str

A sequence of Unicode codepoints.

You can iterate over the grapheme clusters of the string using a for loop. Grapheme clusters are basically characters but keep together things that belong together, e.g. multiple codepoints that together form a flag emoji. Strings can be added with the + operator, joined together and multiplied with integers.

Typst provides utility methods for string manipulation. Many of these methods (e.g., `split`, `trim` and `replace`) operate on *patterns:* A pattern can be either a string or a regular expression. This makes the methods quite versatile.

All lengths and indices are expressed in terms of UTF-8 bytes. Indices are zero-based and negative indices wrap around to the end of the string.

You can convert a value to a string with this type's constructor.

Example

```
#"hello world!" \
#"\"hello\n  world\"!" \
#"1 2 3".split() \
#"1,2;3".split(regex("[,;]")) \
#(regex("\d+") in "ten euros") \
#(regex("\d+") in "10 euros")
```

hello world!
"hello
 world"!
("1", "2", "3")

```
("1", "2", "3")
false
true
```

Escape sequences

Just like in markup, you can escape a few symbols in strings:

- `\\` for a backslash
- `\"` for a quote
- `\n` for a newline
- `\r` for a carriage return
- `\t` for a tab
- `\u{1f600}` for a hexadecimal Unicode escape sequence

Constructor

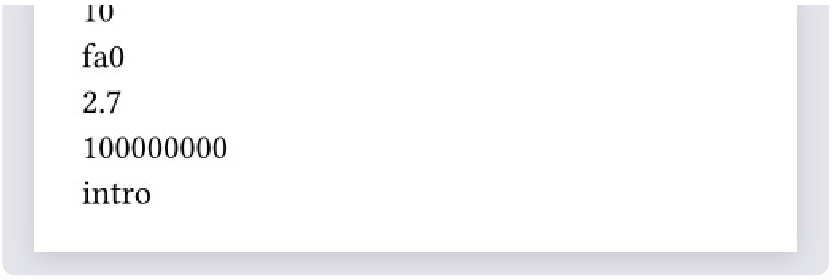
Converts a value to a string.

- Integers are formatted in base 10. This can be overridden with the optional `base` parameter.
- Floats are formatted in base 10 and never in exponential notation.
- From labels the name is extracted.
- Bytes are decoded as UTF-8.

If you wish to convert from and to Unicode code points, see the `to-unicode` and `from-unicode` functions.

```
str(
  int float str bytes label version type ,
  base: int ,
) -> str
```

```
#str(10) \
#str(4000, base: 16) \
#str(2.7) \
#str(1e8) \
#str(<intro>)
```


10
fa0
2.7
100000000
intro

`value`

`int` or `float` or `str` or `bytes` or `label` or `version` or `type` *Required* *Positional*

The value that should be converted to a string.

`base` `int`

The base (radix) to display integers in, between 2 and 36.

Default: `10`

Definitions

`len`

The length of the string in UTF-8 encoded bytes.

```
self.len() -> int
```

`first`

Extracts the first grapheme cluster of the string. Fails with an error if the string is empty.

```
self.first() -> str
```

`last`

Extracts the last grapheme cluster of the string. Fails with an error if the string is empty.

```
self.last() -> str
```

at

Extracts the first grapheme cluster after the specified index. Returns the default value if the index is out of bounds or fails with an error if no default value was specified.

```
self.at(
  int,
  default: any,
) -> any
```

index `int` *Required* *Positional*

The byte index. If negative, indexes from the back.

default `any`

A default value to return if the index is out of bounds.

slice

Extracts a substring of the string. Fails with an error if the start or end index is out of bounds.

```
self.slice(
  int,
  none int,
  count: int,
) -> str
```

start `int` *Required* *Positional*

The start byte index (inclusive). If negative, indexes from the back.

end `none` or `int` *Positional*

The end byte index (exclusive). If omitted, the whole slice until the end of the string is extracted. If negative, indexes from the back.

Default: none

count `int`

The number of bytes to extract. This is equivalent to passing `start + count` as the `end` position. Mutually exclusive with `end`.

`clusters`

Returns the grapheme clusters of the string as an array of substrings.

```
self.clusters() -> array
```

`codepoints`

Returns the Unicode codepoints of the string as an array of substrings.

```
self.codepoints() -> array
```

`to-unicode`

Converts a character into its corresponding code point.

```
str.to-unicode( str ) -> int
```

```
#"a".to-unicode() \
#("a\u{0300}"
    .codepoints()
    .map(str.to-unicode))
```

```
97
(97, 768)
```

`character` `str` *Required* *Positional*

The character that should be converted.

`from-unicode`

Converts a unicode code point into its corresponding string.

```
str.from-unicode( int ) -> str
```

```
#str.from-unicode(97)
```

```
a
```

`value` `int` *Required* *Positional*

The code point that should be converted.

`contains`

Whether the string contains the specified pattern.

This method also has dedicated syntax: You can write `"bc" in "abcd"` instead of `"abcd".contains("bc")`.

```
self.contains( str regex ) -> bool
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

`starts-with`

Whether the string starts with the specified pattern.

```
self.starts-with( str regex ) -> bool
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern the string might start with.

`ends-with`

Whether the string ends with the specified pattern.

```
self.ends-with( str regex ) -> bool
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern the string might end with.

`find`

Searches for the specified pattern in the string and returns the first match as a string or `none` if there is no match.

```
self.find( str regex ) -> none str
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

`position`

Searches for the specified pattern in the string and returns the index of the first match as an integer or `none` if there is no match.

```
self.position( str regex ) -> none int
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

`match`

Searches for the specified pattern in the string and returns a dictionary with details about the first match or `none` if there is no match.

The returned dictionary has the following keys:

- `start`: The start offset of the match

- `end`: The end offset of the match
- `text`: The text that matched.
- `captures`: An array containing a string for each matched capturing group. The first item of the array contains the first matched capturing, not the whole match! This is empty unless the `pattern` was a regex with capturing groups.

```
self.match( str regex ) -> none dictionary
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

`matches`

Searches for the specified pattern in the string and returns an array of dictionaries with details about all matches. For details about the returned dictionaries, see above.

```
self.matches( str regex ) -> array
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

`replace`

Replace at most `count` occurrences of the given pattern with a replacement string or function (beginning from the start). If no count is given, all occurrences are replaced.

```
self.replace(
  str regex ,
  str function ,
  count: int ,
) -> str
```

`pattern` `str` or `regex` *Required* *Positional*

The pattern to search for.

replacement `str` or `function` *Required* *Positional*

The string to replace the matches with or a function that gets a dictionary for each match and can return individual replacement strings.

count `int`

If given, only the first **count** matches of the pattern are placed.

trim

Removes matches of a pattern from one or both sides of the string, once or repeatedly and returns the resulting string.

```
self.trim(
  none str regex ,
  at: alignment ,
  repeat: bool ,
) -> str
```

pattern `none` or `str` or `regex` *Positional*

The pattern to search for. If `none`, trims white spaces.

Default: `none`

at `alignment`

Can be **start** or **end** to only trim the start or end of the string. If omitted, both sides are trimmed.

repeat `bool`

Whether to repeatedly removes matches of the pattern or just once. Defaults to `true`.

Default: `true`

split

Splits a string at matches of a specified pattern and returns an array of the resulting parts.

```
self.split( none  str  regex ) -> array
```

`pattern` `none` or `str` or `regex` *Positional*

The pattern to split at. Defaults to whitespace.

Default: none

rev

Reverse the string.

```
self.rev() -> str
```

Selector
Previous page

Style
Next page