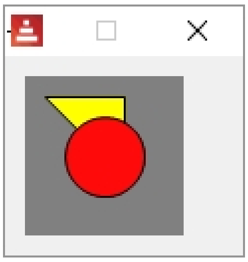
DRAW **pen** <color>

```
Red [needs: view]
view [
	base draw [
		pen yellow ; color as word!
		triangle 10x10 50x50 50x10
		pen 255.10.10 ; color as tuple!
		circle 40x40 20
	]
]
```



DRAW **fill-pen** <color>

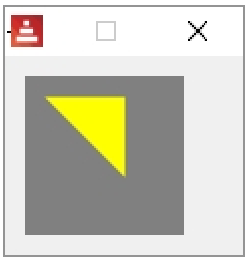
```
Red [needs: view]
view [
	base draw [
		fill-pen yellow ; color as word!
		triangle 10x10 50x50 50x10
		fill-pen 255.10.10 ; color as tuple!
		circle 40x40 20
	]
]
```



Turning off the pen and the fill-pen:

```
Red [needs: view]
```

```
view [
      base draw [
            pen off
            fill-pen yellow ; color as word!
            triangle 10x10 50x50 50x10
            fill-pen off
            circle 40x40 20
      ]
]
```



DRAW **linear** - linear color gradient

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> linear <color1> <offset> ... <colorN> <offset> <start>
<end> <spread>

<color1/N>       : list of colors for the gradient (tuple! word!).
<offset>         : (optional) offset of gradient color (float!).
<start>          : (optional) starting point (pair!).
<end>            : (optional unless <start>) ending point (pair!).
<spread>         : (optional) spread method (word!).
```

Description

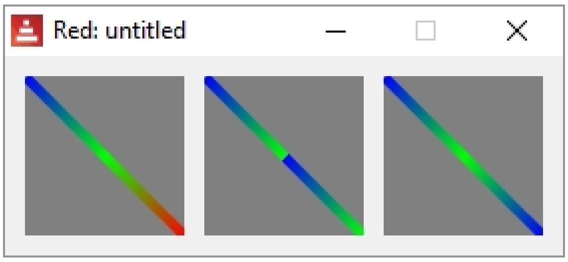
Sets a linear gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same as `repeat` for Windows platform).

When used, the start/end points define a line where the gradient paints along. If they are not used, the gradient will be paint along a horizontal line inside the shape currently drawing.

Pen

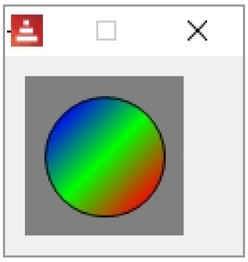
```
Red [needs: view]
view [
      base draw [
            pen linear blue green red 0x0 80x80
            line-width 5
            line 0x0 80x80
      ]
            base draw [
            pen linear blue green 0x0 40x40 pad
            line-width 5
            line 0x0 80x80
```

```
        ]
                base draw [
                pen linear blue green 0x0 40x40 reflect
                line-width 5
                line 0x0 80x80
        ]
]
```



Fill-pen

```
Red [needs: view]
view [
        base draw [
                fill-pen linear blue green red 18x18 62x62
                circle 40x40 30
        ]
]
```



DRAW **radial** - radial color gradient

From Red's official documentation (with eventual minor changes):

Syntax

```
<pen/fill-pen> radial <color1> <offset> ... <colorN> <offset>
<center> <radius> <focal> <spread>

<color1/N> : list of colors for the gradient (tuple! word!).
<offset> : (optional) offset of gradient color (float!).
<center> : (optional) center point (pair!).
<radius> : (optional unless <center>) radius of the circle to paint
along (integer! float!).
<focal> : (optional) focal point (pair!).
<spread> : (optional) spread method (word!).
```

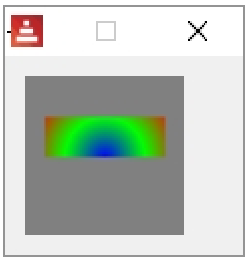
Description

Sets a radial gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same as `repeat` for Windows platform).

The radial gradient will be painted from focal point to the edge of a circle defined by center point and radius. The start color will be painted in focal point and the end color will be painted in the edge of the circle.

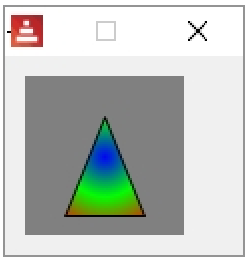
Pen

```
Red [needs: view]
view [
      base draw [
            pen radial blue green red 40x40 40 ; colors center radius
            line-width 20
            line 10x30 70x30
      ]
]
```



Fill-pen

```
Red [needs: view]
view [
      base draw [
            fill-pen radial blue green red 40x40 40 ; colors center
radius
            triangle 20x70 60x70 40x20
      ]
]
```



DRAW **diamond** - diamond color gradient

From Red's official documentation (with eventual minor changes):

Syntax

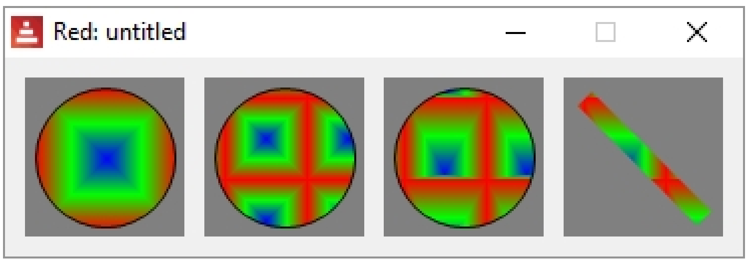
```
<pen/fill-pen> diamond <color1> <offset> ... <colorN> <offset> <upper>
<lower> <focal> <spread>

<color1/N> : list of colors for the gradient (tuple! word!).
<offset> : (optional) offset of gradient color (float!).
<upper> : (optional) upper corner of a rectangle. (pair!).
<lower> : (optional unless <upper>) lower corner of a rectangle
(pair!).
<focal> : (optional) focal point (pair!).
<spread> : (optional) spread method (word!).
```

Description

Sets a diamond-shaped gradient to be used for drawing operations. The following values are accepted for the spread method: `pad`, `repeat`, `reflect` (currently `pad` is same as `repeat` for Windows platform).

The diamond gradient will be painted from focal point to the edge of a rectangle defined by upper and lower. The start color will be painted in focal point and the end color will be painted in the edge of the diamond.

```
Red [needs: view]
view [
      base draw [
            fill-pen diamond  blue green red ; just centers the gradient
            circle 40x40 35
      ]
      base draw [
            fill-pen diamond  blue green red 10x10 50x50 ;added
coordinates of the gradient "box"
            circle 40x40 35
      ]
      base draw [
            fill-pen diamond  blue green red 10x10 50x50 30x48; added a
point of focus
            circle 40x40 35
      ]
      base draw [
            pen diamond  blue green red 10x10 50x50 30x48
            ; a line over the last gradient:
            line-width 10
            line 10x10 70x70
      ]
]
```



DRAW **bitmap** - bitmap fill

From Red's official documentation (with eventual minor changes):

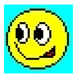
Syntax

```
<pen/fill-pen> bitmap <image> <start> <end> <mode>

<image> : image used for tiling (image!).
<start> : (optional) upper corner for crop section within image
(pair!).
<end> : (optional) lower corner for crop section within image
(pair!).
<mode> : (optional) tile mode (word!).
```

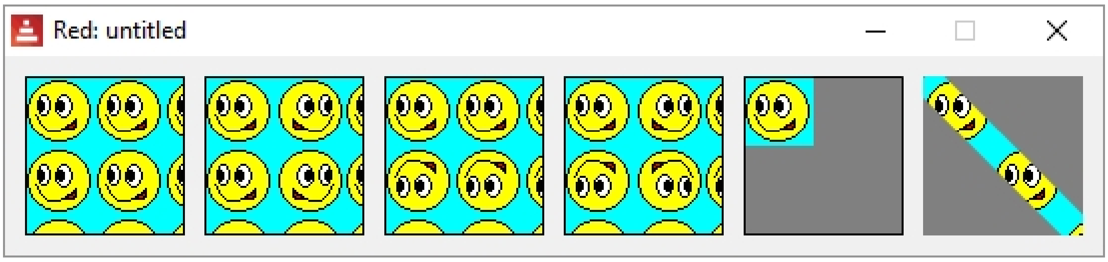
Description

Sets an image as pattern to be used for filling operations. The following values are accepted for the tile mode: `tile` (default), `flip-x`, `flip-y`, `flip-xy`, `clamp`.

Starting default point is 0x0 and ending point is image's size.

The sample bitmap loaded for the following example is: 

```
Red [needs: view]
myimage: load %asprite.bmp  ; bitmap must be loaded first
view [
      base draw [
            fill-pen bitmap myimage tile ; default
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-x
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-y
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage flip-xy
            box 0x0 79x79
      ]
      base draw [
            fill-pen bitmap myimage clamp
            box 0x0 79x79
      ]
            base draw [
            pen bitmap myimage
            line-width 15
            line 0x0 80x80
      ]
]
```



DRAW **pattern** - draw pattern fill

From [Red's official documentation]() (with eventual minor changes):

Syntax

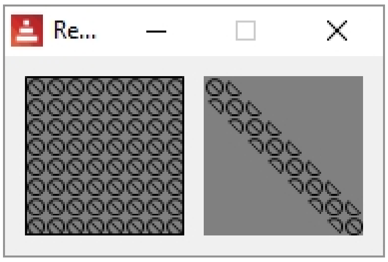
```
<pen-fill-pen> pattern <size> <start> <end> <mode> [<commands>]

<size> : size of the internal image where <commands> will be drawn
(pair!).
<start> : (optional) upper corner for crop section within internal
image (pair!).
<end> : (optional) lower corner for crop section within internal image
(pair!).
<mode> : (optional) tile mode (word!).
<commands> : block of Draw commands to define the pattern.
```

Description

Sets a custom shape as pattern to be used for filling operations. The following values are accepted for the tile mode: `tile` (default), `flip-x`, `flip-y`, `flip-xy`, `clamp`.

Starting default point is 0x0 and ending point is `<size>`.

```
Red [needs: view]
view [
      ; first we draw a filled box:
      base draw [
            fill-pen pattern 10x10 [
                  circle 5x5 4
                  line 3x3 7x7
            ]
            box 0x0 79x79
      ]
      ; then we draw a line:
      base draw [
            pen pattern 10x10 [
                  circle 5x5 4
                  line 3x3 7x7
            ]
            line-width 15
            line 0x0 79x79
      ]
]
```



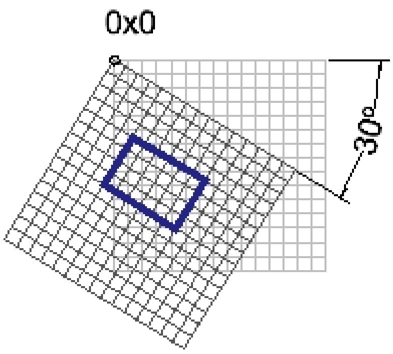
< Previous topic

Next topic >

DRAW - 2D transforms

DRAW rotate

Example of a rotation of 30º centered at 0x0:



From Red's official documentation (with eventual minor changes):

Syntax

```
rotate <angle> <center> [<commands>]
rotate 'pen <angle> rotate 'fill-pen <angle>

<angle> : the angle in degrees (integer! float!).
<center> : (optional) center of rotation (pair!).
<commands> : (optional) Draw dialect commands.
```

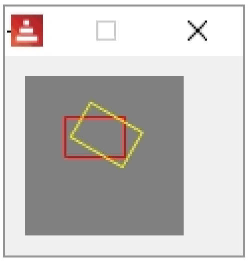
Description

Sets the clockwise rotation about a given point, in degrees. If optional `center` is not supplied, the rotate is about the origin of the current user coordinate system. Negative numbers can be used for counter-clockwise rotation. When a block is provided as last argument, the rotation will be applied only to the commands in that block.

When the `'pen` or `'fill-pen` lit-words are used, the rotation is applied respectively to the current pen or current fill-pen.

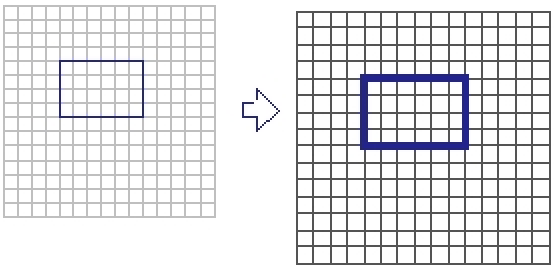
```
Red [needs: view]
view [
      base draw [
          pen red
            box 20x20 50x40 ; horizontal rectangle
            rotate 30 40x40 ; rotate 30 degrees centered at 40x40
            pen yellow
```

```
            box 20x20 50x40 ; same command, different box
            ]
]
```



DRAW scale

Example of a 1.2 scale increase in both x and y axis:



From Red's official documentation (with eventual minor changes):

Syntax

```
scale <scale-x> <scale-y> [<commands>]
scale 'pen <scale-x> <scale-y>
scale 'fill-pen <scale-x> <scale-y>

<scale-x> : the scale amount in X (number!).
<scale-y> : the scale amount in Y (number!).
<commands> : (optional) Draw dialect commands.
```

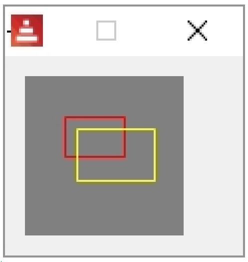
Description

Sets the scale amounts. The values given are multipliers; use values greater than one to increase the scale; use values less than one to decrease it. When a block is provided as last argument, the scaling will be applied only to the commands in that block.

When the `'pen` or `'fill-pen` lit-words are used, the scaling is applied respectively to the current pen or current fill-pen.

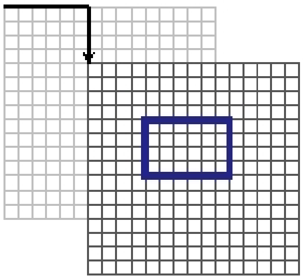
```
Red [needs: view]
view [
      base draw [
          pen red
            box 20x20 50x40 ; horizontal rectangle
            scale 1.3 1.3 ;30% bigger in both x and y
            pen yellow
```

```
                box 20x20 50x40 ; same command, different box
                ]
]
```



DRAW translate

Example of a translation in the x and y axis:



Translates the entire coordinate system.

From Red's official documentation (with eventual minor changes):

Syntax

```
translate <offset> [<commands>]
translate 'pen <offset>
translate 'fill-pen <offset>

<offset> : the translation amounts (pair!).
<commands> : (optional) Draw dialect commands.
```

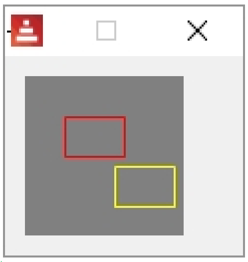
Description

Sets the origin for drawing commands. Multiple translate commands will have a cumulative effect. When a block is provided as last argument, the translation will be applied only to the commands in that block.

When the `'pen` or `'fill-pen` lit-words are used, the translation is applied respectively to the current pen or current fill-pen.

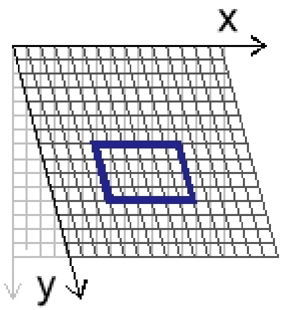
```
Red [needs: view]
view [
        base draw [
            pen red
              box 20x20 50x40 ; horizontal rectangle
              translate 25x25
              pen yellow
```

```
            box 20x20 50x40 ; same command, different rectangle
            ]
]
```



DRAW skew

A skewed coordinate system is when the axis are not orthogonal.



The skew command tilts the x axis and/or the y axis by a given number of degrees.

From Red's official documentation (with eventual minor changes):

Syntax

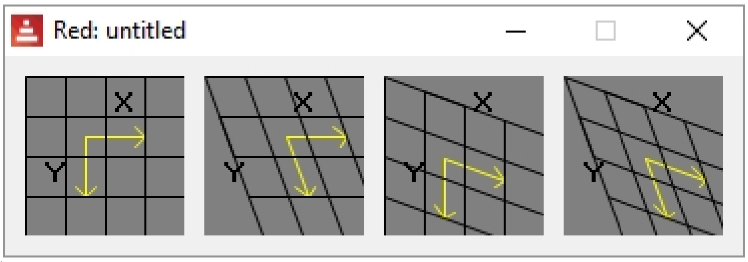
```
skew <skew-x> <skew-y> [<commands>]
skew 'pen <skew-x> <skew-y>
skew 'fill-pen <skew-x> <skew-y>

<skew-x> : skew along the x-axis in degrees (integer! float!).
<skew-y> : (optional) skew along the y-axis in degrees (integer!
float!).
<commands> : (optional) Draw dialect commands.
```

Description

Sets a coordinate system skewed from the original by the given number of degrees. If `<skew-y>` is not provided, it is assumed to be zero. When a block is provided as last argument, the skewing will be applied only to the commands in that block.

When the `'pen` or `'fill-pen` lit-words are used, the skewing is applied respectively to the current pen or current fill-pen.

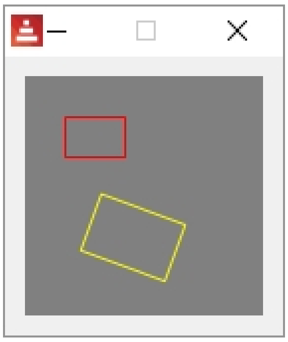
```
Red [needs: view]
view [
      base draw [
            pen yellow        ; Just draw two arrows
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black         ; Just draw a grid
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
      ]

      base draw [
            skew 20 0 ;skew X axis 20 degrees
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X" ;the text does not follow skew!
            text 10x40 "Y"
      ]
      base draw [
            skew 0 20 ; skew Y axis 20 degrees
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
      ]
      base draw [
            skew 20 20 ; skew both axis 20 degrees
            pen yellow
          line 30x30 30x60 25x55
            line 30x60 35x55
            line 30x30 60x30 55x35
            line 60x30 55x25
            pen black
            box 0x0 80x80
            line 0x20 80x20 0x20 0x40 80x40 80x60 0x60
            line 20x0 20x80 20x0 40x0 40x80 60x80 60x0
            text 45x5 "X"
            text 10x40 "Y"
      ]
]
```



DRAW transform

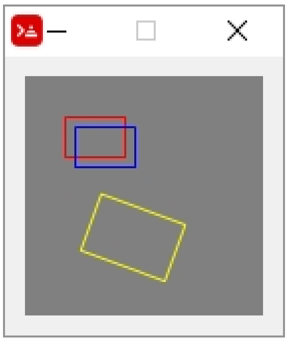
Performs translation, rotation and scaling on a single command. The `transform` below uses 0x0 as anchor point (reference point), rotates 20º, scales to 1.5 in both axis and translates 20 units both in the x and y axis:

```
Red [needs: view]
view [
      base 120x120 draw [
          pen red
            box 20x20 50x40 ; horizontal rectangle
            transform 0x0 20 1.5 1.5 20x20
            pen yellow
            box 20x20 50x40 ; same command, different rectangle
            ]
]
```



If a block is provided as last argument, these transformations are applied only to the commands in that block.

```
Red [needs: view]
view [
      base 120x120 draw [
          pen red
            box 20x20 50x40 ; first rectangle, red
            transform 0x0 20 1.5 1.5 20x20 [
                  pen yellow
                  box 20x20 50x40 ; second rectangle, yellow
            ]
            pen blue
            box 25x25 55x45 ; third rectangle, blue
            ]
]
```



From Red's official documentation (with eventual minor changes):

Syntax

```
transform <center> <angle> <scale-x> <scale-y> <translation>
[<commands>]
transform 'pen <center> <angle> <scale-x> <scale-y> <translation>
transform 'fill-pen <center> <angle> <scale-x> <scale-y>
<translation>

<center> : (optional) center of rotation (pair!).
<angle> : the rotation angle in degrees (integer! float!).
<scale-x> : the scale amount in X (number!).
<scale-y> : the scale amount in Y (number!).
<translation> : the translation amounts (pair!).
<commands> : (optional) Draw dialect commands.
```

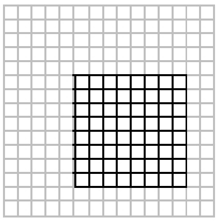
Description

Sets a transformation such as translation, scaling, and rotation. When a block is provided as last argument, the transformation will be applied only to the commands in that block.

When the `'pen` or `'fill-pen` lit-words are used, the transformation is applied respectively to the current pen or current fill-pen.

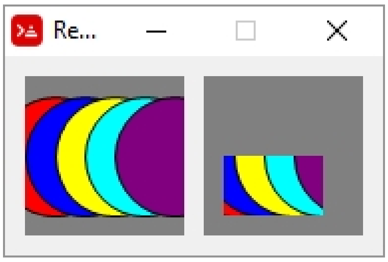
DRAW clip

Limits the drawing area to a rectangle.



```
Red [needs: view]
view [
```

```
        base
        draw [
            pen black
            fill-pen red circle 15x40 30
            fill-pen blue circle 30x40 30
            fill-pen yellow circle 45x40 30
            fill-pen cyan circle 60x40 30
            fill-pen purple circle 75x40 30
        ]
        base
        draw [
            clip 10x40 60x70
            pen black
            fill-pen red circle 15x40 30
            fill-pen blue circle 30x40 30
            fill-pen yellow circle 45x40 30
            fill-pen cyan circle 60x40 30
            fill-pen purple circle 75x40 30
        ]
]
```



If a block is provided as last argument, the clipping is applied only to the commands in that block, i.e. after the block, the whole area becomes canvas again.

From Red's official documentation (with eventual minor changes):

Syntax

```
clip <start> <end> <mode> [<commands>]
clip [<shape>] <mode> [<commands>]

<start> : top-left corner point of clipping area (pair!)
<end> : bottom-right corner point of clipping area (pair!)
<mode> : (optional) merging mode between clipped regions (word!)
<commands> : (optional) Draw dialect commands.
<shape> : Shape dialect commands.
```

Description

Defines a clipping rectangular region defined with two points (start and end) or an arbitrarily shaped region defined by a block of Shape sub-dialect commands. Such clipping applies to all subsequent Draw commands. When a block is provided as last argument, the clipping will be applied only to the commands in that block.

Additionally, the combining mode between a new clipping region and the previous one, can be set to one of the following:

- `replace` (default)

- `intersect`
- `union`
- `xor`
- `exclude`

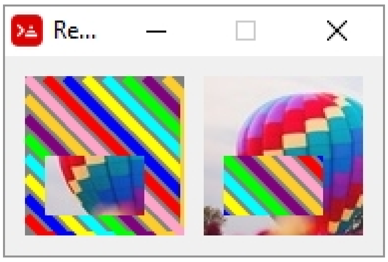
About those modes, I could only figure out `replace` and `exclude`. You may try the others.

```
Red [needs: view]

view [
      base
      draw [
            line-width 5
            pen red line 0x70 10x80 80x80 80x70 10x0
            pen blue line 0x60 20x80 80x80 80x60 20x0
            pen yellow line 0x50 30x80 80x80 80x50 30x0
            pen cyan line 0x40 40x80 80x80 80x40 40x0
            pen green line 0x30 50x80 80x80 80x30 50x0
            pen purple line 0x20 60x80 80x80 80x20 60x0
            pen gold line 0x10 70x80 80x80 80x10 70x0
            pen pink line 0x0 80x80 80x80

            clip 10x40 60x70 replace ;default

            pen red line 0x10 10x0 80x0 80x10 10x80
            pen blue line 0x20 20x0 80x0 80x20 20x80
            pen yellow line 0x30 30x0 80x0 80x30 30x80
            pen cyan line 0x40 40x0 80x0 80x40 40x80
            pen green line 0x50 50x0 80x0 80x50 50x80
            pen purple line 0x60 60x0 80x0 80x60 60x80
            pen gold line 0x70 70x0 80x0 80x70 70x80
            pen pink line 0x80 80x0 80x80

      ]
      base
      draw [
            line-width 5
            pen red line 0x70 10x80 80x80 80x70 10x0
            pen blue line 0x60 20x80 80x80 80x60 20x0
            pen yellow line 0x50 30x80 80x80 80x50 30x0
            pen cyan line 0x40 40x80 80x80 80x40 40x0
            pen green line 0x30 50x80 80x80 80x30 50x0
            pen purple line 0x20 60x80 80x80 80x20 60x0
            pen gold line 0x10 70x80 80x80 80x10 70x0
            pen pink line 0x0 80x80 80x80

            clip 10x40 60x70 exclude

            pen red line 0x10 10x0 80x0 80x10 10x80
            pen blue line 0x20 20x0 80x0 80x20 20x80
            pen yellow line 0x30 30x0 80x0 80x30 30x80
            pen cyan line 0x40 40x0 80x0 80x40 40x80
            pen green line 0x50 50x0 80x0 80x50 50x80
            pen purple line 0x60 60x0 80x0 80x60 60x80
            pen gold line 0x70 70x0 80x0 80x70 70x80
            pen pink line 0x80 80x0 80x80
```

```
        ]
]
```

Or using an image:

```
Red [needs: view]
picture: load %smallballoon.jpeg
view [
        base
        draw [
                line-width 5
                pen red line 0x70 10x80 80x80 80x70 10x0
                pen blue line 0x60 20x80 80x80 80x60 20x0
                pen yellow line 0x50 30x80 80x80 80x50 30x0
                pen cyan line 0x40 40x80 80x80 80x40 40x0
                pen green line 0x30 50x80 80x80 80x30 50x0
                pen purple line 0x20 60x80 80x80 80x20 60x0
                pen gold line 0x10 70x80 80x80 80x10 70x0
                pen pink line 0x0 80x80 80x80

                clip 10x40 60x70 replace ;default

                image picture
        ]
        base
        draw [
                line-width 5
                pen red line 0x70 10x80 80x80 80x70 10x0
                pen blue line 0x60 20x80 80x80 80x60 20x0
                pen yellow line 0x50 30x80 80x80 80x50 30x0
                pen cyan line 0x40 40x80 80x80 80x40 40x0
                pen green line 0x30 50x80 80x80 80x30 50x0
                pen purple line 0x20 60x80 80x80 80x20 60x0
                pen gold line 0x10 70x80 80x80 80x10 70x0
                pen pink line 0x0 80x80 80x80

                clip 10x40 60x70 exclude

                image picture
        ]
]
```



< Previous topic Next topic >

DRAW - Shape sub-dialect

The shape sub-dialect allows you to create shapes (drawings) as blocks.
Some aspects of it remind me of "turtle-graphics". You can move your pen without drawing and coordinates can be absolute (relative to the face) or relative (relative to last position).

Shape sub-dialect also "closes" the shapes for you, allowing you to use `fill-pen` to add colors or patterns.

You may use `fill-pen` , `pen` , `line-width` , `line-join` and `line-cap` as commands in the shape block, but only the last command will be used for the entire shape.

The shape sub-dialect is based on SVG graphics. I found the following links to be helpful in understanding some of the concepts:

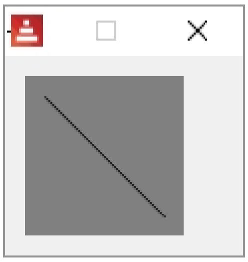
https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Paths

http://www.w3.org/TR/SVG11/paths.html

⊕ line

The most basic example:

```
Red [needs: view]

myshape: [line 10x10 70x70]

view compose/deep/only [
      base draw [
          shape (myshape)
      ]
]
```

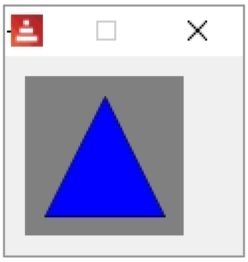


Notice the `compose/deep/only` and the **parentheses** around the shape name. As far as I know, you must use those when working with shapes.

Automatic closing

In the example below, only two lines were actually drawn. I added `fill-pen` to illustrate it better:

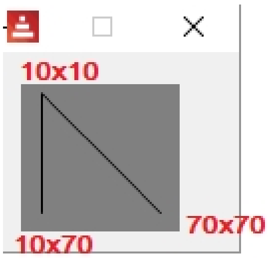
```
Red [needs: view]

myshape: [
        line 10x70 40x10 70x70 ;two lines only
]
view compose/deep/only [base draw [ fill-pen blue shape (myshape)]]
```



⊕ move

The most basic example:

```
Red [needs: view]

myshape: [
        line 10x10 70x70 ;line from 10x10 to 70x70
        move 10x70 ;moves the pen without drawing to 10x70
        line 10x10 ;draws a line from current pen position (10x70) to 10x10
]

view compose/deep/only [base draw [shape (myshape)]]
```

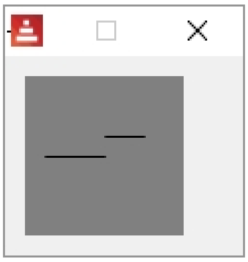


relative positions

Coordinates become relative if you add an apostrophe (') before the command:

```
Red [needs: view]

myshape: [
        line 10x40 40x40 ;horizontal line to the middle
        'move 0x-10 ;new current position RELATIVE to old (up from the middle)
```

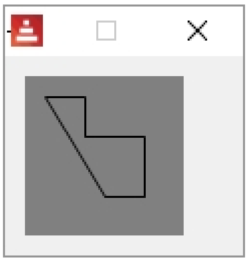
```
      'line 20x0 ;draws a little horizontal line RELATIVE TO current
position
]

view compose/deep/only [base draw [shape (myshape)]]
```



⊕ hline and ⊕ vline

Draws a horizontal or a vertical line from current pen position.

```
Red [needs: view]

myshape: [
      move 10x10 ; puts pen at 10x10
      hline 30 ;horizontal line to coordinate X =30
      vline 30 ;vertical line to coordinate Y = 30
      'hline 30 ;horizontal line 30 pixels long (longer than hline
above)
      'vline 30 ;vertical line 30 pixels long
      'hline -20 ; just to show the use of RELATIVE negative lenghts
      ; shape dialect will close the shape now
]

view compose/deep/only [base draw [shape (myshape)]]
```



⊕ arc

From Red's official documentation (with eventual minor changes):

Syntax

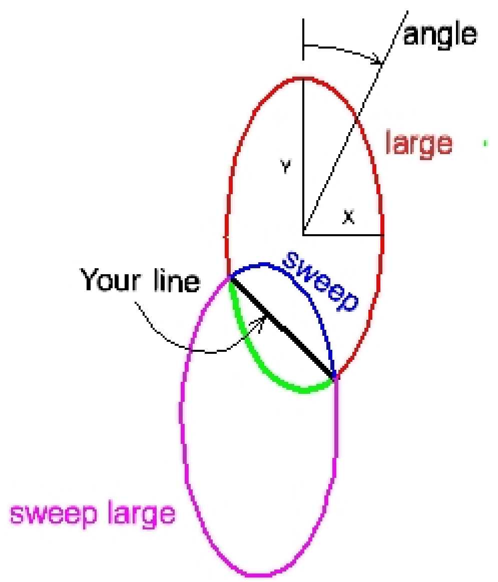
```
arc <end> <radius-x> <radius-y> <angle> sweep large (absolute)
'arc <end> <radius-x> <radius-y> <angle> sweep large (relative)
```

```
<end> : arc's end point (pair!).
<radius-x> : radius of the circle along x axis (integer! float!).
<radius-y> : radius of the circle along y axis (integer! float!).
<angle> : angle between the starting and ending points of the arc
in degrees (integer! float!).
sweep : (optional) draw the arc in the positive angle direction.
large : (optional) produces an inflated arc (goes with 'sweep
option).
```

Description

Draws the arc of a circle between the current pen position and the end point, using radius values. The arc is defined by one angle value.

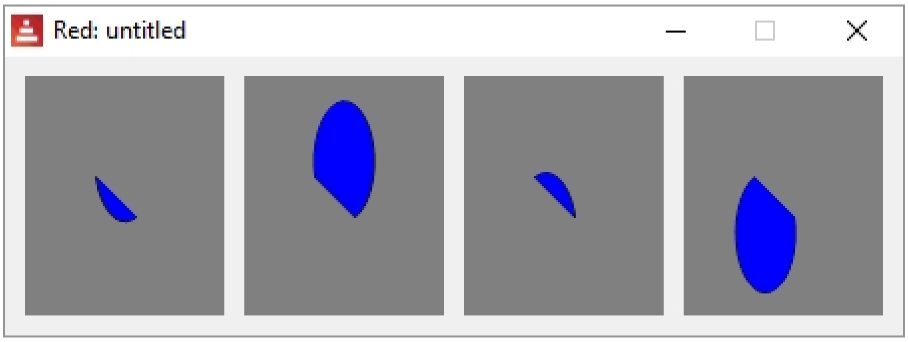
Here is an explanation about how arc works. Since you define your line (two points) and your ellipse (x-radius, y-radius and angle), there are only two positions for the ellipse that make your line a chord to it. The options `sweep`, `large` and `sweep large` define which arc of these ellipses will show in your drawing. Notice that in the illustration below, the angle of the ellipse is zero.



In the `arc` definition you only inform the arc's end position. That is because the start position is the current pen position. So, if `arc` is your first command in a shape, you must first `move` to the position you want to start at.

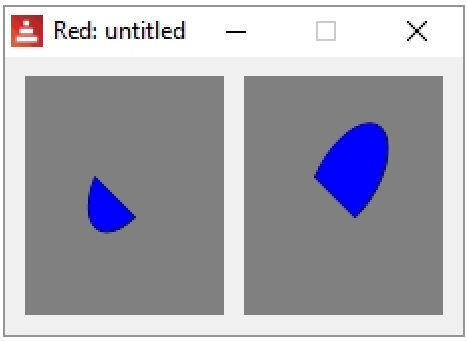
```
Red [needs: view]

myshape_1: [
    move 35x50
    arc  55x70 15 30 0
]
myshape_2: [
    move 35x50
    arc  55x70 15 30 0 large sweep
]
myshape_3: [
    move 35x50
    arc  55x70 15 30 0 sweep
```

```
]
myshape_4: [
    move 35x50
    arc  55x70 15 30 0 large
]
view compose/deep/only [
      base 100x120 draw [fill-pen blue shape (myshape_1)]
      base 100x120 draw [fill-pen blue shape (myshape_2)]
      base 100x120 draw [fill-pen blue shape (myshape_3)]
      base 100x120 draw [fill-pen blue shape (myshape_4)]
]
```



With an angle:

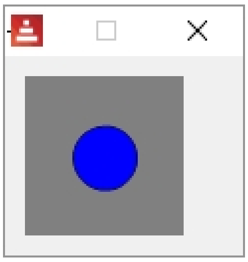
```
Red [needs: view]

myshape_1: [
    move 35x50
    arc  55x70 15 30 30
]
myshape_2: [
    move 35x50
    arc  55x70 15 30 30 large sweep
]

view compose/deep/only [
      base 100x120 draw [fill-pen blue shape (myshape_1)]
      base 100x120 draw [fill-pen blue shape (myshape_2)]
]
```



A circle:

```
Red [needs: view]
```

```
myshape_1: [
    move 56x40
    arc  56x41 16 16 0 large
]
view compose/deep/only [base  draw [fill-pen blue shape (myshape_1)]]
```



⊕ qcurve

From Red's official documentation (with eventual minor changes):

Syntax

```
qcurve <point> <point> ... (absolute)
'qcurve <point> <point> ... (relative)

<point> : coordinates of a point (pair!).
```

Description

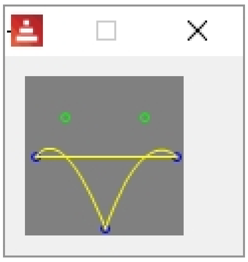
Draws a quadratic Bezier curve from a sequence of points, starting from the current pen position. At least 2 points are required to produce a curve (the first point is the implicit starting point).

Draw a quadratic Bezier curve from a sequence of 3 points. The following script draws two qcurves using <start-point> <control-point > <end-point/start-point> <control-point > <end-point>. Allows absolute or relative (for relative, use `'qcurve`) coordinates.

```
Red [needs: view]
myshape: [
    move 5x40
    qcurve  20x20 40x76 60x20 76x40
]
view compose/deep/only [
      base  draw [
            pen blue
            circle 5x40 2 ;shows start point 1
            circle 40x76 2 ;shows endpoint 1/start point 2
            circle 76x40 2 ;shows endpoint 2
            pen green
            circle 20x20 2 ;shows control point 1
            circle 60x20 2 ;shows control point 2
            pen yellow
            shape (myshape)
            ]
```

```
]
```

I added the approximate location of the fixed-points (blue) and the control-points (green) in the image bellow. They are not generated by the program, I edited the image.



⊕ curve

From Red's official documentation (with eventual minor changes):

Syntax

```
curve <point> <point> <point> ... (absolute)
'curve <point> <point> <point> ... (relative)

<point> : coordinates of a point (pair!).
```

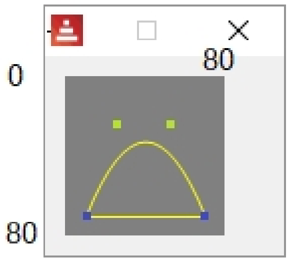
Description

Draws a cubic Bezier curve from a sequence of points, starting from the current pen position. At least 3 points are required to produce a curve (the first point is the implicit starting point).

Draws a cubic Bezier curve using <start-point (current pen position)> <control-point1 (argument)> <control-point2 (argument)> <end-point (argument)> . Allows absolute or relative (for relative, use `'curve`) coordinates.

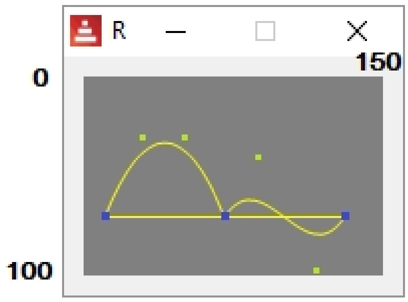
```
Red [needs: view]

myshape_1: [
      move 10x70 ; start-point
      curve 30x20 50x20 70x70 ; control-point  control-point end-point
]
view compose/deep/only [base draw [ pen yellow shape (myshape_1)]]
```

I added the approximate location of the fixed-points (blue) and the control-points (green) in the images bellow. They are not generated by the program, I edited them.



You may add more points to the `curve` command, they will create a new independent curve:

```
Red [needs: view]
myshape_1: [
      move 10x70 ; start-point
      curve 30x20 ;first control point
            50x20 ;second control point
            70x70 ;end-point first curve/ new start-point second curve
            90x40 ;first control point second curve
            110x100 ;second control point second curve
            130x70 ;end-point second curve
]
view compose/deep/only [base 150x100 draw [ pen yellow
shape(myshape_1)]]
```



⊕ qcurv

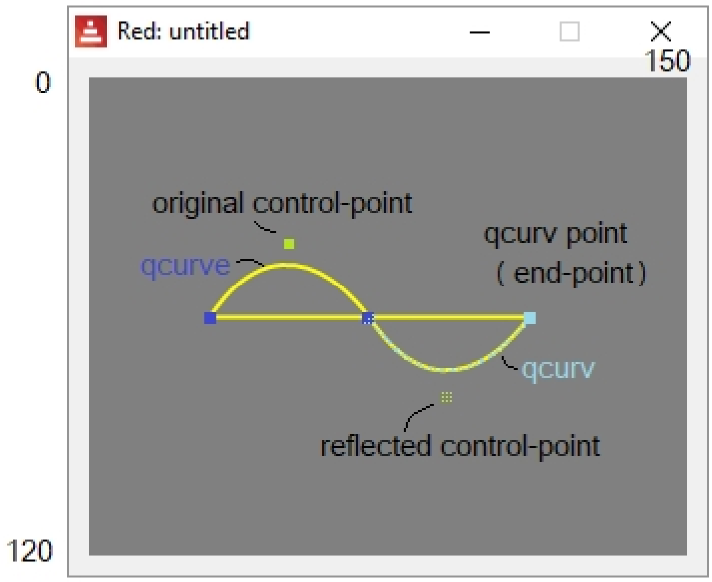
Syntax

```
qcurv <point> (absolute)
'qcurv <point> (relative)

<point> : coordinates of the ending point (pair!).
```

`qcurv` draws a smooth quadratic Bezier from the current pen position to the specified point.
You don't have to provide the control-point between start-point and end-point, `qcurv` creates this control-points as a reflection of the last control point given in the shape block, so, you must have a command that uses a control-point before using `qcurv`.

```
Red [needs: view]
```

```
myshape_1: [
      move 30x60 ;start-point of qcurve
      qcurve 50x30 70x60 ;control-point  end-point
      qcurv 110x60 ; end-point of qcurv
]
view compose/deep/only [
      base 300x240 draw [
            scale 2 2 ; just to make it bigger
            pen yellow
            shape (myshape_1)
            ]
]
```



As of april 2018, `qcurv` only works with one endpoint as argument.

⊕ curv

Draws a smooth cubic Bezier curve from a sequence of points.

Just like `qcurv`, `curv` creates control-points reflected relative to the last control-point in the shape block. But since cubic Beziers require 2 control-points, you must provide the second for each segment. This second control-point will be reflected as the first control-point of the next segment.

From Red's official documentation (with eventual minor changes):

Syntax

```
curv <point> <point> ... (absolute)
'curv <point> <point> ... (relative)
```

```
<point> : coordinates of a point (pair!).
```

Description

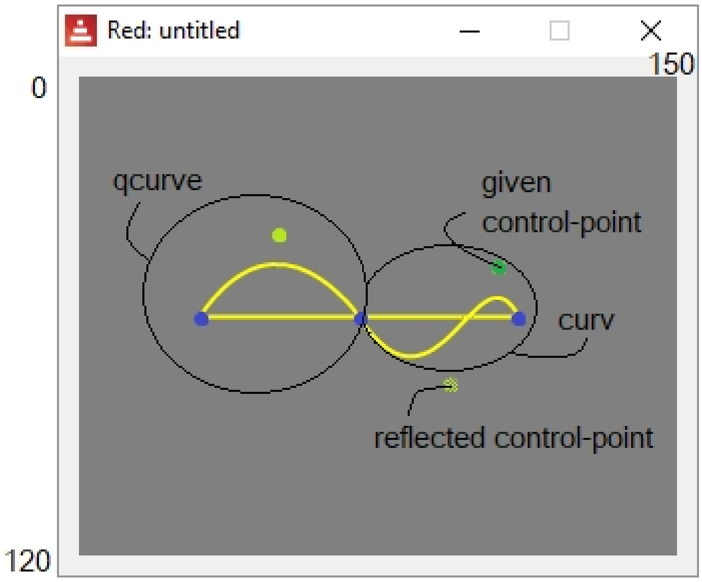
Draws a smooth cubic Bezier curve from a sequence of points, starting from the current pen position. At least 2 points are required to produce a curve (the first point is the implicit starting point).

"The first control point is assumed to be the reflection of the second control point on the previous command relative to the current point. (If there is no previous curve command, the first control point is the current point.)"

So, `curv` draws a cubic Bezier using <current pen position start-point ><automatically created reflected control-point1><control-point2> <end-point>.

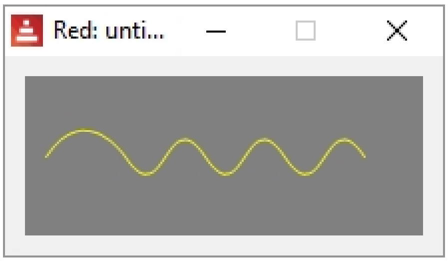
So, the arguments you actually pass to `curv` are only: <control-point2> <end-point>[...]

```
Red [needs: view]

myshape_1: [
      move 30x60 ;start-point of qcurve
      qcurve 50x30 70x60 ;control-point  end-point
      curv 100x40 110x60 ; curv's second control-point and end-point
]
view compose/deep/only [
      base 300x240 draw [
            scale 2 2 ; just to make it bigger
            pen yellow
            shape (myshape_1)
            ]
]
```



`curv` may use many consecutive control-points and points:

```
Red [needs: view]

      ;second control-point    point

myshape_1: [
      move 10x40
      qcurve 30x10 50x40
      curv  70x10 90x40 110x10 130x40 150x10 170x40
      move 10x40
]
view compose/deep/only [base 200x80 draw [ pen yellow shape (myshape_1)]
]
```



< Previous topic

Next topic >

DRAW - Programmatic drawing and Animation

Executing drawings using Red programming tools (loops, math, branching etc.) requires some structuring of the script. I found the following to be a rule-of-thumb structure:

```
Red [needs: view]
draw-changing: function [  ]
view compose/deep/only [
      face focus
      draw[commands (arguments)]
      on-event [ draw-changing ]
]
```

draw-changing - This are the functions to be called from an event to do calculations and then change the "draw" field of the face's object. You must change this field from here because you can't change it from inside the dialect block.

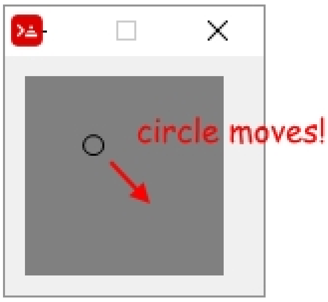
face focus - Some events (as `key`) seem to only be generated with `focus` on graphic faces like base or box, so beware.

draw - Executes the draw dialect. Any calculated argument (variable) should be within parenthesis to be evaluated by `compose/deep/only`.

on-event - Calls the appropriate draw-changing function considering the type of event.

Very simple animation:

```
Red [needs 'view]

position: 0x0

update-canvas: func [] [
    position: position + 1x1
    canvas/draw:  reduce ['circle position 5]
]

view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```



The code explained:

```
Red [needs 'view]

{ "position" is the center of the circle
that will be moved. Here it's at the top left corner}

position: 0x0

{ the "update-canvas" function does all the
necessary processing and "passes" the draw
routine to the draw field of the canvas
object.  Notice three things in the code below:
1- Yes, draw is a field of an object!
2- You must use "reduce" to send the
current value of position;
3- There must be an apostrophe before
"circle". "circle" is a keyword of the draw
dialect, and so it must be passed "as is"}

update-canvas: func [] [
    position: position + 1x1
    canvas/draw:  reduce ['circle position 5]
]

{The view routine creates a base named
"canvas" that updates itself 25 times
per second}

view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```

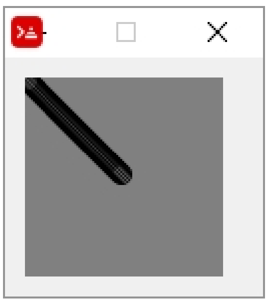
To show that canvas is an object!, close the graphic view after it runs a bit, but leave the console open. Type `? canvas` in the console. You will get:

```
>> ? canvas
CANVAS is an object! with the following words and values:
     type              word!          base
     offset            pair!          10x10
     size              pair!          100x100
     text              none!          none
     image             none!          none
     color             tuple!         128.128.128
     menu              none!          none
```

```
      data             none!          none
      enabled?         logic!         true
      visible?         logic!         true
      selected         none!          none
      flags            none!          none
      options          block!         length: 6  [style: base vid-align: 
top at-o...
      parent           object!        [type offset size text image color 
menu dat...
      pane             none!          none
      state            none!          none
      rate             integer!       25
      edge             none!          none
      para             none!          none
      font             none!          none
      actors           object!        [on-time]
      extra            none!          none
      draw             block!         length: 3  [circle 37x37 5]
      on-change*       function!      [word old new /local srs same-pane? 
f saved]
      on-deep-change*  function!      [owner word target action new index 
part]
```

In the next example, instead of changing the `draw` block, we will `append` it with new `draw` commands. The result is that all the previous drawings are kept, and not deleted (in fact they are redrawn, but...), creating a trail of drawings:

```
Red [ needs 'view ]
position: 0x0
command: [] ; initialized to prevent error.

update-canvas: func [] [
    position: position + 1x1
      {I could not figure out how to append the draw
      method directly, so a block named "command" was
      used to pass words to draw dialect}
    append command reduce ['circle position 5]
    canvas/draw:  command
]

view [
    canvas: base 100x100  rate 25
    on-time [update-canvas]
]
```

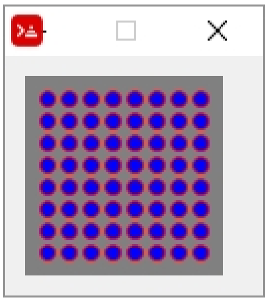


Note that if you close the graphic window and type `? canvas` in the console you will see a long block as the value of draw:

```
>> ? canvas
...
    draw    block!   length: 84  [circle 1x1 5 circle 2x2 5 circle 3x3
5 circle 4x4 5 ...
...
```

An example of programmed drawing:

```
Red [needs: view]

drawcircles: does [
    command: [pen red fill-pen blue]
    repeat x 8 [
        repeat y 8 [
            position:(x * 11x0) + (y * 0x11)
            append command reduce ['circle position 4]
        ]
    ]
    canvas/draw:  command
]

view [
    canvas: base 100x100
    do [drawcircles]
]
```



You could have written the program above without using a function, but you would need the `no-wait` refinement for `view`, like this:

```
Red [needs: view]

command: [pen red fill-pen blue]

view/no-wait [
    canvas: base 100x100
]
{the "no-wait" refinement above allows the
script do create the view (base) and then keep
going, to the nested "repeats" below.
Without "no-wait" the script would stay in the
"view" block}
```

```
repeat x 8 [
    repeat y 8 [
        position:(x * 11x0) + (y * 0x11)
        append command reduce ['circle position 4]
    ]
]

canvas/draw:  command
probe command {just to show you what was sent to draw.
you must use probe instead of print, because print
tries to evaluate things, and "pen" and "circle" have
no value}
```

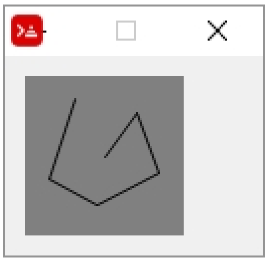
```
[pen red fill-pen blue circle 11x11 4 circle 11x22 4 circle 11x33 4
circle 11x44 4 circle 11x55 4 circle 11x66 4 circle 11x77 4 circle 11x88
4 circle 22x11 4 circle 22x22 4 circle 22x33 4 circle 22x44 4 circle
22x55 4 circle 22x66 4 circle ...
```

You see that Red updates the base with the drawings generated by the `draw` block even after the face was created by View. That happens because in Red, unlike Rebol, the default is that whenever you change some field of the face object, the face is updated automatically. That wouldn't have happened if you added the statement `system/view/auto-sync?: off` at the beginning of the script as described here .

The simplest Paint program ever:

```
Red [needs: view]
newposition: 40x40            ;sorry, but always starts in center.
linedraw: func [offset] [     ;func, not function. Variables are global.
    oldposition: newposition
    newposition: offset
      ; now we keep adding lines to the draw block:
    append canvas/draw reduce['line oldposition newposition]
]

view [
canvas: base draw[]           ;creates a draw field in the object.
    on-down [                 ;when button is clicked...
        do [linedraw event/offset]  ;sends mouse position.
    ]
]
```

Every time you click the mouse on the base, a new line segment is drawn:



Here is a much improved version of the script that, however, does not use the "rule-of-thumb" structure:

```
Red [needs: view]
EnableWrite: false
view [
    canvas: base 150x150 white all-over
    draw[]
    on-down [                       ;when mouse button is pressed...
        EnableWrite: true           ;... enables drawing...
        startpoint: event/offset    ;...and get cursor position
    ]
    on-up [EnableWrite: false]      ;when mouse button is released,
disable drawing
    on-over [                       ;when cursor is moved on canvas...
        if EnableWrite [
            endpoint: event/offset  ;get cursor position
                  ; now we keep adding lines to the draw block
            append canvas/draw reduce['line startpoint endpoint]
            startpoint: endpoint
        ]
    ]
]
```

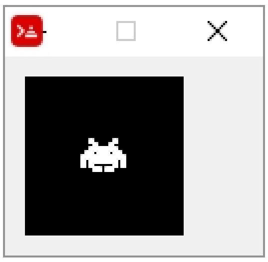
Note that the `all-over` flag allows the `over` event to create events for every mouse movement, as explained here.

Moving a shape with arrow keys

This script draws an "alien" in the center of a `base`, and allows the arrow keys to move the shape up, down, left and right. It uses the `translate` transform to do the moving. Note the use of **`compose`** to evaluate things in parenthesis.

```
Red [needs: view]
pos: 28x31        ; This is the initial position of the "alien"

{ The following block is just the shape of an "alien"}
alien: [line 4x0 4x2
 'hline 2 'vline 2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -2 'vline 6
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline 2 'hline 4 'vline -2
 'hline -4 'vline -2 'hline 10 'vline 2
```

```
 'hline -4 'vline 2 'hline 4 'vline -2
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline -6 'hline -2  'vline -2
 'hline -2 'vline -2 'hline -2 'vline -2
 'hline 2 'vline -2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -6 'vline -2
 'hline -2 'vline -2 'hline -2
 move 6x6 'hline 2 'vline 2 'hline -2 'vline -2
 move 14x6 'hline 2 'vline 2 'hline -2 'vline -2
]

{Next function updates the 'draw' block of the cosmos object.
It removes the word 'translate and the following pair!
from the beginning of the block and then inserts the
word 'translate again and the updated position's pair!}
update-cosmos: func [] [
      remove/part cosmos/draw 2
      insert cosmos/draw reduce ['translate pos]
]

view compose/deep/only [
      cosmos: base black focus ; use focus to get on-key event
      draw [
            translate (pos)     ; initial translation. Centers "alien"
            pen white
            fill-pen white
          shape (alien)
      ]

      on-key  [
            switch event/key [
                  up     [pos: pos - 0x1]  ; decreases y axis
                  down   [pos: pos + 0x1]  ; increases y axis
                  left   [pos: pos - 1x0]  ; decreases x axis
                  right  [pos: pos + 1x0]  ; increases x axis
            ]
            update-cosmos
      ]
]
```



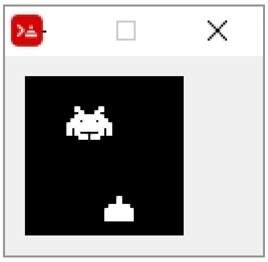
I suggest you try to change the code to `rotate` it.

Moving two or more shapes separately

The following script uses the left and right arrow to move the "spaceship" and "z" and "x" keys to move the "alien". Note the scope of `reduce` and `compose`:

```
Red [needs: view]
;======= initial positions: ========
```

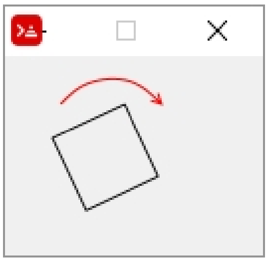
```
alienposition: 28x15
shipposition: 32x60

;======= just the shapes ===========
alien: [line 4x0 4x2
 'hline 2 'vline 2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -2 'vline 6
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline 2 'hline 4 'vline -2
 'hline -4 'vline -2 'hline 10 'vline 2
 'hline -4 'vline 2 'hline 4 'vline -2
 'hline 2 'vline -4 'hline 2 'vline 4
 'hline 2 'vline -6 'hline -2  'vline -2
 'hline -2 'vline -2 'hline -2 'vline -2
 'hline 2 'vline -2 'hline -2 'vline 2
 'hline -2 'vline 2 'hline -6 'vline -2
 'hline -2 'vline -2 'hline -2
 move 6x6 'hline 2 'vline 2 'hline -2 'vline -2
 move 14x6 'hline 2 'vline 2 'hline -2 'vline -2
]
spaceship: [move 0x12 'hline 14 'vline -6
'hline -2 'vline -2 'hline -4 'vline -4 'hline -2
'vline 4  'hline -4 'vline 2 'hline -2 'vline 6
]
;======= The draw block updating function ======
; this time we create the whole block and just replace
; the original cosmos/draw
update-cosmos: does[
    drawblock: reduce compose/deep[
        'pen white
        'fill-pen white
        'translate alienposition [shape [(alien)]]
        'translate shipposition [shape [(spaceship)]]
    ]
    ;probe drawblock ;uncomment if you want to see it
    cosmos/draw: drawblock
]

view compose/deep/only [
    cosmos: base black focus
;this "draw" be "executed" only once:
    draw [
        pen white
        fill-pen white
        translate (alienposition) [shape (alien)]
        translate (shipposition) [shape (spaceship)]
    ]
; now the draw block will be recreated on every key press
    on-key  [
            switch event/key [
                  #"z"  [alienposition: alienposition - 1x0]      ;
decreases x axis
                  #"x"  [alienposition: alienposition + 1x0]      ;
increases x axis
                  left [shipposition: shipposition - 1x0]       ;
decreases x axis
                  right [shipposition: shipposition + 1x0]  ; increases
x axis
        ]
       update-cosmos ; calls the "draw block recreating function"
      ]
]
```



Curiouser and curiouser...

The following script creates a rotating square using a different, somewhat strange technique:

```
Red [needs: view]
tick: 1
view[
    mybox: box rate 10 draw [
        mytransform: rotate  1  40x40
        box 20x20 60x60
        ]

    on-time [
        tick: tick + 1
        mytransform/2: tick
    ]
]
```



In this script, `mytransform/2` refers to the second element of `mytransform` (`1`). 1 is the starting value, but is increased on every `on-time` event. Since this second element is an argument of the `rotate` transform, on every `on-time` event the rotation increases!
A side note is that the first `box` is a face of **View** dialect, while the second `box` is a command of the **Draw** dialect that creates a rectangle.

< Previous topic Next topic >

What is in "system"

If you type `? system` on the console, you get:

```
>> ? system
SYSTEM is an object! with the following words and values:
     version      tuple!         0.6.3
     build        object!        [date git config]
     words        object!        [datatype! unset! none! logic!...
     platform     function!      Return a word identifying the operating
system.
     catalog      object!        [datatypes actions natives accessors
errors]
     state        object!        [interpreted? last-error trace]
     modules      block!         length: 0  []
     codecs       block!         length: 8  [png make object! [title:...
     schemes      object!        []
     ports        object!        []
     locale       object!        [language language* locale locale* months
days]
     options      object!        [boot home path script cache thru-cache
...
     script       object!        [title header parent path args]
     standard     object!        [header error file-info]
     lexer        object!        [pre-load throw-error make-hm make-msf...
     console      object!        [prompt result history size running?
catch? ...
     view         object!        [screens event-port metrics fonts
platform ...
     reactivity   object!        [relations stack queue eat-events? debug?
...
```

You may explore these paths using either `?` or `probe`.

Interesting things you can do:

Accessing words, not only system's but your own.

If you type `? system/words`, you get a very, very long list of all words you have in your Red session:

```
>> ? system/words
```

```
SYSTEM/WORDS is an object! with the following words and values:
     datatype!                 datatype!     datatype!
     unset!                    datatype!     unset!
     none!                     datatype!     none!
   ...
   ...
     right-command             unset!
     caps-lock                 unset!
     num-lock                  unset!
```

Type a new word like `banana` on your console, press enter (you get an error) then type `? system/words` again. You will see that `banana` was added to your session's list of words:

```
>> banana
*** Script Error: banana has no value
*** Where: catch
*** Stack:

>> ? system/words
SYSTEM/WORDS is an object! with the following words and values:
     datatype!                 datatype!     datatype!
     unset!                    datatype!     unset!
   ...
   ...
     caps-lock                 unset!
     num-lock                  unset!
     banana                    unset!
```

If you assign a value to banana and repeat `? system/words` you will see that the value is now linked to the word:

```
>> banana: "hello"
   ...
   ...
     caps-lock                 unset!
     num-lock                  unset!
     banana                    string!       "Hello"
```

Changing console's prompt:

```
>> ? system/console/prompt
SYSTEM/CONSOLE/PROMPT is a string! value: ">> "

>> system/console/prompt: "@*=> "
== "@*=> "
@*=>                   ;this is the prompt now
```

Seeing command history:

```
>> probe system/console/history
["probe system/console/history" "?
system/console" {system/console/prompt: "@*=> "} "
" {system/console/prompt: "@*"} "? system/console/prompt" "?
console/prompt" "? system" "? system/standard/error" "? system" "probe
last system/word" "probe last system" "probe last a" "a: [a b c]" "probe
last sys ...
```

Changing error messages:

```
>> ? system/catalog/errors/script
SYSTEM/CATALOG/ERRORS/SCRIPT is an object! with the following words and
values:
     code                 integer!      300
     type                 string!       "Script Error"
     no-value             block!        length: 2  [:arg1 "has no value"]
    ...
     lib-invalid-arg      block!        length: 2  ["LIBRED - invalid
argument for" :arg1]

>> system/catalog/errors/script/type: "Don't be silly!! "
== "Don't be silly!! "

>> nono
*** Don't be silly!! : nono has no value
*** Where: catch
*** Stack:
```

Choose procedures according to OS:

```
>> either system/platform = 'Windows [print "Do this"] [print "Do that"]
Do this
```

Notice the apostrophe before "Windows". This is a `word!` not a `string!`

Get the size of screen:

```
>> print system/view/screens/1/size
1366x768
```

Debug View:

Use `system/view/debug?: yes` , as explained in the GUI Advanced topics chapter.

< Previous topic Next topic >

Appendix I - While we wait for serial port...

(temporary chapter)

Warning 1: This information is mostly for Windows' users;

Warning 2: Serial communication can be tricky, with hidden characters and configuration details. If you are not familiar with it, I suggest you start with a more friendly tutorial.

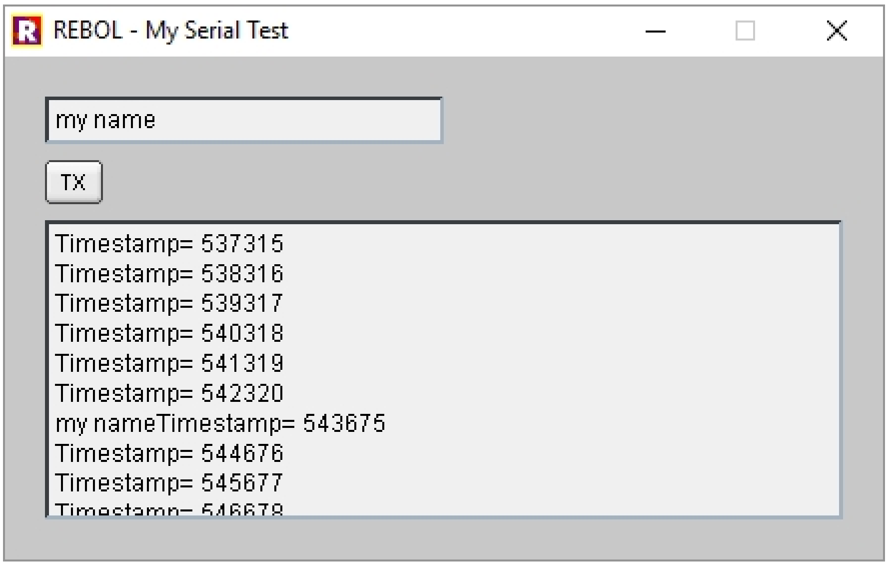
Red does not yet (october 2018) support serial port access. This may be disappointing if you plan to use Red with Arduino, IoT, ESP8266 and hardware in general. So, while we wait for serial port support, I list here a few tricks and tips I have found useful. They are mostly related to sending commands to Windows' cmd using `call`, but Linux users may also find interesting information here.

How Rebol does it. Probably Red will be the same:

Look at Rebol's documentation;

It seems to me that in Rebol you have to tell what your COM port is, create a "serial thing" (named "ser" in the example below) and configure it. Then, to send messages to serial, you `insert` your messages in this "thing", and to read what is received, you `copy` , `pick` or `first` this "thing".

```
Rebol []

System/ports/serial: [ com7 ]
ser: open/direct/no-wait serial://port1/9600/none/8/1
ser/rts-cts: false

view/title layout [
      f: field 200
      btn "TX" [insert ser f/text update ser]
      t: area
      rate 20 feel[engage: [append t/text copy ser show t]]
] "My Serial Test"
```



In this example, what is sent by the device is shown in the `area`, and when you press TX, whatever you wrote in the `field` will be sent to the device.

I tested it with an ESP8266 program that sends a timestamp every second, but also transmits back whatever it receives. The sketch also sends a ctrl-z (0x1A) every 10 messages. In case you are interested, here is the Arduino sketch:

```
long interval = 1000;     //milliseconds between sending timestamps
long previousMillis = 0;
long count = 0;
void setup(){
  Serial.begin(9600);
}

void loop()
{ // this first part "echoes" whatever is sent
  // when characters arrive over the serial port...
  if (Serial.available()) {
    // ...wait a second and send them back.
    delay(1000);
    while (Serial.available() > 0) {
    Serial.write(Serial.read());
    }
  }

  // this second part sends a timestamp every interval
  long currentMillis = millis();
  if(currentMillis - previousMillis > interval) {
      if (count > 10){
        count =  0;
        Serial.print("stop\x1A"); // string "stop" & ctrl-z
      }
    previousMillis = currentMillis;
    Serial.print("Timestamp= ");
    Serial.println(currentMillis);
    count = count +1;
  }
}
```

And now for tips and tricks to use Red as it is...

A function to get the COM ports available:

Sends the command `mode` to cmd and parses (not using `parse`) the returned value:

```
Red []
funcGetComPorts:
; Uses Windows' cmd  to obtain the COM ports available
      has[cmdOutput com-ports b c i] [

      cmdOutput: ""     ;this will hold the output from cmd as text
      com-ports: []   ;this series will contain the COM ports
; now we send  the command "mode" to Windows system (cmd)
; we store the system's return in "cmdOutput"
      call/output "mode" cmdOutput
; now we remove all "-", otherwise they are "glued" to COM text
      trim/with cmdOutput "-"
; now we split cmdOutput into a series
      cmdOutput: split cmdOutput " "
; now we do some editing to get the ports in a nice format
      foreach i cmdOutput [
            b: copy/part i 3
            if b = "COM" [
                  c: copy/part i 4
                  append com-ports c
            ]
      ]
      return com-ports
]

probe funcGetComPorts
```

```
["COM7" "COM3"]
```

Configuring a serial port:

The complete cmd's command to configure a COM port would be:

```
mode  COM7  BAUD=9600  PARITY=n  DATA=8
```

So this would be a COM port configuring function:

```
Red []
SerialConfig: function [COMport baud parity datasize][
      command: ""
      command: rejoin [command "mode " COMport " BAUD=" baud
                        " PARITY=" parity " DATA=" datasize]
      print command
      call/shell form command
]
SerialConfig "COM7" "9600" "n" "8"
```

You can check if it works by typing `mode` on cmd before and after you run the script above. `mode` shows the current configuration of your ports.

Using ComPrinter.exe and SerialSend.exe :

These small executables (available for download here) may be accessed using a `call` command inside a Red script to send and receive data from a serial port. They are open source programs by Ted Burke (thanks!). These are great little programs that, with some creativity, may allow Red to do a lot!

The Red scripts examples here assume these executables are in the same folder as the script. Just paste a copy of them (the executables) there.

ComPrinter *

*look for the updated version you will find in the comments (bottom) of its page (direct download link).

From webpage: "ComPrinter is a console application (i.e. a command line program) that opens a serial port and displays incoming text characters in the console. It features several very useful options."

Options for ComPrinter.exe:

/devnum - Use this to specify a COM port. The default is the highest available com port, including ports >= 10. For example, to set COM7 use `/devnum 7`

/baudrate - Use this to specify the baud rate. Default is 2400 bits per second. For example, to set baud rate to 9600, use `/baudrate 9600`

/keystrokes - Use this to simulate a keystroke for each incoming character, for example to type text into an application.

/debug - Use this to display additional information when opening the COM port.

/quiet - Use this to supress the welcome message text and other information. Only text received via the COM port will be displayed.

The following options are only available in the updated version:

/charcount - Exit after a certain number of characters. For example, to exit after 6 characters, use `/charcount 6`

/timeout - Exit after a timeout – i.e. no data received for the specified number of milliseconds. For example, to exit after 2 seconds of no data, use `/timeout 2000`

/endchar - Exit when a certain character is received. For example, to exit when the letter 'x' is received, use `/endchar x`

/endhex - Exit when a certain hex byte is received. For example, to exit when the hex value 0xFF is received, use `/endhex FF`

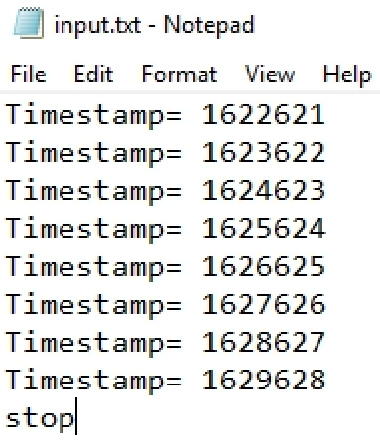
Example:

The example below sends what it receives in COM7 at baud 9600 to file "input.txt" until it receives a ctrl-z. It creates the file automatically or appends an existing file. The Arduino sketch above sends a ctrl-z every now and then, so your output may be longer or shorter:

```
Red[]
```

```
call/output form "ComPrinter.exe /devnum 7 /baudrate 9600 /endhex 1A"
%"input.txt"
;     ComPrinter.exe     - the executable called
;     /devnum 7          - selects COM7
;     /baudrate 9600     - selects baud rate 9600
;     /endhex 1A         - stops ComPrinter when receives a ctrl-z (0x1A)
;     %"input.txt"       - the output file (remember the refinement of
call?)
```

Content of input.txt file after running the script:


input.txt - Notepad

File Edit Format View Help

Timestamp= 1622621
Timestamp= 1623622
Timestamp= 1624623
Timestamp= 1625624
Timestamp= 1626625
Timestamp= 1627626
Timestamp= 1628627
Timestamp= 1629628
stop

In case you want your Red script to do something else while cmd reads the serial port, you could use a cmd redirection (">") to send the output to a file. In this case, it seems to work only with `call/shell`:

```
Red[]

call/shell form "ComPrinter.exe /devnum 7 /baudrate 9600 /endhex 1A >
input.txt"
print "This is printed immediately, while the input.txt file is still
being created"
```

Unfortunately, you can't write to the serial port while cmd is receiving serial data. And by the way, Windows takes a few seconds to update the file, so if you open "input.txt" too quickly, it may be empty. Of course, it may also be empty because something went wrong...

SerialSend

From webpage: "SerialSend is a little command line application I created to send text strings via a serial port. I mainly use it to send information to microcontroller circuits via a USB-to-serial converter, so it’s designed to work well in that context."

The following command sends the characters “abc 123” via the highest available serial port at the default baud rate (38400 baud).

```
SerialSend.exe "abc 123"
```

Options for SerialSend.exe:

/devnum - Use this to specify a COM port. The default is the highest available com port, including ports >= 10. For example, to set COM7 use `/devnum 7`

/ baudr at e - Use this to specify the baud rate. Default is 38400 bits per second. For example, to set baud rate to 9600, use `/baudrate 9600`

/ hex - Arbitrary bytes, including non-printable characters can be included in the string as hex values using the "/hex" command line option and the "\x" escape sequence in the specified text. For example, the following command sends the string "abc" followed by a line feed character (hex value 0x0A) – i.e. 4 bytes in total. use `SerialSend.exe /hex "abc\x0A"`

Example:

```
Red[]
call form {SerialSend.exe /devnum 7 /baudrate 9600 "abc 123"}
```

Example that sends variables and functions:

```
Red[]
myVariable: "Time now is: "    ; a string
txt: rejoin [{"} myVariable now {"}] ; now returns time and date
command: form rejoin ["SerialSend.exe /devnum 7 /baudrate 115200 " txt]
print command     ;just to help you see what will be sent to cmd
call command
```

Note that I increased the baudrate to 115200 in this second example. That is because I was having troubles at 9600 baud: for some reason, the message was being truncated to about a dozen characters. After many hours trying to isolate the bug (a null modem cable would have helped, but I don't have one at the moment), I gave up and just increased the speed, both in the Red script and in the Arduino sketch. That did not completely fix it, but I could send strings over 200 chars long, which is good enough for now.

A utility similar to SerialSend and ComPrinter, based on the work of Ted Burke, is comsniff - This utility not only prints what it receives on the cmd console, but also sends whatever you type, as you type, to the serial port. I had many problems trying to use it, but it's open source and worth a mention here.

Other useful (?) info in case you really don't want to use external executables:

Sending characters to a COM port: (not extensively tested)

I found useful information about sending characters to the serial port in Windows here. Basically, you may send data to the serial port using:

- `echo hello > COM1`

But this command also sends an extra space, a CR and a LF. Besides, it does not recognize higher port numbers (above 9?). You may choose to send a more universal command as this:

- `set /p x="hello" <nul >\\.\COM22`

Here is a function that uses the first command:

```
Red []
SerialSender: function [stringtosend COMport][
```

```
        command: []
        append command "e "
        append command stringtosend
        append command " > "
        append command COMport
        call/shell form command
    ]

    SerialSender "hello world" "COM7"
```

You may send whole files to the serial port using `copy yourfile.txt com1`, or, for port numbers >= 10, `copy yourfile.txt \\.\COM21`

(Supposed to) redirect serial inputs to a file: (well, kind of tested but...)

These commands are supposed to send the input of a serial port to a file:

- `COPY COM4 data.txt`
- `type com1: >> data.txt`

I've had very bad results with that. Windows' cmd seems to start reading when it pleases and that may take tens of seconds, even minutes, or never at all. Anyway,if you are brave, don't forget to match the baud rate, parity and data size first!

By the way, to stop cmd from recording the data, the device should send a ctrl-z character. That would be `Serial.write ("26")` or `Serial.print("<Stuff>\x1A")` in Arduino. This seems to work with `copy` (when `copy` works at all) but not with `type`.

Terminals:

Here is a nice article about serial terminals.

Terminal - com port development tool - Lovely, very complete, but takes some getting used to.

PuTTY can be configured to work as a very nice serial terminal. It can save your session to a log file.

But to be honest, I mostly just use Arduino IDE's Serial Monitor.

< Previous topic Next topic >

Appendix II -CGI and RSP using Cheyenne server

Red does not have CGI full support as of november 2018. The first chapters here cover the very basic steps using Rebol. I believe that Red behavior will be very similar, if not the same. That does not mean you cannot use Red for CGI. You can find a good reference of how to use it here.

There is plenty of information about CGI in the Internet. However, I had difficulty with the very first steps, specially how to use the minimal Cheyenne server on my own computer, as guinea pig for my tests. So I wrote this as a "get-started-guide", **not** a full comprehensive tutorial about CGI and RSP.

What is CGI

Common Gateway Interface (CGI), is a protocol that allows servers to execute programs that generate web pages dynamically, that is: programs that generate HTML code on-the-fly, "tailored" to the user's input.

CGI has been replaced by a vast variety of web programming technologies. Most developers today use scripting languages like PHP to do what CGI does.

Then why should you bother? Well, maybe you don't want to be a web developer, just connect your Red/Rebol scripts to web browsers, create some webapps, whatever. Web browsers are a great way to display information and interface with the user. And yes, you can get access to the Internet too.

What is RSP

I may be wrong on this, but I believe RSP is a Cheyenne-only thing. Its a kind of simplified way to do CGI, using Rebol embedded in the HTML code. What goes on is that Cheyenne packs a Rebol interpreter embedded in its code, so, unlike regular CGI, where you have to have to call some script interpreter (an executable) to run your script and create the HTML, RSP are files that are interpreted by a sort of native Rebol in Cheyenne. Besides, Cheyenne offers some nice RSP APIs to work with your scripts.

Why Cheyenne?

Because its incredibly small, just about 500 KB! It has one single configuration file and is fully portable. Besides, it's written in Rebol by Nenad Rakocevic and, as mentioned, natively interprets it. You can easily pack the whole thing plus your scripts in a project and still be below 1MB.

Basic HTTP information link:

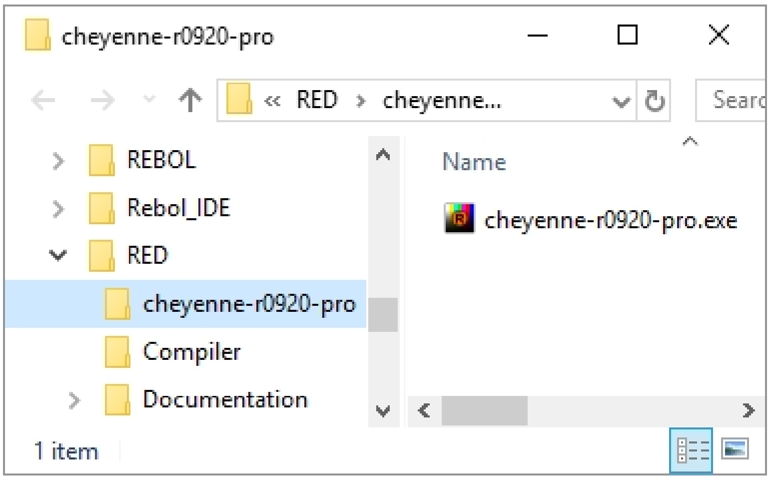
A primer on HTTP - Very good, and has links to more detailed information.

< Previous topic Next topic >

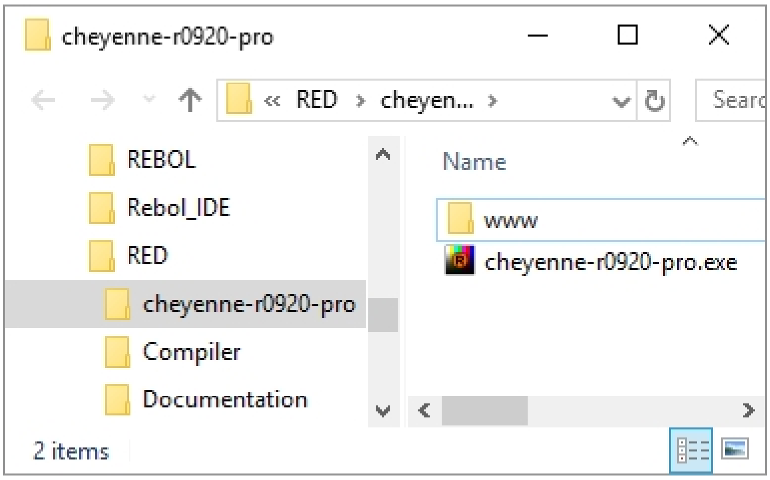
Installing and configuring Cheyenne

Go to https://www.cheyenne-server.org/download.shtml and download the zip. I chose **Cheyenne Pro** because it's smaller, but you may get **Cheyenne Command** if you want some extras.

Unzip it anywhere on your computer. I unziped it in a folder named RED, So I got this:



Now create a folder named "www" inside Cheyenne's folder, like this:

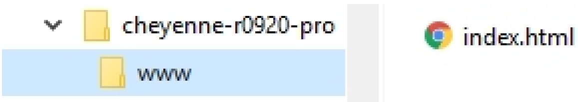


Now copy the HTML below to some pure text editor and save it as **index.html** inside the **www** folder:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
```

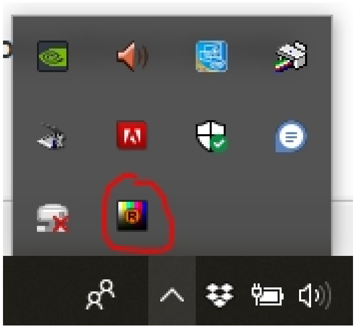
```
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title></title>
</head>
<body>
<h2 style="text-align: center;">Congratulations! Your
Cheyenne server is working!</h2>
<div style="text-align: center;">Have a nice day!</div>
</body>
</html>
```

You should have this:

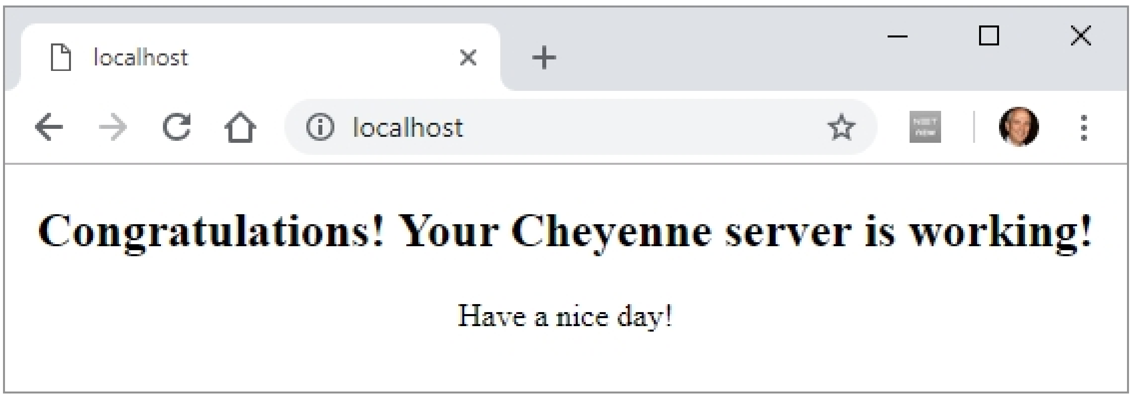


Now double-click on the Cheyenne executable. I had a couple of Windows Defender warnings, I chose **more info/run anyway**.

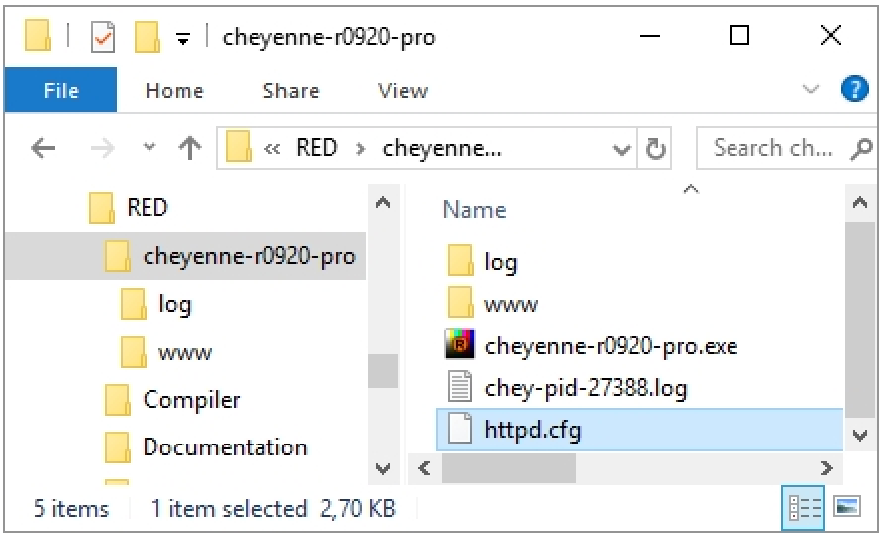
On the task bar, a little Rebol Icon tells me Cheyenne is running:



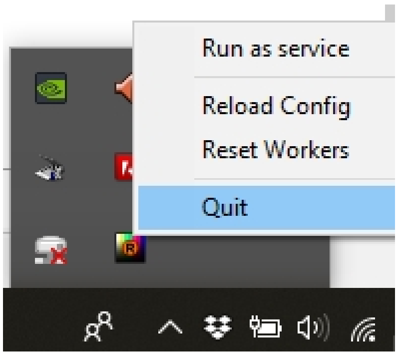
Now open your favorite browser, type "localhost" in the address bar and press enter. You should access the html page you just created:



After this first run, Cheyenne creates a few extra files and folders and it should look like this now:



You may quit Cheyenne right-clicking on the taskbar icon and choosing **Quit**:



Ports are the "channels" of computer communication. By default Cheyenne listens to port 80, but you may want to change that, either to avoid conflicts or to, arguably, add some extra security. A port number is a 16-bit unsigned integer, thus ranging from 0 to 65535, but I suggest you choose a random number around 30000.

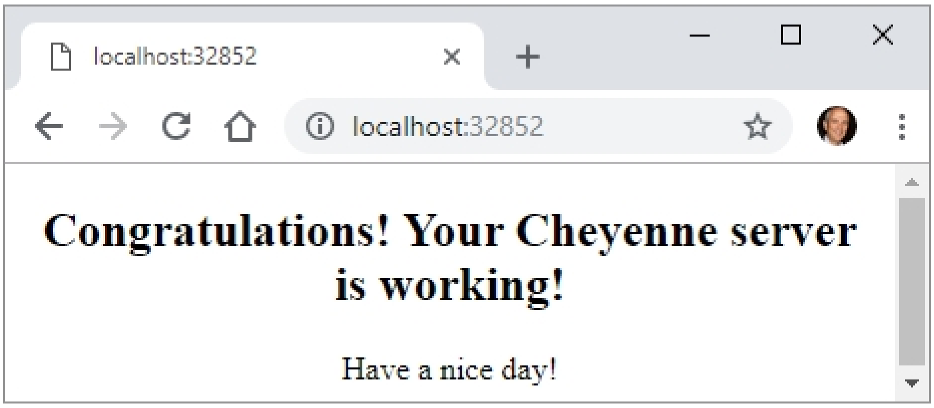
By the way, using Cheyenne as described in this text should be secure, unless you explicitly open your ports on your DSL modem and firewall on your PC.

To change the port Cheyenne listens to, for example, 32852, open the **httpd.cfg** file with any plain text editor, and add the following line:

```
...
;--- define alternative and/or multiple listen ports (by default, cheyenne will run on 80)

;listen [80 10443]

listen [32852]

bind SSI to [.shtml .shtm]

bind php-fcgi to [.php .php3 .php4]
...
```

Or may just uncomment the line above that and change the port numbers (Cheyenne may listen to more than one port).

Now you can access your **index.html** typing in the address bar of your browser **localhost:<port number>** :



For the record, the common port numbers (avoid them) are:

20: File Transfer Protocol (FTP) Data Transfer
21: File Transfer Protocol (FTP) Command Control
22: Secure Shell (SSH) Secure Login
23: Telnet remote login service, unencrypted text messages
25: Simple Mail Transfer Protocol (SMTP) E-mail routing
53: Domain Name System (DNS) service
80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web - *Cheyenne default*
110: Post Office Protocol (POP3)
119: Network News Transfer Protocol (NNTP)
123: Network Time Protocol (NTP)
143: Internet Message Access Protocol (IMAP) Management of digital mail
161: Simple Network Management Protocol (SNMP)
194: Internet Relay Chat (IRC)
443: HTTP Secure (HTTPS) HTTP over TLS/SSL

If you were to remove all commented lines from **httpd.cfg** file (don't do it), you would get the text below, which I think is a self-explanatory simple configuration:

```
modules [
        userdir
        internal
        extapp
        static
        upload
        action
        fastcgi
        rsp
        ssi
```

```
        alias
        socket
]

globals [
        bind SSI to [.shtml .shtm]
        bind php-fcgi to [.php .php3 .php4]
        bind-extern CGI to [.cgi]
        bind-extern RSP to [.j .rsp .r]
]

default [

        root-dir %www/
        default [%index.html %index.rsp %index.php]
        on-status-code [
                404     "/custom404.html"
        ]
        socket-app "/ws.rsp"        ws-test-app
        socket-app "/chat.rsp"      chat
        webapp [
                virtual-root "/testapp"
                root-dir %www/testapp/
                auth "/testapp/login.rsp"
        ]
]
```

< Previous topic   Next topic >

RSP -"Hello world"

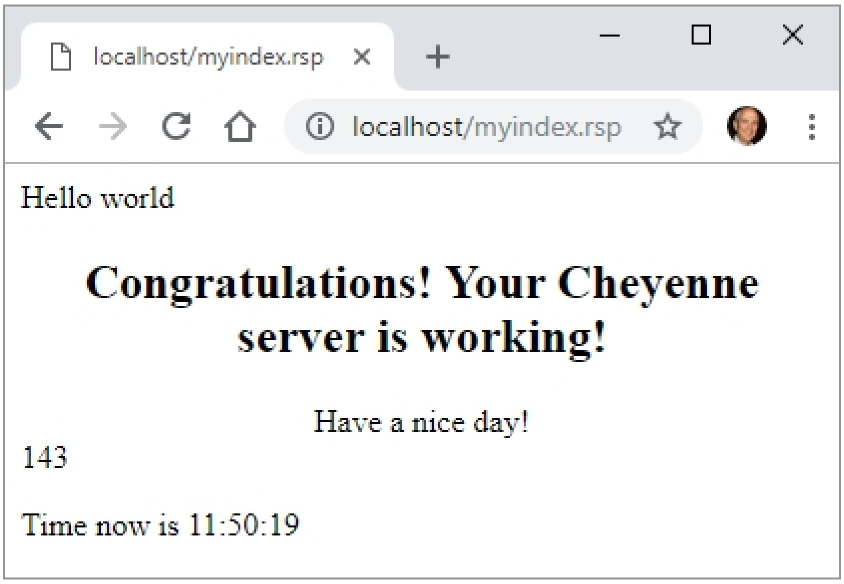
Also check Cheyenne's page about RSP

In RSP scripts, Cheyenne interprets anything in between "**<%**" and "**%>**" as Rebol code!

Open your **index.html** (the one you created in the "Installing and configuring..." chapter) with a plain text editor, add the following highlighted lines and save it in the www folder as **myindex.rsp**.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<% print "Hello world" %>
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title></title>
</head>
<body>
<h2 style="text-align: center;">Congratulations! Your
Cheyenne server is working!</h2>
<div style="text-align: center;">Have a nice day!</div>
<% print 55 + 88 %>
</br>
</body>
</html>
<% print rejoin ["Time now is " now/time] %>
```

With Cheyenne running (listening to default port 80), type **localhost/myindex.rsp** on your browser's address bar. You should get this:



< Previous topic Next topic >

RSP - Request and Response

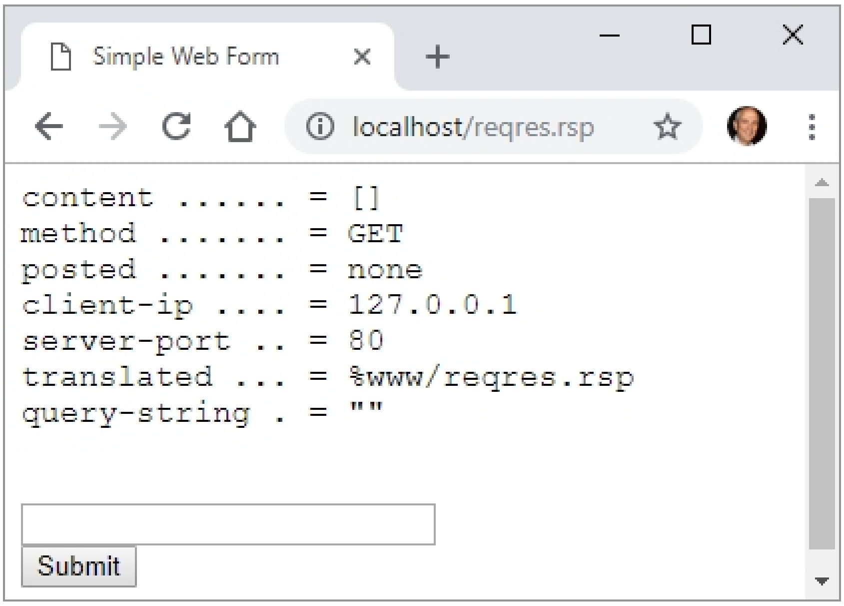
You should refer to this page while reading this.

Requests:

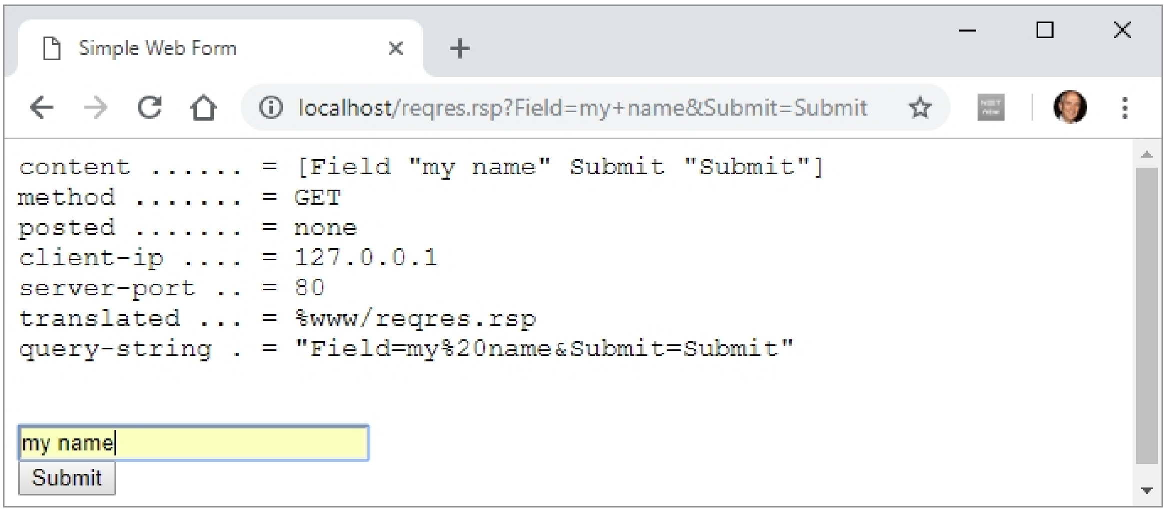
Create the following script on a plain text editor and save it in the www folder as **reqres.rsp**.

```
<%
print {<font face="courier">}
print "content ...... = " probe request/content      print "<br>"
print "method ....... = " probe request/method       print "<br>"
print "posted ....... = " probe request/posted       print "<br>"
print "client-ip .... = " probe request/client-ip    print "<br>"
print "server-port .. = " probe request/server-port  print "<br>"
print "translated ... = " probe request/translated   print "<br>"
print "query-string . = " probe request/query-string print "<br>"
%>
<br><br>
<HTML>
<TITLE>Simple Web Form</TITLE>
<BODY>
<FORM ACTION="reqres.rsp">
<INPUT TYPE="TEXT" NAME="Field" SIZE="25"><BR>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

With Cheyenne running (listening to default port 80), type **localhost/reqres.rsp** on your browser's address bar. You should get this:



Now type something in the field, and press the submit button. Your browser should look like this:



What's happenning:

It's clear that Cheyenne picks the client's (browser) request, decode it, and stores all important values in internal variables of the object **request**.

When you click Submit button, `ACTION="reqres.rsp"` sends you to the same (refreshed) page! But, to do that, the browser sends a **request** that is split and stored in the **request object's** variables, which are shown in the refreshed page.

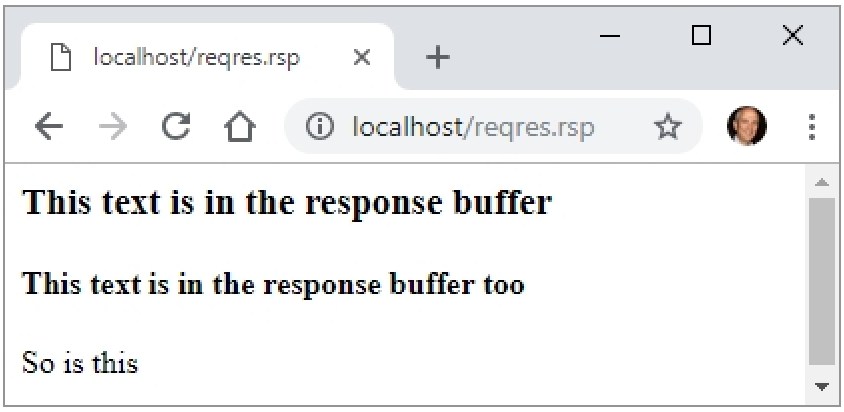
Responses:

In the same way that requests have the **request object**, responses have the **response object**. However, most of this object's fields are functions (actions). The most relevant exception is **response/buffer,** that is where Cheyenne's RSP stores all that is to be sent to the client. It's a block, and so you can manipulate it as any series.

If you change the **reqres.rsp** code to:

```
<%
append response/buffer "<HTML>"
append response/buffer "<h3>This text is in the response buffer</h3>"
append response/buffer "<h4>This text is in the response buffer
too</h4>"
append response/buffer "<p>So is this</p>"
%>
```

You get:

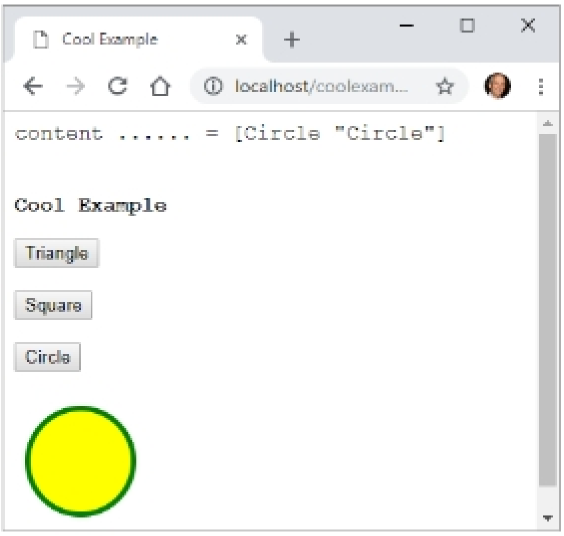


Cool example:

Create and save the following RSP script as **coolexample.rsp** in Cheyenne's **www** folder. Open *localhost/coolexample.rsp* on your browser and click a button. If your browser support HTML's SVG (most do), a corresponding image should show under it's button.

```
<%
print {<font face="courier">}
print "content ...... = " probe request/content      print "<br>"

%>

<HTML>
<br><br>
<TITLE>Cool Example</TITLE>
<BODY>
<b>Cool Example</b><p>
<FORM ACTION="coolexample.rsp">
<INPUT TYPE="SUBMIT" NAME="Triangle" VALUE="Triangle"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Triangle  [
        print {<svg width="100" height="100">
        <polygon points="0,100 50,0 100,100"
        style="fill:lime;stroke:purple;stroke-width:5;fill-
rule:evenodd;" />
        </svg> <br>}
    ]
]
%>
```

```
<INPUT TYPE="SUBMIT" NAME="Square" VALUE="Square"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Square  [
        print {<svg width="100" height="100">
        <rect width="100" height="100" style="fill:rgb(0,0,255);stroke-
width:10;stroke:rgb(0,0,0)" />
        </svg> <br>}
    ]
]
%>
<INPUT TYPE="SUBMIT" NAME="Circle" VALUE="Circle"><br><br>
<%
if not empty? request/content [
    if (first request/content) = 'Circle  [
        print {<svg width="100" height="100">
        <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4"
fill="yellow" />
        </svg> <br>}
    ]
]
%>
</FORM>
</BODY>
</HTML>
```



< Previous topic

Next topic >

CGI - "Hello world"

See also: Quick and Easy CGI - A Beginner's Tutorial and Guide

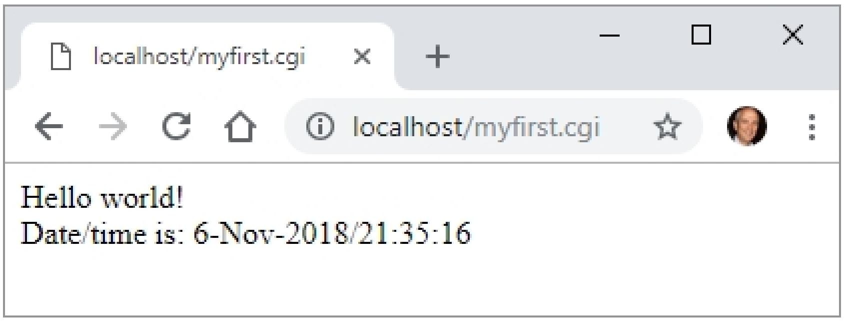
Download "rebol core" interpreter from Rebol's download page. Save that executable to the **www** folder of your Cheyenne.

Now create the following script in a plain text editor and save it as **myfirst.cgi** in the same **www** folder.

```
#!www/rebol.exe -c
REBOL []
print "Hello world!"
print "<br/>"
print ["Date/time is:" now]
```

Your www folder now should look like this:

Now if your server is running (port 80) and you type **localhost/myfirst.cgi** in your browser's address bar, you get:



Explaining the script:

```
#!www/rebol.exe -c             ; This line is very important
                               ; it tells your server the
                               ; path to the interpreter.
                               ; The -c option tells Rebol to
                               ; run on CGI mode.
REBOL []
print "Hello world!"           ; Sends "Hello world!" to the browser.
print "<br/>"                  ; Sends an HTML code for carriage return.
print ["Date/time is:" now]    ; Sends time and date
```

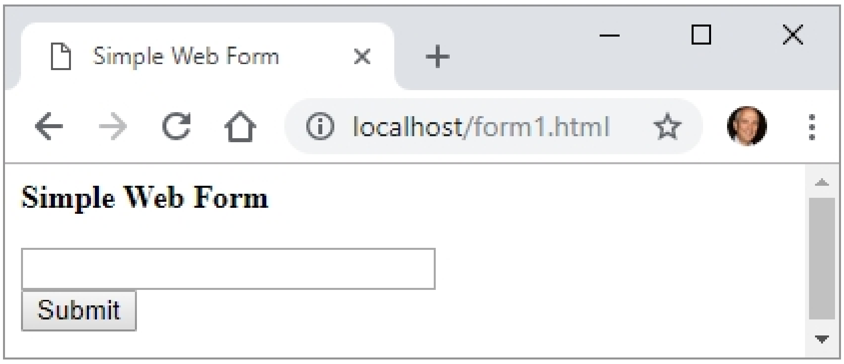
< Previous topic Next topic >

CGI - Processing web forms

See also: Creating and Processing Web Forms with CGI (Tutorial)

Create the following **form1.html** file on your **www** folder:

```
<HTML>
<TITLE>Simple Web Form</TITLE>
<BODY>
<b>Simple Web Form</b><p>
<FORM ACTION="formhandler.cgi">
<INPUT TYPE="TEXT" NAME="Field" SIZE="25"><BR>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
</FORM>
</BODY>
</HTML>
```



Now create and save in the same folder the **formhandler.cgi** script:

```
#!www/rebol.exe -c
Rebol []
print [<HTML><PRE> mold system/options/cgi </HTML>]
```

When you write "My Name" in the field and press the Submit button, your **form1.html** will call **formhandler.cgi**, and this script will print what the CGI protocol passes to Rebol and is stored in **system/options/cgi** which is:

```
make object! [
   server-software: "Cheyenne/1.0"
   server-name: "Ungaretti"
   gateway-interface: "CGI/1.1"
   server-protocol: "HTTP/1.1"
   server-port: "80"
   request-method: "GET"
   path-info: "/formhandler.cgi"
   path-translated: "www\formhandler.cgi"
   script-name: "/formhandler.cgi"
   query-string: "Field=My+Name&Submit=Submit"
```

```
    remote-host: none
    remote-addr: "127.0.0.1"
    auth-type: none
    remote-user: none
    remote-ident: none
    Content-Type: none
    content-length: "0"
    other-headers:
["HTTP_ACCEPT" {text/html,application/xhtml+xml,application/...
]
```

This is good to know, but Rebol offers a function to decode the CGI, named `decode-cgi` that converts the raw form data into a REBOL block that contains words followed by their values. The information we want (the contents of the field), are in the query-string variable. So change **formhandler.cgi** script as follows:

```
#!www/rebol.exe -c
Rebol []
print [<HTML><PRE> decode-cgi system/options/cgi/query-string </HTML>]
```

The browser output now is:

```
 Field My Name Submit Submit
```

CGI cool example

This is the CGI version of the RSP's cool example. Save it as **coolexample.cgi** in Cheyenne's **www** folder. Open in browser using *localhost/coolexample.cgi.*

```
#!www/rebol.exe -c
Rebol []
; First, a not very elegant way of avoiding crashes:
either system/options/cgi/query-string = none [
    system/options/cgi/query-string: ""
    decoded: ""
    ][
    decoded: second decode-cgi system/options/cgi/query-string
]

; Lets show what's in "decoded":
print {<font face="courier">}
print "decoded = " probe decoded      print "<br>"

; Here we start HTML
print {
    <HTML>
    <br><br>
    <TITLE>Cool Example</TITLE>
    <BODY>
    <b>Cool Example</b><p>
    <FORM ACTION="coolexample.cgi">}

print {<INPUT TYPE="SUBMIT" NAME="Triangle" VALUE="Triangle"><br><br>}
 if decoded = "Triangle"  [
     print {<svg width="120" height="120">
    <polygon points="0,100 50,0 100,100"
    style="fill:lime;stroke:purple;stroke-width:5;fill-rule:evenodd;" />
    </svg> <br>}
```

```
]


print {<INPUT TYPE="SUBMIT" NAME="Square" VALUE="Square"><br><br>}
if decoded = "Square"  [
    print {<svg width="120" height="120">
    <rect width="100" height="100" style="fill:rgb(0,0,255);stroke-
width:10;stroke:rgb(0,0,0)" />
    </svg> <br>}
]


print {<INPUT TYPE="SUBMIT" NAME="Circle" VALUE="Circle"><br><br>}
if decoded = "Circle"  [
    print {<svg width="120" height="120">
    <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4"
fill="yellow" />
    </svg> <br>}
]

; Now we finish HTML
print {
    </FORM>
    </BODY>
    </HTML>}
```

< Previous topic Next topic >

CGI using Red

Hello World!

See also: *Using Red as CGI*

Make a copy of the Red interpreter and save that executable to the **www** folder of your Cheyenne, just like you did to Rebol.

Rename Red's executable to something like **redcgi.exe**. I found that to be important because I have Red already "installed" in my computer (where my server is running - localhost), and the operating system tries to just run the script, not "CGI it".

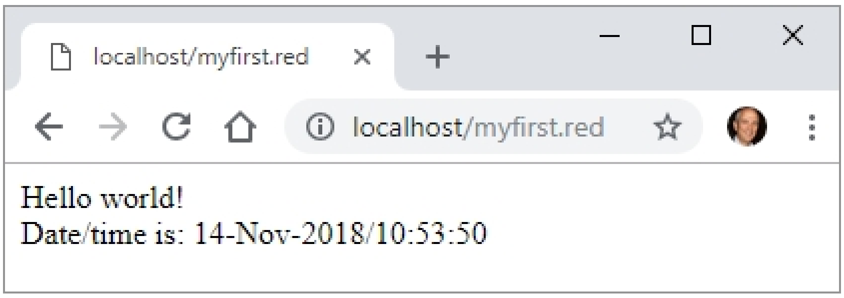
Open the **httpd.cfg** file in a plain text editor, and add **.red** to the "bind-extern CGI to" block, as shown:

```
globals [
	;--- define alternative and/or multiple listen ports (by default, cheyenne
will run on 80)
	;listen [80 10443]
	bind SSI to [.shtml .shtm]
	bind php-fcgi to [.php .php3 .php4]
	bind-extern CGI to [.cgi .red]
	bind-extern RSP to [.j .rsp .r]
```

Now create the following script in a plain text editor and save it as **myfirst.red** in the same **www** folder. --cli is important, otherwise Red may try to compile and open the GUI console.

```
#!www/redcgi.exe --cli
Red []
print "Hello world!"
print "<br/>"
print ["Date/time is:" now]
```

Now if your server is running (port 80) and you type **localhost/myfirst.red** in your browser's address bar, you get:



Processing web forms.

As mentioned, Red does not have yet full support for CGI. However, I believe it's possible to retrieve and decode HTTP messages in **Linux**, using Boleslav Březovský's http-tools.red . I don't know how to do that in Windows.

< Previous topic Next topic >

Appendix III -MQTT using Red

MQTT has become **the** popular protocol for IoT (Internet of Things) communication. On the Internet Protocol Stack, it works on the same layer as HTTP, but MQTT is lighter, uses less bandwidth, and allows keeping a steady line to devices and near real time communication.

Unlike CGI or serial port support, MQTT is not a priority in Red's development, and it will depend on the community to create native libraries. However, it's possible to publish and subscribe to topics (as client) using Red and some external executables and DLLs.

I'll not go in details about MQTT, I assume you know the basics of it. In case you don't, the best information I found is in the Hivemq tutorials.

To monitor MQTT messages, you can use any of the tools listed here. I use MQTT-spy, but any client utility will do, including some Android apps that you can install on your phone (search Google-Play).

I used a free "Cute cat" account on CloudMQTT MQTT broker for my tests.

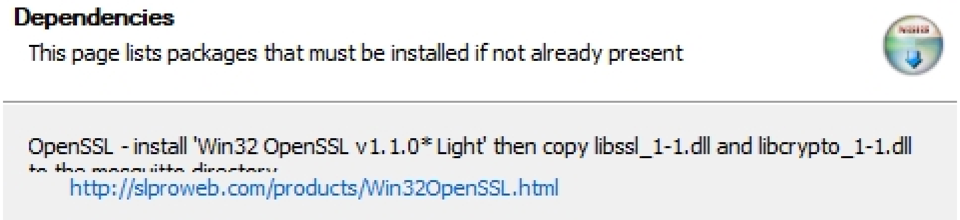
What you need:

You must have in your script's folder:

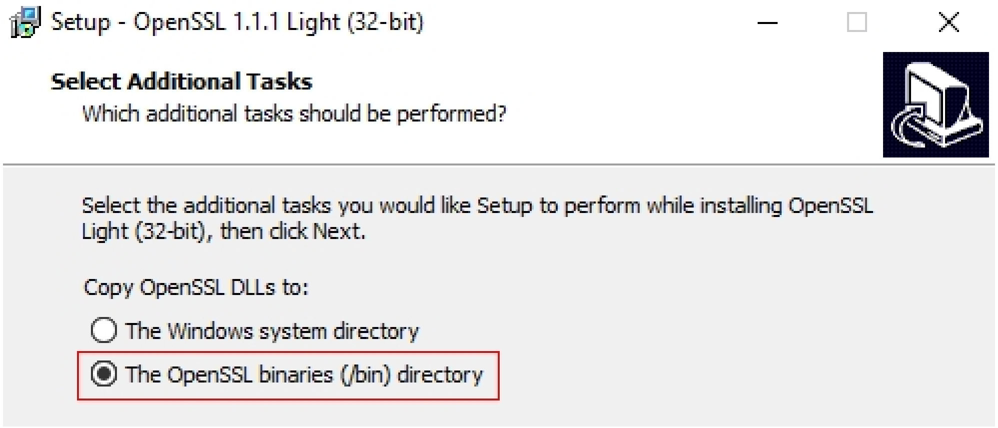
- mosquitto_pub.exe
- mosquitto_sub.exe
- mosquitto.dll
- libssl-1_1.dll
- libcrypto-1_1.dll

I obtained **mosquitto_pub.exe, mosquitto_sub.exe** and **mosquitto.dll** by installing mosquitto downloaded from here. I used the 32bit install. These files are in the "mosquitto" folder created by installation.

During installation, you get the following warning:



The **libssl-1_1.dll** and **libcrypto-1_1.dll** are files of the OpenSSL toolkit. So, as recommended, I downloaded OpenSSL from http://slproweb.com/products/Win32OpenSSL.html and installed it. During installation, make sure you choose to install the DLLs to OpeSSL folder, it will make them a lot easier to find:



Then I copied and pasted **libssl-1_1.dll** and **libcrypto-1_1.dll** not only to mosquitto directory, but also to my script's folder.

To understand the use of **mosquitto_pub.exe** check this page, and for **mosquitto_sub.exe** there is this page. A good page with examples is Using The Mosquitto_pub and Mosquitto_sub MQTT Client Tools- Examples, and its respective video.

Publishing:

The following script is a crude MQTT publisher. It doesn't offer many options, but it's enough to show how to create a **mosqutto_pub** command line:

```
Red [needs view]
view [
    text "broker:" 50 right broker: field "m12.cloudmqtt.com" 150
    text "port:" 30 right port: field "13308" 50
    text "user:" 30 right user: field "qenkXXX"
    text "password:" 60 right password: field "CRfa8kuXXX" 120
    return
    text "topic:" 50 right topic: field 200 "/test"
    text "message" 60 right message: field 300 "Hello World!"
    return
    button "Publish" [
        call rejoin ["mosquitto_pub.exe -h " broker/text " -p "
port/text " -u " user/text
        " -P " password/text { -t "} topic/text {"} { -m "} message/text
 {"}
        ]
    ]
]
```

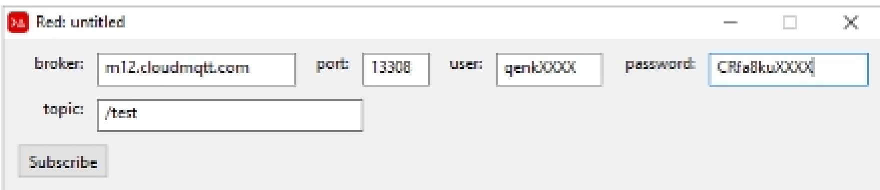
You can use `print` instead of `call` in the script above to see the full command passed to **mosquitto_pub.exe.**

Subscribing:

Subscribing using **mosquitto_sub.exe** is a little less straightforward, because it outputs the published messages on cmd's CLI console. I haven't figured out how to constantly feed this to a Red script. My solution so far is to redirect the output of **mosquitto_sub.exe** to a text file and pool it constantly to detect any file size changes. If it changes, the Red script reads it to get the new messages.

This script subscribes the topic and redirects the outputs to **mqttlog.txt** using cmd redirection command ">":

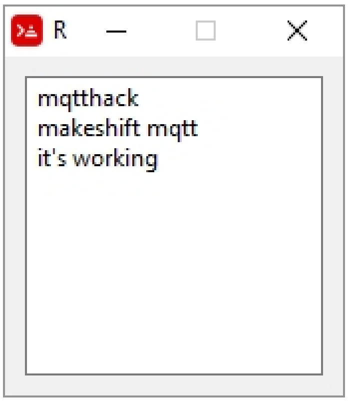
```
Red [needs view]
view [
    text "broker:" 50 right broker: field "m12.cloudmqtt.com" 150
    text "port:" 30 right port: field "13308" 50
    text "user:" 30 right user: field "qenkXXXX"
    text "password:" 60 right password: field "CRfa8kuXXXX" 120
    return
    text "topic:" 50 right topic: field 200 "/test"
    return
    button "Subscribe" [
        call/shell rejoin ["mosquitto_sub.exe -h " broker/text " -p "
port/text " -u " user/text
        " -P " password/text { -t "} topic/text {" > mqttlog.txt}
        ]
    ]
]
```



And this script constantly checks **mqttlog.txt** for updates and puts them on an `area`:

```
Red [needs: view]
oldsize: 0
view [
    mqttlog: area rate 2 ;checks txt file twice per second
    on-time [
        newsize: size? %"mqttlog.txt"
        if newsize <> oldsize [
```

```
                mqttlog/text: read %"mqttlog.txt"
                oldsize: newsize
            ]
        ]
    ]
```



< Previous topic