Base / Strings Edit on GitHub

Strings

Core.AbstractChar — Type

The AbstractChar type is the supertype of all character implementations in Julia. A character represents a Unicode code point, and can be converted to an integer via the codepoint function in order to obtain the numerical value of the code point, or constructed from the same integer. These numerical values determine how characters are compared with < and ==, for example. New T <: AbstractChar types should define a codepoint(::T) method and a T(::UInt32) constructor, at minimum.

A given AbstractChar subtype may be capable of representing only a subset of Unicode, in which case conversion from an unsupported UInt32 value may throw an error. Conversely, the built-in Char type represents a *superset* of Unicode (in order to losslessly encode invalid byte streams), in which case conversion of a non-Unicode value *to* UInt32 throws an error. The isvalid function can be used to check which codepoints are representable in a given AbstractChar type.

Internally, an AbstractChar type may use a variety of encodings. Conversion via codepoint(char) will not reveal this encoding because it always returns the Unicode value of the character. print(io, c) of any c::AbstractChar produces an encoding determined by io (UTF-8 for all built-in IO types), via conversion to Char if necessary.

write(io, c), in contrast, may emit an encoding depending on typeof(c), and read(io, typeof(c)) should read the same encoding as write. New AbstractChar types must provide their own implementations of write and read.

Core.Char — Type

```
Char(c::Union{Number,AbstractChar})
```

Char is a 32-bit AbstractChar type that is the default representation of characters in Julia. Char is the type used for character literals like 'x' and it is also the element type of String.

In order to losslessly represent arbitrary byte streams stored in a String, a Char value may store information that cannot be converted to a Unicode codepoint — converting such a Char to

`UInt32` will throw an error. The `isvalid(c::Char)` function can be used to query whether `c` represents a valid Unicode character.

`Base.codepoint` — Function

```
codepoint(c::AbstractChar) -> Integer
```

Return the Unicode codepoint (an unsigned integer) corresponding to the character `c` (or throw an exception if `c` does not represent a valid character). For `Char`, this is a `UInt32` value, but `AbstractChar` types that represent only a subset of Unicode may return a different-sized integer (e.g. `UInt8`).

`Base.length` — Method

```
length(s::AbstractString) -> Int
length(s::AbstractString, i::Integer, j::Integer) -> Int
```

The number of characters in string `s` from indices `i` through `j`. This is computed as the number of code unit indices from `i` to `j` which are valid character indices. With only a single string argument, this computes the number of characters in the entire string. With `i` and `j` arguments it computes the number of indices between `i` and `j` inclusive that are valid indices in the string `s`. In addition to in-bounds values, `i` may take the out-of-bounds value `ncodeunits(s) + 1` and `j` may take the out-of-bounds value `0`.

See also: `isvalid`, `ncodeunits`, `lastindex`, `thisind`, `nextind`, `prevind`

Examples

```
julia> length("jμΛIα")
5
```

`Base.sizeof` — Method

```
sizeof(str::AbstractString)
```

Size, in bytes, of the string `str`. Equal to the number of code units in `str` multiplied by the size, in bytes, of one code unit in `str`.

Examples

```
julia> sizeof("")
0

julia> sizeof("∀")
3
```

`Base.:*` — Method

```
*(s::Union{AbstractString, AbstractChar}, t::Union{AbstractString, AbstractChar
```

Concatenate strings and/or characters, producing a `String`. This is equivalent to calling the `string` function on the arguments. Concatenation of built-in string types always produces a value of type `String` but other string types may choose to return a string of a different type as appropriate.

Examples

```
julia> "Hello " * "world"
"Hello world"

julia> 'j' * "ulia"
"julia"
```

`Base.:^` — Method

```
^(s::Union{AbstractString,AbstractChar}, n::Integer)
```

Repeat a string or character `n` times. This can also be written as `repeat(s, n)`.

See also: `repeat`

Examples

```
julia> "Test "^3
"Test Test Test "
```

`Base.string` — Function

```
string(n::Integer; base::Integer = 10, pad::Integer = 1)
```

Convert an integer `n` to a string in the given `base`, optionally specifying a number of digits to pad to.

```
julia> string(5, base = 13, pad = 4)
"0005"

julia> string(13, base = 5, pad = 4)
"0023"
```

```
string(xs...)
```

Create a string from any values, except `nothing`, using the `print` function.

`string` should usually not be defined directly. Instead, define a method `print(io::IO, x::MyType)`. If `string(x)` for a certain type needs to be highly efficient, then it may make sense to add a method to `string` and define `print(io::IO, x::MyType) = print(io, string(x))` to ensure the functions are consistent.

Examples

```
julia> string("a", 1, true)
"a1true"
```

`Base.repeat` — Method

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also: `^`

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

`Base.repeat` — Method

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

`Base.repr` — Method

```
repr(x; context=nothing)
```

Create a string from any value using the `show` function. You should not add methods to `repr`; define a `show` method instead.

The optional keyword argument `context` can be set to an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

Note that `repr(x)` is usually similar to how the value of `x` would be entered in Julia. See also `repr(MIME("text/plain"), x)` to instead return a "pretty-printed" version of `x` designed more for human consumption, equivalent to the REPL display of `x`.

Examples

```
julia> repr(1)
"1"

julia> repr(zeros(3))
```

```
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.333333333333333314829616256247390992939472198486328125"

julia> repr(big(1/3), context=:compact => true)
"0.333333"
```

`Core.String` — Method

```
String(s::AbstractString)
```

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

`Base.SubString` — Type

```
SubString(s::AbstractString, i::Integer, j::Integer=lastindex(s))
SubString(s::AbstractString, r::UnitRange{<:Integer})
```

Like `getindex`, but returns a view into the parent string `s` within range `i:j` or `r` respectively instead of making a copy.

Examples

```
julia> SubString("abc", 1, 2)
"ab"

julia> SubString("abc", 1:2)
"ab"

julia> SubString("abc", 2)
"bc"
```

`Base.transcode` — Function

```
transcode(T, src)
```

Convert string data between Unicode encodings. `src` is either a `String` or a `Vector{UIntXX}` of UTF-XX code units, where `XX` is 8, 16, or 32. `T` indicates the encoding of the return value: `String` to return a (UTF-8 encoded) `String` or `UIntXX` to return a `Vector{UIntXX}` of UTF-`XX` data. (The alias `Cwchar_t` can also be used as the integer type, for converting `wchar_t*` strings used by external C libraries.)

The `transcode` function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

`Base.unsafe_string` — Function

```
unsafe_string(p::Ptr{UInt8}, [length::Integer])
```

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labeled "unsafe" because it will crash if `p` is not a valid memory address to data of the requested length.

`Base.ncodeunits` — Method

```
ncodeunits(s::AbstractString) -> Int
```

Return the number of code units in a string. Indices that are in bounds to access this string must satisfy `1 ≤ i ≤ ncodeunits(s)`. Not all such indices are valid – they may not be the start of a character, but they will return a code unit value when calling `codeunit(s,i)`.

Examples

```
julia> ncodeunits("The Julia Language")
18
```

```
julia> ncodeunits("∫eˣ")
6

julia> ncodeunits('∫'), ncodeunits('e'), ncodeunits('ˣ')
(3, 1, 2)
```

See also: `codeunit`, `checkbounds`, `sizeof`, `length`, `lastindex`

`Base.codeunit` — Function

```
codeunit(s::AbstractString) -> Type{<:Union{UInt8, UInt16, UInt32}}
```

Return the code unit type of the given string object. For ASCII, Latin-1, or UTF-8 encoded strings, this would be `UInt8`; for UCS-2 and UTF-16 it would be `UInt16`; for UTF-32 it would be `UInt32`. The unit code type need not be limited to these three types, but it's hard to think of widely used string encodings that don't use one of these units. `codeunit(s)` is the same as `typeof(codeunit(s,1))` when `s` is a non-empty string.

See also: `ncodeunits`

```
codeunit(s::AbstractString, i::Integer) -> Union{UInt8, UInt16, UInt32}
```

Return the code unit value in the string `s` at index `i`. Note that

```
codeunit(s, i) :: codeunit(s)
```

I.e. the value returned by `codeunit(s, i)` is of the type returned by `codeunit(s)`.

Examples

```
julia> a = codeunit("Hello", 2)
0x65

julia> typeof(a)
UInt8
```

See also: `ncodeunits`, `checkbounds`

`Base.codeunits` — Function

```
codeunits(s::AbstractString)
```

Obtain a vector-like object containing the code units of a string. Returns a `CodeUnits` wrapper by default, but `codeunits` may optionally be defined for new string types if necessary.

Examples

```
julia> codeunits("Juλia")
6-element Base.CodeUnits{UInt8,String}:
 0x4a
 0x75
 0xce
 0xbb
 0x69
 0x61
```

`Base.ascii` — Function

```
ascii(s::AbstractString)
```

Convert a string to `String` type and check that it contains only ASCII data, otherwise throwing an `ArgumentError` indicating the position of the first non-ASCII byte.

Examples

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
[...]

julia> ascii("abcdefgh")
"abcdefgh"
```

`Base.@r_str` — Macro

```
@r_str -> Regex
```

Construct a regex, such as `r"^[a-z]*$"`, without interpolation and unescaping (except for quotation mark `"` which still has to be escaped). The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- `i` enables case-insensitive matching
- `m` treats the `^` and `$` tokens as matching the start and end of individual lines, as opposed to the whole string.
- `s` allows the `.` modifier to match newlines.
- `x` enables "comment mode": whitespace is enabled except when escaped with `\`, and `#` is treated as starting a comment.
- `a` disables `UCP` mode (enables ASCII mode). By default `\B`, `\b`, `\D`, `\d`, `\S`, `\s`, `\W`, `\w`, etc. match based on Unicode character properties. With this option, these sequences only match ASCII characters.

See `Regex` if interpolation is needed.

Examples

```
julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

This regex has the first three flags enabled.

`Base.SubstitutionString` — Type

```
SubstitutionString(substr)
```

Stores the given string `substr` as a `SubstitutionString`, for use in regular expression substitutions. Most commonly constructed using the `@s_str` macro.

```
julia> SubstitutionString("Hello \\g<name>, it's \\1")
s"Hello \\g<name>, it's \\1"

julia> subst = s"Hello \g<name>, it's \1"
s"Hello \\g<name>, it's \\1"
```

```
julia> typeof(subst)
SubstitutionString{String}
```

Base.@s_str — Macro

```
@s_str -> SubstitutionString
```

Construct a substitution string, used for regular expression substitutions. Within the string, sequences of the form `\N` refer to the Nth capture group in the regex, and `\g<groupname>` refers to a named capture group with name `groupname`.

```
julia> msg = "#Hello# from Julia";

julia> replace(msg, r"#(.+)# from (?<from>\w+)" => s"FROM: \g<from>; MESSAGE: \
"FROM: Julia; MESSAGE: Hello"
```

Base.@raw_str — Macro

```
@raw_str -> String
```

Create a raw string without interpolation and unescaping. The exception is that quotation marks still must be escaped. Backslashes escape both quotation marks and other backslashes, but only when a sequence of backslashes precedes a quote character. Thus, 2n backslashes followed by a quote encodes n backslashes and the end of the literal while 2n+1 backslashes followed by a quote encodes n backslashes followed by a quote character.

Examples

```
julia> println(raw"\ $x")
\ $x

julia> println(raw"\"")
"

julia> println(raw"\\\"")
\"
```

```
julia> println(raw"\\x \\\"")
\\x \"
```

Base.@b_str — Macro

```
@b_str
```

Create an immutable byte (`UInt8`) vector using string syntax.

Examples

```
julia> v = b"12\x01\x02"
4-element Base.CodeUnits{UInt8,String}:
 0x31
 0x32
 0x01
 0x02

julia> v[2]
0x32
```

Base.Docs.@html_str — Macro

```
@html_str -> Docs.HTML
```

Create an `HTML` object from a literal string.

Base.Docs.@text_str — Macro

```
@text_str -> Docs.Text
```

Create a `Text` object from a literal string.

Base.isvalid — Method

```
isvalid(value) -> Bool
```

Returns `true` if the given value is valid for its type, which currently can be either `AbstractChar` or `String` or `SubString{String}`.

Examples

```
julia> isvalid(Char(0xd800))
false

julia> isvalid(SubString(String(UInt8[0xfe,0x80,0x80,0x80,0x80,0x80]),1,2))
false

julia> isvalid(Char(0xd799))
true
```

`Base.isvalid` — Method

```
isvalid(T, value) -> Bool
```

Returns `true` if the given value is valid for that type. Types currently can be either `AbstractChar` or `String`. Values for `AbstractChar` can be of type `AbstractChar` or `UInt32`. Values for `String` can be of that type, or `Vector{UInt8}` or `SubString{String}`.

Examples

```
julia> isvalid(Char, 0xd800)
false

julia> isvalid(String, SubString("thisisvalid",1,5))
true

julia> isvalid(Char, 0xd799)
true
```

`Base.isvalid` — Method

```
isvalid(s::AbstractString, i::Integer) -> Bool
```

Predicate indicating whether the given index is the start of the encoding of a character in `s` or not. If `isvalid(s, i)` is true then `s[i]` will return the character whose encoding starts at that index, if it's false, then `s[i]` will raise an invalid index error or a bounds error depending on if `i` is in bounds. In order for `isvalid(s, i)` to be an O(1) function, the encoding of `s` must be self-synchronizing this is a basic assumption of Julia's generic string support.

See also: `getindex`, `iterate`, `thisind`, `nextind`, `prevind`, `length`

Examples

```
julia> str = "αβγdef";

julia> isvalid(str, 1)
true

julia> str[1]
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> isvalid(str, 2)
false

julia> str[2]
ERROR: StringIndexError("αβγdef", 2)
Stacktrace:
[...]
```

`Base.match` — Function

```
match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression `r` in `s` and return a `RegexMatch` object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing `m.match` and the captured sequences can be retrieved by accessing `m.captures` The optional `idx` argument specifies an index at which to start the search.

Examples

```
julia> rx = r"a(.)a"
r"a(.)a"

julia> m = match(rx, "cabac")
RegexMatch("aba", 1="b")

julia> m.captures
1-element Array{Union{Nothing, SubString{String}},1}:
 "b"

julia> m.match
"aba"

julia> match(rx, "cabac", 3) === nothing
true
```

`Base.eachmatch` — Function

```
eachmatch(r::Regex, s::AbstractString; overlap::Bool=false)
```

Search for all matches of a the regular expression `r` in `s` and return a iterator over the matches. If overlap is `true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

Examples

```
julia> rx = r"a.a"
r"a.a"

julia> m = eachmatch(rx, "a1a2a3a")
Base.RegexMatchIterator(r"a.a", "a1a2a3a", false)

julia> collect(m)
2-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a3a")

julia> collect(eachmatch(rx, "a1a2a3a", overlap = true))
3-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a2a")
RegexMatch("a3a")
```

`Base.isless` — Method

```
isless(a::AbstractString, b::AbstractString) -> Bool
```

Test whether string `a` comes before string `b` in alphabetical order (technically, in lexicographical order by Unicode code points).

Examples

```
julia> isless("a", "b")
true

julia> isless("β", "α")
false

julia> isless("a", "a")
false
```

`Base.:==` — Method

```
==(a::AbstractString, b::AbstractString) -> Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point).

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

`Base.cmp` — Method

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return `0` if both strings have the same length and the character at each index is the same in both strings. Return `-1` if `a` is a prefix of `b`, or if `a` comes before `b` in alphabetical order. Return `1` if `b` is a prefix of `a`, or if `b` comes before `a` in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

`Base.lpad` — Function

```
lpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify `s` and pad the resulting string on the left with `p` to make it `n` characters (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> lpad("March", 10)
"     March"
```

`Base.rpad` — Function

```
rpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify `s` and pad the resulting string on the right with `p` to make it `n` characters (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> rpad("March", 20)
"March               "
```

`Base.findfirst` — Method

```
findfirst(pattern::AbstractString, string::AbstractString)
findfirst(pattern::Regex, string::String)
```

Find the first occurrence of `pattern` in `string`. Equivalent to `findnext(pattern, string, firstindex(s))`.

Examples

```
julia> findfirst("z", "Hello to the world") # returns nothing, but not printed

julia> findfirst("Julia", "JuliaLang")
1:5
```

`Base.findnext` — Method

```
findnext(pattern::AbstractString, string::AbstractString, start::Integer)
findnext(pattern::Regex, string::String, start::Integer)
```

Find the next occurrence of `pattern` in `string` starting at position `start`. `pattern` can be either a string, or a regular expression, in which case `string` must be of type `String`.

The return value is a range of indices where the matching sequence is found, such that `s[findnext(x, s, i)] == x`:

`findnext("substring", string, i) == start:stop` such that `string[start:stop] == "substring"` and `i <= start`, or `nothing` if unmatched.

Examples

```
julia> findnext("z", "Hello to the world", 1) === nothing
true

julia> findnext("o", "Hello to the world", 6)
8:8

julia> findnext("Lang", "JuliaLang", 2)
6:9
```

`Base.findnext` — Method

```
findnext(ch::AbstractChar, string::AbstractString, start::Integer)
```

Find the next occurrence of character `ch` in `string` starting at position `start`.

> **Julia 1.3**
>
> This method requires at least Julia 1.3.

Examples

```
julia> findnext('z', "Hello to the world", 1) === nothing
true

julia> findnext('o', "Hello to the world", 6)
8
```

`Base.findlast` — Method

```
findlast(pattern::AbstractString, string::AbstractString)
```

Find the last occurrence of `pattern` in `string`. Equivalent to `findprev(pattern, string, lastindex(string))`.

Examples

```
julia> findlast("o", "Hello to the world")
15:15

julia> findfirst("Julia", "JuliaLang")
1:5
```

`Base.findlast` — Method

```
findlast(ch::AbstractChar, string::AbstractString)
```

Find the last occurrence of character `ch` in `string`.

> ❗ Julia 1.3
>
> This method requires at least Julia 1.3.

Examples

```
julia> findlast('p', "happy")
4

julia> findlast('z', "happy") === nothing
true
```

`Base.findprev` — Method

```
findprev(pattern::AbstractString, string::AbstractString, start::Integer)
```

Find the previous occurrence of `pattern` in `string` starting at position `start`.

The return value is a range of indices where the matching sequence is found, such that `s[findprev(x, s, i)] == x`:

`findprev("substring", string, i)` == `start:stop` such that `string[start:stop] == "substring"` and `stop <= i`, or `nothing` if unmatched.

Examples

```
julia> findprev("z", "Hello to the world", 18) === nothing
true

julia> findprev("o", "Hello to the world", 18)
15:15

julia> findprev("Julia", "JuliaLang", 6)
1:5
```

Base.occursin — Function

```
occursin(needle::Union{AbstractString,Regex,AbstractChar}, haystack::AbstractSt
```

Determine whether the first argument is a substring of the second. If `needle` is a regular expression, checks whether `haystack` contains a match.

Examples

```
julia> occursin("Julia", "JuliaLang is pretty cool!")
true

julia> occursin('a', "JuliaLang is pretty cool!")
true

julia> occursin(r"a.a", "aba")
true

julia> occursin(r"a.a", "abba")
```

```
false
```

See also: contains.

Base.reverse — Method

```
reverse(s::AbstractString) -> AbstractString
```

Reverses a string. Technically, this function reverses the codepoints in a string and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also reverseind to convert indices in s to indices in reverse(s) and vice-versa, and graphemes from module Unicode to operate on user-visible "characters" (graphemes) rather than codepoints. See also Iterators.reverse for reverse-order iteration without making a copy. Custom string types must implement the reverse function themselves and should typically return a string with the same type and encoding. If they return a string with a different encoding, they must also override reverseind for that string type to satisfy s[reverseind(s,i)] == reverse(s)[i].

Examples

```
julia> reverse("JuliaLang")
"gnaLailuJ"

julia> reverse("ax̂e") # combining characters can lead to surprising results
"êxa"

julia> using Unicode

julia> join(reverse(collect(graphemes("ax̂e")))) # reverses graphemes
"ex̂a"
```

Base.replace — Method

```
replace(s::AbstractString, pat=>r; [count::Integer])
```

Search for the given pattern pat in s, and replace each occurrence with r. If count is provided, replace at most count occurrences. pat may be a single character, a vector or a set of characters, a string, or a regular expression. If `r` is a function, each occurrence is replaced with `r(s)` where `s` is the matched substring (when `pat` is a `Regex` or `AbstractString`) or character (when `pat` is an `AbstractChar` or a collection of `AbstractChar`). If `pat` is a regular expression and `r` is a `SubstitutionString`, then capture group references in `r` are replaced with the corresponding matched text. To remove instances of `pat` from `string`, set `r` to the empty `String` (`""`).

Examples

```
julia> replace("Python is a programming language.", "Python" => "Julia")
"Julia is a programming language."

julia> replace("The quick foxes run quickly.", "quick" => "slow", count=1)
"The slow foxes run quickly."

julia> replace("The quick foxes run quickly.", "quick" => "", count=1)
"The  foxes run quickly."

julia> replace("The quick foxes run quickly.", r"fox(es)?" => s"bus\1")
"The quick buses run quickly."
```

`Base.split` — Function

```
split(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
split(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` into an array of substrings on occurrences of the delimiter(s) `dlm`. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also `rsplit`.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
2-element Array{SubString{String},1}:
 "Ma"
 "rch"
```

`Base.rsplit` — Function

```
rsplit(s::AbstractString; limit::Integer=0, keepempty::Bool=false)
rsplit(s::AbstractString, chars; limit::Integer=0, keepempty::Bool=true)
```

Similar to `split`, but starting from the end of the string.

Examples

```
julia> a = "M.a.r.c.h"
"M.a.r.c.h"

julia> rsplit(a, ".")
5-element Array{SubString{String},1}:
 "M"
 "a"
 "r"
 "c"
 "h"

julia> rsplit(a, "."; limit=1)
1-element Array{SubString{String},1}:
 "M.a.r.c.h"

julia> rsplit(a, "."; limit=2)
2-element Array{SubString{String},1}:
 "M.a.r.c"
 "h"
```

`Base.strip` — Function

```
strip([pred=isspace,] str::AbstractString) -> SubString
strip(str::AbstractString, chars) -> SubString
```

Remove leading and trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, vector or set of characters.

> Julia 1.2
>
> The method which accepts a predicate function requires Julia 1.2 or later.

Examples

```
julia> strip("{3, 5}\n", ['{', '}', '\n'])
"3, 5"
```

`Base.lstrip` — Function

```
lstrip([pred=isspace,] str::AbstractString) -> SubString
lstrip(str::AbstractString, chars) -> SubString
```

Remove leading characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

Examples

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

`Base.rstrip` — Function

```
rstrip([pred=isspace,] str::AbstractString) -> SubString
rstrip(str::AbstractString, chars) -> SubString
```

Remove trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

Examples

```
julia> a = rpad("March", 20)
"March               "

julia> rstrip(a)
"March"
```

`Base.startswith` — Function

```
startswith(s::AbstractString, prefix::AbstractString)
```

Return `true` if `s` starts with `prefix`. If `prefix` is a vector or set of characters, test whether the first character of `s` belongs to that set.

See also `endswith`.

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

```
startswith(prefix)
```

Create a function that checks whether its argument starts with `prefix`, i.e. a function equivalent to `y -> startswith(y, prefix)`.

The returned function is of type `Base.Fix2{typeof(startswith)}`, which can be used to implement specialized methods.

> Julia 1.5
>
> The single argument `startswith(prefix)` requires at least Julia 1.5.

```
startswith(s::AbstractString, prefix::Regex)
```

Return `true` if `s` starts with the regex pattern, `prefix`.

> Note
>
> `startswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"^...", s)` is faster than `startswith(s, r"...")`.

See also `occursin` and `endswith`.

> Julia 1.2
>
> This method requires at least Julia 1.2.

Examples

```
julia> startswith("JuliaLang", r"Julia|Romeo")
true
```

`Base.endswith` — Function

```
endswith(s::AbstractString, suffix::AbstractString)
```

Return `true` if `s` ends with `suffix`. If `suffix` is a vector or set of characters, test whether the last character of `s` belongs to that set.

See also `startswith`.

Examples

```
julia> endswith("Sunday", "day")
true
```

```
endswith(suffix)
```

Create a function that checks whether its argument ends with `suffix`, i.e. a function equivalent to `y -> endswith(y, suffix)`.

The returned function is of type `Base.Fix2{typeof(endswith)}`, which can be used to implement specialized methods.

> Julia 1.5
>
> The single argument `endswith(suffix)` requires at least Julia 1.5.

```
endswith(s::AbstractString, suffix::Regex)
```

Return `true` if `s` ends with the regex pattern, `suffix`.

> Note

`endswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"...$", s)` is faster than `endswith(s, r"...")`.

See also `occursin` and `startswith`.

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> endswith("JuliaLang", r"Lang|Roberts")
true
```

`Base.contains` — Function

```
contains(haystack::AbstractString, needle)
```

Return `true` if `haystack` contains `needle`. This is the same as `occursin(needle, haystack)`, but is provided for consistency with `startswith(haystack, needle)` and `endswith(haystack, needle)`.

Examples

```
julia> contains("JuliaLang is pretty cool!", "Julia")
true

julia> contains("JuliaLang is pretty cool!", 'a')
true

julia> contains("aba", r"a.a")
true

julia> contains("abba", r"a.a")
false
```

> ❗ Julia 1.5
>
> The `contains` function requires at least Julia 1.5.

```
contains(needle)
```

Create a function that checks whether its argument contains `needle`, i.e. a function equivalent to `haystack -> contains(haystack, needle)`.

The returned function is of type `Base.Fix2{typeof(contains)}`, which can be used to implement specialized methods.

`Base.first` — Method

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first `n` characters of `s`.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

`Base.last` — Method

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last `n` characters of `s`.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

`Base.Unicode.uppercase` — Function

```
uppercase(s::AbstractString)
```

Return `s` with all characters converted to uppercase.

Examples

```
julia> uppercase("Julia")
"JULIA"
```

`Base.Unicode.lowercase` — Function

```
lowercase(s::AbstractString)
```

Return `s` with all characters converted to lowercase.

Examples

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

`Base.Unicode.titlecase` — Function

```
titlecase(s::AbstractString; [wordsep::Function], strict::Bool=true) -> String
```

Capitalize the first character of each word in `s`; if `strict` is true, every other character is converted to lowercase, otherwise they are left unchanged. By default, all non-letters are considered as word separators; a predicate can be passed as the `wordsep` keyword to determine which characters should be considered as word separators. See also `uppercasefirst` to capitalize only the first character in `s`.

Examples

```
julia> titlecase("the JULIA programming language")
"The Julia Programming Language"

julia> titlecase("ISS - international space station", strict=false)
"ISS - International Space Station"

julia> titlecase("a-a b-b", wordsep = c->c==' ')
"A-a B-b"
```

`Base.Unicode.uppercasefirst` — Function

```
uppercasefirst(s::AbstractString) -> String
```

Return `s` with the first character converted to uppercase (technically "title case" for Unicode). See also `titlecase` to capitalize the first character of every word in `s`.

See also: `lowercasefirst`, `uppercase`, `lowercase`, `titlecase`

Examples

```
julia> uppercasefirst("python")
"Python"
```

`Base.Unicode.lowercasefirst` — Function

```
lowercasefirst(s::AbstractString)
```

Return `s` with the first character converted to lowercase.

See also: `uppercasefirst`, `uppercase`, `lowercase`, `titlecase`

Examples

```
julia> lowercasefirst("Julia")
"julia"
```

`Base.join` — Function

```
join([io::IO,] strings [, delim [, last]])
```

Join an array of `strings` into a single string, inserting the given delimiter (if any) between adjacent strings. If `last` is given, it will be used instead of `delim` between the last two strings. If `io` is given, the result is written to `io` rather than returned as as a `String`.

`strings` can be any iterable over elements `x` which are convertible to strings via `print(io::IOBuffer, x)`. `strings` will be printed to `io`.

Examples

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"

julia> join([1,2,3,4,5])
"12345"
```

`Base.chop` — Function

```
chop(s::AbstractString; head::Integer = 0, tail::Integer = 1)
```

Remove the first `head` and the last `tail` characters from `s`. The call `chop(s)` removes the last character from `s`. If it is requested to remove more characters than `length(s)` then an empty string is returned.

Examples

```
julia> a = "March"
"March"
```

```
julia> chop(a)
"Marc"

julia> chop(a, head = 1, tail = 2)
"ar"

julia> chop(a, head = 5, tail = 5)
""
```

Base.chomp — Function

```
chomp(s::AbstractString) -> SubString
```

Remove a single trailing newline from a string.

Examples

```
julia> chomp("Hello\n")
"Hello"
```

Base.thisind — Function

```
thisind(s::AbstractString, i::Integer) -> Int
```

If `i` is in bounds in `s` return the index of the start of the character whose encoding code unit `i` is part of. In other words, if `i` is the start of a character, return `i`; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to 0 or `ncodeunits(s)+1` return `i`. In all other cases throw `BoundsError`.

Examples

```
julia> thisind("α", 0)
0

julia> thisind("α", 1)
1

julia> thisind("α", 2)
```

```
1

julia> thisind("α", 3)
3

julia> thisind("α", 4)
ERROR: BoundsError: attempt to access String
  at index [4]
[...]

julia> thisind("α", -1)
ERROR: BoundsError: attempt to access String
  at index [-1]
[...]
```

`Base.nextind` — Function

```
nextind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts after index `i`. In other words, if `i` is the start of a character, return the start of the next character; if `i` is not the start of a character, move forward until the start of a character and return that index. If `i` is equal to `0` return `1`. If `i` is in bounds but greater or equal to `lastindex(str)` return `ncodeunits(str)+1`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `nextind` for `n==1`. The only difference is that if `n` is so large that applying `nextind` would reach `ncodeunits(str)+1` then each remaining iteration increases the returned value by `1`. This means that in this case `nextind` can return a value greater than `ncodeunits(str)+1`.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `s` or is equal to `0`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> nextind("α", 0)
```

```
1

julia> nextind("α", 1)
3

julia> nextind("α", 3)
ERROR: BoundsError: attempt to access String
  at index [3]
[...]

julia> nextind("α", 0, 2)
3

julia> nextind("α", 1, 2)
4
```

`Base.prevind` — Function

```
prevind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts before index `i`. In other words, if `i` is the start of a character, return the start of the previous character; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to `1` return `0`. If `i` is equal to `ncodeunits(str)+1` return `lastindex(str)`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `prevind` for `n==1`. The only difference is that if `n` is so large that applying `prevind` would reach `0` then each remaining iteration decreases the returned value by `1`. This means that in this case `prevind` can return a negative value.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `str` or is equal to `ncodeunits(str)+1`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> prevind("α", 3)
```

```
1

julia> prevind("α", 1)
0

julia> prevind("α", 0)
ERROR: BoundsError: attempt to access String
  at index [0]
[...]

julia> prevind("α", 2, 2)
0

julia> prevind("α", 2, 3)
-1
```

`Base.Unicode.textwidth` — Function

```
textwidth(c)
```

Give the number of columns needed to print a character.

Examples

```
julia> textwidth('α')
1

julia> textwidth('⛵')
2
```

```
textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

```
julia> textwidth("March")
5
```

`Base.isascii` — Function

```
isascii(c::Union{AbstractChar,AbstractString}) -> Bool
```

Test whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

Examples

```
julia> isascii('a')
true

julia> isascii('α')
false

julia> isascii("abc")
true

julia> isascii("αβγ")
false
```

`Base.Unicode.iscntrl` — Function

```
iscntrl(c::AbstractChar) -> Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

Examples

```
julia> iscntrl('\x01')
true

julia> iscntrl('a')
false
```

`Base.Unicode.isdigit` — Function

```
isdigit(c::AbstractChar) -> Bool
```

Tests whether a character is a decimal digit (0-9).

Examples

```
julia> isdigit('❤')
false

julia> isdigit('9')
true

julia> isdigit('α')
false
```

`Base.Unicode.isletter` — Function

```
isletter(c::AbstractChar) -> Bool
```

Test whether a character is a letter. A character is classified as a letter if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

Examples

```
julia> isletter('❤')
false

julia> isletter('α')
true

julia> isletter('9')
false
```

`Base.Unicode.islowercase` — Function

```
islowercase(c::AbstractChar) -> Bool
```

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

Examples

```
julia> islowercase('α')
true

julia> islowercase('Γ')
false

julia> islowercase('❤')
false
```

Base.Unicode.isnumeric — Function

```
isnumeric(c::AbstractChar) -> Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

Note that this broad category includes characters such as ¾ and ௰. Use `isdigit` to check whether a character a decimal digit between 0 and 9.

Examples

```
julia> isnumeric('௰')
true

julia> isnumeric('9')
true

julia> isnumeric('α')
false

julia> isnumeric('❤')
false
```

Base.Unicode.isprint — Function

```
isprint(c::AbstractChar) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

Examples

```
julia> isprint('\x01')
false

julia> isprint('A')
true
```

`Base.Unicode.ispunct` — Function

```
ispunct(c::AbstractChar) -> Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

Examples

```
julia> ispunct('α')
false

julia> ispunct('/')
true

julia> ispunct(';')
true
```

`Base.Unicode.isspace` — Function

```
isspace(c::AbstractChar) -> Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

Examples

```
julia> isspace('\n')
true

julia> isspace('\r')
true

julia> isspace(' ')
true

julia> isspace('\x20')
true
```

`Base.Unicode.isuppercase` — Function

```
isuppercase(c::AbstractChar) -> Bool
```

Tests whether a character is an uppercase letter. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

Examples

```
julia> isuppercase('γ')
false

julia> isuppercase('Γ')
true

julia> isuppercase('❤')
false
```

`Base.Unicode.isxdigit` — Function

```
isxdigit(c::AbstractChar) -> Bool
```

Test whether a character is a valid hexadecimal digit. Note that this does not include `x` (as in the standard `0x` prefix).

Examples

```
julia> isxdigit('a')
true

julia> isxdigit('x')
false
```

`Base.escape_string` — Function

```
escape_string(str::AbstractString[, esc])::AbstractString
escape_string(io, str::AbstractString[, esc::])::Nothing
```

General escaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to `io`.

Backslashes (`\`) are escaped with a double-backslash (`"\\"`). Non-printable characters are escaped either with their standard C escape codes, `"\0"` for NUL (if unambiguous), unicode code point (`"\u"` prefix) or hex (`"\x"` prefix).

The optional `esc` argument specifies any additional characters that should also be escaped by a prepending backslash (`"` is also escaped by default in the first form).

Examples

```
julia> escape_string("aaa\nbbb")
"aaa\\nbbb"

julia> escape_string("\xfe\xff") # invalid utf-8
"\\xfe\\xff"

julia> escape_string(string('\u2135','\0')) # unambiguous
"ℵ\\0"

julia> escape_string(string('\u2135','\0','0')) # \0 would be ambiguous
"ℵ\\x000"
```

See also

`unescape_string` for the reverse operation.

`Base.unescape_string` — Function

```
unescape_string(str::AbstractString, keep = ())::AbstractString
unescape_string(io, s::AbstractString, keep = ())::Nothing
```

General unescaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to `io`. The argument `keep` specifies a collection of characters which (along with backlashes) are to be kept as they are.

The following escape sequences are recognised:

- Escaped backslash (`\\`)
- Escaped double-quote (`\"`)
- Standard C escape sequences (`\a`, `\b`, `\t`, `\n`, `\v`, `\f`, `\r`, `\e`)
- Unicode BMP code points (`\u` with 1-4 trailing hex digits)
- All Unicode code points (`\U` with 1-8 trailing hex digits; max value = 0010ffff)
- Hex bytes (`\x` with 1-2 trailing hex digits)
- Octal bytes (`\` with 1-3 trailing octal digits)

Examples

```
julia> unescape_string("aaa\\nbbb") # C escape sequence
"aaa\nbbb"

julia> unescape_string("\\u03c0") # unicode
"π"

julia> unescape_string("\\101") # octal
"A"

julia> unescape_string("aaa \\g \\n", ['g']) # using `keep` argument
"aaa \\g \n"
```

See also

`escape_string`.

« Numbers Arrays »

Powered by Documenter.jl and the Julia Programming Language.