GDScript format strings

GDScript offers a feature called *format strings*, which allows reusing text templates to succinctly create different but similar strings.

Format strings are just like normal strings, except they contain certain placeholder character-sequences. These placeholders can then easily be replaced by parameters handed to the format string.

As an example, with `%s` as a placeholder, the format string `"Hello %s, how are you?"` can easily be changed to `"Hello World, how are you?"`. Notice the placeholder is in the middle of the string; modifying it without format strings could be cumbersome.

Usage in GDScript

Examine this concrete GDScript example:

```
# Define a format string with placeholder '%s'
var format_string = "We're waiting for %s."

# Using the '%' operator, the placeholder is replaced with the desired value
var actual_string = format_string % "Godot"

print(actual_string)
# Output: "We're waiting for Godot."
```

Placeholders always start with a `%`, but the next character or characters, the *format specifier*, determines how the given value is converted to a string.

The `%s` seen in the example above is the simplest placeholder and works for most use cases: it converts the value by the same method by which an implicit String conversion or `str()` would convert it. Strings remain unchanged, Booleans turn into either `"True"` or `"False"`, an integral or real number becomes a decimal, other types usually return their data in a human-readable string.

There is also another way to format text in GDScript, namely the `String.format()` method. It replaces all occurrences of a key in the string with the corresponding value. The method can handle arrays or dictionaries for the key/value pairs.

Arrays can be used as key, index, or mixed style (see below examples). Order only matters when the index or mixed style of Array is used.

A quick example in GDScript:

```
# Define a format string
var format_string = "We're waiting for {str}"

# Using the 'format' method, replace the 'str' placeholder
var actual_string = format_string.format({"str": "Godot"})

print(actual_string)
# Output: "We're waiting for Godot"
```

There are other format specifiers, but they are only applicable when using the `%` operator.

Multiple placeholders

Format strings may contain multiple placeholders. In such a case, the values are handed in the form of an array, one value per placeholder (unless using a format specifier with `*`, see dynamic padding):

```
var format_string = "%s was reluctant to learn %s, but now he enjoys it."
var actual_string = format_string % ["Estragon", "GDScript"]

print(actual_string)
# Output: "Estragon was reluctant to learn GDScript, but now he enjoys it."
```

Note the values are inserted in order. Remember all placeholders must be replaced at once, so there must be an appropriate number of values.

Format specifiers

There are format specifiers other than `s` that can be used in placeholders. They consist of one or more characters. Some of them work by themselves like `s`, some appear before other characters, some only work with certain values or characters.

Placeholder types

One and only one of these must always appear as the last character in a format specifier. Apart from `s`, these require certain types of parameters.

`s`	**Simple** conversion to String by the same method as implicit String conversion.
`c`	A single **Unicode character**. Expects an unsigned 8-bit integer (0-255) for a code point or a single-character string.
`d`	A **decimal integral** number. Expects an integral or real number (will be floored).
`o`	An **octal integral** number. Expects an integral or real number (will be floored).
`x`	A **hexadecimal integral** number with **lower-case** letters. Expects an integral or real number (will be floored).
`X`	A **hexadecimal integral** number with **upper-case** letters. Expects an integral or real number (will be floored).
`f`	A **decimal real** number. Expects an integral or real number.

Placeholder modifiers

These characters appear before the above. Some of them work only under certain conditions.

`+`	In number specifiers, **show + sign** if positive.
Integer	Set **padding**. Padded with spaces or with zeroes if integer starts with `0` in an integer placeholder. When used after `.`, see `.`.
`.`	Before `f`, set **precision** to 0 decimal places. Can be followed up with numbers to change. Padded with zeroes.
`-`	**Pad to the right** rather than the left.
`*`	**Dynamic padding**, expect additional integral parameter to set padding or precision after `.`, see dynamic padding.

Padding

The `.` (*dot*), `*` (*asterisk*), `-` (*minus sign*) and digit (`0` - `9`) characters are used for padding. This allows printing several values aligned vertically as if in a column, provided a fixed-width font is used.

To pad a string to a minimum length, add an integer to the specifier:

```
print("%10d" % 12345)
# output: "     12345"
# 5 leading spaces for a total length of 10
```

If the integer starts with `0`, integral values are padded with zeroes instead of white space:

```
print("%010d" % 12345)
# output: "0000012345"
```

Precision can be specified for real numbers by adding a `.` (*dot*) with an integer following it. With no integer after `.`, a precision of 0 is used, rounding to integral value. The integer to use for padding must appear before the dot.

```
# Pad to minimum length of 10, round to 3 decimal places
print("%10.3f" % 10000.5555)
# Output: " 10000.556"
# 1 leading space
```

The `-` character will cause padding to the right rather than the left, useful for right text alignment:

```
print("%-10d" % 12345678)
# Output: "12345678  "
# 2 trailing spaces
```

Dynamic padding

By using the `*` (*asterisk*) character, the padding or precision can be set without modifying the format string. It is used in place of an integer in the format specifier. The values for padding and precision are then passed when formatting:

```
var format_string = "%*.*f"
# Pad to length of 7, round to 3 decimal places:
print(format_string % [7, 3, 8.8888])
# Output: "  8.889"
# 2 leading spaces
```

It is still possible to pad with zeroes in integer placeholders by adding `0` before `*`:

```
print("%0*d" % [2, 3])
# Output: "03"
```

Escape sequence

To insert a literal `%` character into a format string, it must be escaped to avoid reading it as a placeholder. This is done by doubling the character:

```
var health = 56
print("Remaining health: %d%%" % health)
# Output: "Remaining health: 56%"
```

Format method examples

The following are some examples of how to use the various invocations of the `String.format` method.

Type	Style	Example	Result
Dictionary	key	`"Hi, {name} v{version}!".format({"name":"Godette", "version":"3.0"})`	Hi, Godette v3.0!
Dictionary	index	`"Hi, {0} v{1}!".format({"0":"Godette", "1":"3.0"})`	Hi, Godette v3.0!

Dictionary	mix	`"Hi, {0} v{version}!".format({"0":"Godette", "version":"3.0"})`	Hi, Godette v3.0!
Array	key	`"Hi, {name} v{version}!".format([["version","3.0"], ["name","Godette"]])`	Hi, Godette v3.0!
Array	index	`"Hi, {0} v{1}!".format(["Godette","3.0"])`	Hi, Godette v3.0!
Array	mix	`"Hi, {name} v{0}!".format([3.0, ["name","Godette"]])`	Hi, Godette v3.0!
Array	no index	`"Hi, {} v{}!".format(["Godette", 3.0], "{}")`	Hi, Godette v3.0!

Placeholders can also be customized when using `String.format`, here's some examples of that functionality.

Type	**Example**	**Result**
Infix (default)	`"Hi, {0} v{1}".format(["Godette", "3.0"], "{_}")`	Hi, Godette v3.0
Postfix	`"Hi, 0% v1%".format(["Godette", "3.0"], "_%")`	Hi, Godette v3.0
Prefix	`"Hi, %0 v%1".format(["Godette", "3.0"], "%_")`	Hi, Godette v3.0

Combining both the `String.format` method and the `%` operator could be useful, as `String.format` does not have a way to manipulate the representation of numbers.

Example	**Result**
`"Hi, {0} v{version}".format({0:"Godette", "version":"%0.2f" % 3.114})`	Hi, Godette v3.11