class ARGF

ARGF is a stream designed for use in scripts that process files given as command-line arguments or passed in via STDIN.

The arguments passed to your script are stored in the ARGV Array, one argument per element. ARGF assumes that any arguments that aren't filenames have been removed from ARGV.

For example:

```
# if argf.rb contains
p ARGV
p option = ARGV.shift
p ARGV
# then
$ ruby argf.rb --verbose file1 file2
# will output
["--verbose", "file1", "file2"]
"--verbose"
["file1", "file2"]
```

You can now use ARGF to work with a concatenation of each of these named files. For instance, ARGF.read will return the contents of file1 followed by the contents of file2.

After a file in ARGV has been read ARGF removes it from the Array. Thus, after all files have been read ARGV will be empty.

You can manipulate ARGV yourself to control what ARGF operates on. If you remove a file from ARGV, it is ignored by ARGF; if you add files to ARGV, they are treated as if they were named on the command line.

For example:

```
# argf.rb contains
p ARGV.replace ["file1"]
p ARGF.readlines
p ARGV
p ARGV.replace ["file2", "file3"]
p ARGF.read
# then
$ ruby argf.rb file1 file2 file3
# will output
["file1"]
["I am file1\n"]
[]
["file2", "file3"]
"I am file2\nI am file3\n"
```

If ARGV is empty, ARGF acts as if it contained STDIN, i.e. the data piped to your script.

For example:

```
$ echo "glark" | ruby -e 'p ARGF.read'
"glark\n"
```

In Files

io.c

Parent

Object

Included Modules

Enumerable

Public Instance Methods

----------;

argv → ARGV

Returns the ARGV array, which contains the arguments passed to your script, one per element.

For example:

```
$ ruby argf.rb -v glark.txt

p ARGF.argv   #=> ["-v", "glark.txt"]
```

----------;

binmode → ARGF

Puts ARGF into binary mode. Once a stream is in binary mode, it cannot be reset to non-binary mode. This option has the following effects:

Newline conversion is disabled.

Encoding conversion is disabled.

Content is treated as ASCII-8BIT.

---;

binmode? → true or false

Returns true if ARGF is being read in binary mode; false otherwise. (To enable binary mode use ARGF.binmode.

For example:

```
ARGF.binmode?  #=> false
ARGF.binmode
ARGF.binmode?  #=> true
```

---;

bytes()

This is a deprecated alias for each_byte.

chars()

This is a deprecated alias for each_char.

---;

close → ARGF

Closes the current file and skips to the next in the stream. Trying to close a file that has already been closed causes an IOError to be raised.

For example:

```
$ ruby argf.rb foo bar

ARGF.filename  #=> "foo"
ARGF.close
ARGF.filename  #=> "bar"
ARGF.close
ARGF.close     #=> closed stream (IOError)
```

---;

closed? → true or false

Returns true if the current file has been closed; false otherwise. Use ARGF.close to actually close the current file.

---;

codepoints()

This is a deprecated alias for each_codepoint.

---;

each(sep=$/) {|line| block } → ARGF

each(sep=$/,limit) {|line| block } → ARGF

each(...) → an_enumerator

Returns an enumerator which iterates over each line (separated by sep, which defaults to your platform's newline character) of each file in ARGV. If a block is supplied, each line in turn will be yielded to the block, otherwise an enumerator is returned. The optional limit argument is a Fixnum specifying the maximum length of each line; longer lines will be split according to this limit.

This method allows you to treat the files supplied on the command line as a single file consisting of the concatenation of each named file. After the last line of the first file has been returned, the first line of the second file is returned. The ARGF.filename and ARGF.lineno methods can be used to determine the filename and line number, respectively, of the current line.

For example, the following code prints out each line of each named file prefixed with its line number, displaying the filename once per file:

```
ARGF.lines do |line|
  puts ARGF.filename if ARGF.lineno == 1
  puts "#{ARGF.lineno}: #{line}"
end
```

---;

bytes {|byte| block } → ARGF

bytes → an_enumerator

each_byte {|byte| block } → ARGF

each_byte → an_enumerator

Iterates over each byte of each file in ARGV. A byte is returned as a Fixnum in the range 0..255.

This method allows you to treat the files supplied on the command line as a single file consisting of the concatenation of each named file. After the last byte of the first file has been returned, the first byte of the second file is returned. The ARGF.filename method can be used to determine the filename of the current byte.

If no block is given, an enumerator is returned instead.For example:

```
ARGF.bytes.to_a  #=> [35, 32, ... 95, 10]
```

---;

each_char {|char| block } → ARGF

each_char → an_enumerator

Iterates over each character of each file in ARGF.

This method allows you to treat the files supplied on the command line as a single file consisting of the concatenation of each named file. After the last character of the first file has been returned, the first character of the second file is returned. The ARGF.filename method can be used to determine the name of the file in which the current character appears.

If no block is given, an enumerator is returned instead.

---;

each_codepoint {|codepoint| block } → ARGF

each_codepoint → an_enumerator

Iterates over each codepoint of each file in ARGF.

This method allows you to treat the files supplied on the command line as a single file consisting of the concatenation of each named file. After the last codepoint of the first file has been returned, the first codepoint of the second file is returned. The ARGF.filename method can be used to determine the name of the file in which the current codepoint appears.

If no block is given, an enumerator is returned instead.

---;

each_line(sep=$/) {|line| block } → ARGF

each_line(sep=$/,limit) {|line| block } → ARGF

each_line(...) → an_enumerator

Returns an enumerator which iterates over each line (separated by sep, which defaults to your platform's newline character) of each file in ARGV. If a block is supplied, each line in turn will be yielded to the block, otherwise an enumerator is returned. The optional limit argument is a Fixnum specifying the maximum length of each line; longer lines will be split according to this limit.

This method allows you to treat the files supplied on the command line as a single file consisting of the concatenation of each named file. After the last line of the first file has been returned, the first line of the second file is returned. The ARGF.filename and ARGF.lineno methods can be used to determine the filename

and line number, respectively, of the current line.

For example, the following code prints out each line of each named file prefixed with its line number, displaying the filename once per file:

```
ARGF.lines do |line|
  puts ARGF.filename if ARGF.lineno == 1
  puts "#{ARGF.lineno}: #{line}"
end
```

---;

eof? → true or false

eof → true or false

Returns true if the current file in ARGF is at end of file, i.e. it has no data to read. The stream must be opened for reading or an IOError will be raised.

```
$ echo "eof" | ruby argf.rb

ARGF.eof?                 #=> false
3.times { ARGF.readchar }
ARGF.eof?                 #=> false
ARGF.readchar             #=> "\n"
ARGF.eof?                 #=> true
```

---;

external_encoding → encoding

Returns the external encoding for files read from ARGF as an Encoding object. The external encoding is the encoding of the text as stored in a file. Contrast with ARGF.internal_encoding, which is the encoding used to represent this text within Ruby.

To set the external encoding use ARGF.set_encoding.For example:

```
ARGF.external_encoding  #=>  #<Encoding:UTF-8>
```

---;

file → IO or File object

Returns the current file as an IO or File object. #<IO:> is returned when the current file is STDIN.

For example:

```
$ echo "foo" > foo
$ echo "bar" > bar

$ ruby argf.rb foo bar

ARGF.file      #=> #<File:foo>
ARGF.read(5)   #=> "foo\nb"
ARGF.file      #=> #<File:bar>
```

---;

filename → String

Returns the current filename. "-" is returned when the current file is STDIN.

For example:

```
$ echo "foo" > foo
$ echo "bar" > bar
$ echo "glark" > glark

$ ruby argf.rb foo bar glark

ARGF.filename  #=> "foo"
ARGF.read(5)   #=> "foo\nb"
ARGF.filename  #=> "bar"
ARGF.skip
ARGF.filename  #=> "glark"
```

---;

fileno → fixnum

Returns an integer representing the numeric file descriptor for the current file. Raises an ArgumentError if there isn't a current file.

```
ARGF.fileno    #=> 3
```

---;

getbyte → Fixnum or nil

Gets the next 8-bit byte (0..255) from ARGF. Returns nil if called at the end of the stream.

For example:

```
$ echo "foo" > file
$ ruby argf.rb file

ARGF.getbyte #=> 102
ARGF.getbyte #=> 111
ARGF.getbyte #=> 111
ARGF.getbyte #=> 10
ARGF.getbyte #=> nil
```

----------;

getc → String or nil

Reads the next character from ARGF and returns it as a String. Returns nil at the end of the stream.

ARGF treats the files named on the command line as a single file created by concatenating their contents. After returning the last character of the first file, it returns the first character of the second file, and so on.

For example:

```
$ echo "foo" > file
$ ruby argf.rb file

ARGF.getc  #=> "f"
ARGF.getc  #=> "o"
ARGF.getc  #=> "o"
ARGF.getc  #=> "\n"
ARGF.getc  #=> nil
ARGF.getc  #=> nil
```

----------;

gets(sep=$/) → string

gets(limit) → string

gets(sep, limit) → string

Returns the next line from the current file in ARGF.

By default lines are assumed to be separated by $/; to use a different character as a separator, supply it as a String for the sep argument.

The optional limit argument specifies how many characters of each line to return. By default all characters are returned. ----------;

inplace_mode → String

Returns the file extension appended to the names of modified files under inplace-edit mode. This value can be set using ARGF.inplace_mode= or passing the -i switch to the Ruby binary. ----------;

inplace_mode = ext → ARGF

Sets the filename extension for inplace editing mode to the given String. Each file being edited has this value appended to its filename. The modified file is saved under this new name.

For example:

```
$ ruby argf.rb file.txt

ARGF.inplace_mode = '.bak'
ARGF.lines do |line|
  print line.sub("foo","bar")
end
```

Each line of file.txt has the first occurrence of “foo” replaced with “bar”, then the new line is written out to file.txt.bak. ----------;

inspect()

Alias for: to_s internal_encoding → encoding click to toggle source Returns the internal encoding for strings read from ARGF as an Encoding object.

If ARGF.set_encoding has been called with two encoding names, the second is returned. Otherwise, if Encoding.default_external has been set, that value is returned. Failing that, if a default external encoding was specified on the command-line, that value is used. If the encoding is unknown, nil is returned. ----------;

lineno → integer

Returns the current line number of ARGF as a whole. This value can be set manually with ARGF.lineno=.

For example:

```
ARGF.lineno   #=> 0
ARGF.readline #=> "This is line 1\n"
ARGF.lineno   #=> 1
```

----------;

lineno = integer → integer

Sets the line number of ARGF as a whole to the given Integer.

ARGF sets the line number automatically as you read data, so normally you will not need to set it explicitly. To access the current line number use ARGF.lineno.

For example:

```
ARGF.lineno      #=> 0
ARGF.readline    #=> "This is line 1\n"
ARGF.lineno      #=> 1
ARGF.lineno = 0  #=> 0
ARGF.lineno      #=> 0
```

----------;

lines(*args)

This is a deprecated alias for each_line. ----------;

path → String

Returns the current filename. "-" is returned when the current file is STDIN.

For example:

```
$ echo "foo" > foo
$ echo "bar" > bar
$ echo "glark" > glark

$ ruby argf.rb foo bar glark

ARGF.filename  #=> "foo"
ARGF.read(5)   #=> "foo\nb"
ARGF.filename  #=> "bar"
ARGF.skip
ARGF.filename  #=> "glark"
```

----------;

pos → Integer

Returns the current offset (in bytes) of the current file in ARGF.

```
ARGF.pos    #=> 0
ARGF.gets   #=> "This is line one\n"
```

```
ARGF.pos    #=> 17
```

---;

pos = position → Integer

Seeks to the position given by position (in bytes) in ARGF.

For example:

```
ARGF.pos = 17
ARGF.gets   #=> "This is line two\n"
```

---;

print() → nil

print(obj, ...) → nil

Writes the given object(s) to ios. The stream must be opened for writing. If the output field separator ($,) is not nil, it will be inserted between each object. If the output record separator ($</code>) is not nil, it will be appended to the output. If no arguments are given, prints $. *Objects that aren't strings will be converted by calling their to_s method. With no argument, prints the contents of the variable $*. Returns nil.

```
$stdout.print("This is ", 100, " percent.\n")

produces:

This is 100 percent.
```

---;

printf(format_string [, obj, ...]) → nil

Formats and writes to ios, converting parameters under control of the format string. See Kernel#sprintf for details. ---;

putc(obj) → obj

If obj is Numeric, write the character whose code is the least-significant byte of obj, otherwise write the first byte of the string representation of obj to ios. Note: This method is not safe for use with multi-byte characters as it will truncate them.

```
$stdout.putc "A"
$stdout.putc 65
```

```
produces:

AA
```

---;

puts(obj, ...) → nil

Writes the given objects to ios as with IO#print. Writes a record separator (typically a newline) after any that do not already end with a newline sequence. If called with an array argument, writes each element on a new line. If called without arguments, outputs a single record separator.

```
$stdout.puts("this", "is", "a", "test")

produces:

this
is
a
test
```

---;

read([length [, outbuf]]) → string, outbuf, or nil

Reads *length* bytes from ARGF. The files named on the command line are concatenated and treated as a single file by this method, so when called without arguments the contents of this pseudo file are returned in their entirety.

length must be a non-negative integer or nil. If it is a positive integer, read tries to read at most *length* bytes. It returns nil if an EOF was encountered before anything could be read. Fewer than *length* bytes may be returned if an EOF is encountered during the read.

If *length* is omitted or is *nil*, it reads until EOF. A String is returned even if EOF is encountered before any data is read.

If *length* is zero, it returns "".

If the optional *outbuf* argument is present, it must reference a String, which will receive the data. The *outbuf* will contain only the received data after the method call even if it is not empty at the beginning.For example:

```
    $ echo "small" > small.txt
    $ echo "large" > large.txt
    $ ./glark.rb small.txt large.txt

    ARGF.read      #=> "small\nlarge"
    ARGF.read(200) #=> "small\nlarge"
    ARGF.read(2)   #=> "sm"
```

```
ARGF.read(0)   #=> ""
```

Note that this method behaves like fread() function in C. If you need the behavior like read(2) system call, consider ARGF.readpartial. ----------;

read_nonblock(maxlen) → string

read_nonblock(maxlen, outbuf) → outbuf

Reads at most maxlen bytes from the ARGF stream in non-blocking mode. ----------;

readbyte → Fixnum

Reads the next 8-bit byte from ARGF and returns it as a Fixnum. Raises an EOFError after the last byte of the last file has been read.

For example:

```
$ echo "foo" > file
$ ruby argf.rb file

ARGF.readbyte  #=> 102
ARGF.readbyte  #=> 111
ARGF.readbyte  #=> 111
ARGF.readbyte  #=> 10
ARGF.readbyte  #=> end of file reached (EOFError)
```

----------;

readchar → String or nil

Reads the next character from ARGF and returns it as a String. Raises an EOFError after the last character of the last file has been read.

For example:

```
$ echo "foo" > file
$ ruby argf.rb file

ARGF.readchar  #=> "f"
ARGF.readchar  #=> "o"
ARGF.readchar  #=> "o"
ARGF.readchar  #=> "\n"
ARGF.readchar  #=> end of file reached (EOFError)
```

readline(sep=$/) → string

readline(limit) → string

readline(sep, limit) → string

Returns the next line from the current file in ARGF.

By default lines are assumed to be separated by $/; to use a different character as a separator, supply it as a String for the sep argument.

The optional limit argument specifies how many characters of each line to return. By default all characters are returned.

An EOFError is raised at the end of the file. ----------;

readlines(sep=$/) → array

readlines(limit) → array

readlines(sep, limit) → array

Reads ARGF's current file in its entirety, returning an Array of its lines, one line per element. Lines are assumed to be separated by sep.

```
lines = ARGF.readlines
lines[0]                #=> "This is line one\n"
```

----------;

readpartial(maxlen) → string

readpartial(maxlen, outbuf) → outbuf

Reads at most maxlen bytes from the ARGF stream. It blocks only if ARGF has no data immediately available. If the optional outbuf argument is present, it must reference a String, which will receive the data. The outbuf will contain only the received data after the method call even if it is not empty at the beginning. It raises EOFError on end of file.

readpartial is designed for streams such as pipes, sockets, and ttys. It blocks only when no data is immediately available. This means that it blocks only when following all conditions hold:

The byte buffer in the IO object is empty.

The content of the stream is empty.

The stream has not reached EOF.

When readpartial blocks, it waits for data or EOF. If some data is read, readpartial returns with the data. If EOF is reached, readpartial raises an EOFError.

When readpartial doesn't block, it returns or raises immediately. If the byte buffer is not empty, it returns the data in the buffer. Otherwise, if the stream has some content, it returns the data in the stream. If the stream reaches EOF an EOFError is raised. ----------

----------;

rewind → 0

Positions the current file to the beginning of input, resetting ARGF.lineno to zero.

```
ARGF.readline   #=> "This is line one\n"
ARGF.rewind     #=> 0
ARGF.lineno     #=> 0
ARGF.readline   #=> "This is line one\n"
```

----------;

seek(amount, whence=IO::SEEK_SET) → 0

Seeks to offset amount (an Integer) in the ARGF stream according to the value of whence. See IO#seek for further details. ----------;

set_encoding(ext_enc) → ARGF

set_encoding("ext_enc:int_enc") → ARGF

set_encoding(ext_enc, int_enc) → ARGF

set_encoding("ext_enc:int_enc", opt) → ARGF

set_encoding(ext_enc, int_enc, opt) → ARGF

If single argument is specified, strings read from ARGF are tagged with the encoding specified.

If two encoding names separated by a colon are given, e.g. “ascii:utf-8”, the read string is converted from the first encoding (external encoding) to the second encoding (internal encoding), then tagged with the second encoding.

If two arguments are specified, they must be encoding objects or encoding names. Again, the first specifies the external encoding; the second specifies the internal encoding.

If the external encoding and the internal encoding are specified, the optional Hash argument can be used to adjust the conversion process. The structure of this hash is explained in the String#encode documentation.

For example:

```
ARGF.set_encoding('ascii')         # Tag the input as US-ASCII text
ARGF.set_encoding(Encoding::UTF_8) # Tag the input as UTF-8 text
ARGF.set_encoding('utf-8','ascii') # Transcode the input from US-ASCII
                                   # to UTF-8.
```

----------;

skip → ARGF

Sets the current file to the next file in ARGV. If there aren't any more files it has no effect.For example:

```
$ ruby argf.rb foo bar
ARGF.filename  #=> "foo"
ARGF.skip
ARGF.filename  #=> "bar"
```

---;

tell → Integer

Returns the current offset (in bytes) of the current file in ARGF.

```
ARGF.pos    #=> 0
ARGF.gets   #=> "This is line one\n"
ARGF.pos    #=> 17
```

---;

to_a(sep=$/) → array

to_a(limit) → array

to_a(sep, limit) → array

Reads ARGF's current file in its entirety, returning an Array of its lines, one line per element. Lines are assumed to be separated by sep.

```
lines = ARGF.readlines
lines[0]                #=> "This is line one\n"
```

---;

to_i → fixnum

Returns an integer representing the numeric file descriptor for the current file. Raises an ArgumentError if there isn't a current file.

```
ARGF.to_i    #=> 3
```

---;

to_io → IO

Returns an IO object representing the current file. This will be a File object unless the current file is a stream such as STDIN.

For example:

```
ARGF.to_io    #=> #<File:glark.txt>
ARGF.to_io    #=> #<IO:<STDIN>>
```

----------;

to_s → String

Returns "ARGF".

Also aliased as: inspect ----------;

to_write_io → io

Returns IO instance tied to ARGF for writing if inplace mode is enabled. ----------;

write(string) → integer

Writes string if inplace mode.

----------;