GDScript

See also

See GDScript grammar if you are interested in writing a third-party tool that interacts with GDScript, such as a linter or formatter.

GDScript reference

GDScript is a high-level, object-oriented [https://en.wikipedia.org/wiki/Object-oriented_programming], imperative [https://en.wikipedia.org/wiki/Imperative_programming], and gradually typed [https://en.wikipedia.org/wiki/Gradual_typing] programming language built for Godot. It uses an indentation-based syntax similar to languages like Python [https://en.wikipedia.org/wiki/Python_%28programming_language%29]. Its goal is to be optimized for and tightly integrated with Godot Engine, allowing great flexibility for content creation and integration.

GDScript is entirely independent from Python and is not based on it.

History

Note

Documentation about GDScript's history has been moved to the Frequently Asked Questions.

Example of GDScript

Some people can learn better by taking a look at the syntax, so here's an example of how GDScript looks.

```
# Everything after "#" is a comment.
# A file is a class!

# (optional) icon to show in the editor dialogs:
@icon("res://path/to/optional/icon.svg")

# (optional) class definition:
```

```gdscript
class_name MyClass

# Inheritance:
extends BaseClass


# Member variables.
var a = 5
var s = "Hello"
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var other_dict = {key = "value", other_key = 2}
var typed_var: int
var inferred_type := "String"

# Constants.
const ANSWER = 42
const THE_NAME = "Charly"

# Enums.
enum {UNIT_NEUTRAL, UNIT_ENEMY, UNIT_ALLY}
enum Named {THING_1, THING_2, ANOTHER_THING = -1}

# Built-in vector types.
var v2 = Vector2(1, 2)
var v3 = Vector3(1, 2, 3)


# Functions.
func some_function(param1, param2, param3):
    const local_const = 5

    if param1 < local_const:
        print(param1)
    elif param2 > 5:
        print(param2)
    else:
        print("Fail!")

    for i in range(20):
        print(i)

    while param2 != 0:
        param2 -= 1

    match param3:
        3:
            print("param3 is 3!")
        _:
            print("param3 is not 3!")

    var local_var = param1 + 3
    return local_var


# Functions override functions with the same name on the
base/super class.
# If you still want to call them, use "super":
func something(p1, p2):
    super(p1, p2)


# It's also possible to call another function in the super
class:
func other_something(p1, p2):
    super.something(p1, p2)


# Inner class
class Something:
    var a = 10


# Constructor
func _init():
    print("Constructed!")
    var lv = Something.new()
    print(lv.a)
```

If you have previous experience with statically typed languages such as C, C++, or C# but never used a dynamically typed one before, it is advised you read this tutorial: GDScript: An introduction to dynamic languages.

Language

In the following, an overview is given to GDScript. Details, such as which methods are available to arrays or other objects, should be looked up in the linked class descriptions.

Identifiers

Any string that restricts itself to alphabetic characters (`a` to `z` and `A` to `Z`), digits (`0` to `9`) and _ qualifies as an identifier. Additionally, identifiers must not begin with a digit. Identifiers are case-sensitive (`foo` is different from `FOO`).

Identifiers may also contain most Unicode characters part of UAX#31 [https://www.unicode.org/reports/tr31/]. This allows you to use identifier names written in languages other than English. Unicode characters that are considered "confusable" for ASCII characters and emoji are not allowed in identifiers.

Keywords

The following is the list of keywords supported by the language. Since keywords are reserved words (tokens), they can't be used as identifiers. Operators (like `in`, `not`, `and` or `or`) and names of built-in types as listed in the following sections are also reserved.

Keywords are defined in the GDScript tokenizer [https://github.com/godotengine/godot/blob/master/modules/gdscript/gdscript_tokenizer.cpp] in case you want to take a look under the hood.

Keyword	Description
if	See if/else/elif.
elif	See if/else/elif.
else	See if/else/elif.
for	See for.

Keyword	Description
while	See while.
match	See match.
break	Exits the execution of the current `for` or `while` loop.
continue	Immediately skips to the next iteration of the `for` or `while` loop.
pass	Used where a statement is required syntactically but execution of code is undesired, e.g. in empty functions.
return	Returns a value from a function.
class	Defines an inner class. See Inner classes.
class_name	Defines the script as a globally accessible class with the specified name. See Registering named classes.
extends	Defines what class to extend with the current class.
is	Tests whether a variable extends a given class, or is of a given built-in type.

Keyword	Description
in	Tests whether a value is within a string, array, range, dictionary, or node. When used with `for`, it iterates through them instead of testing.
as	Cast the value to a given type if possible.
self	Refers to current class instance.
signal	Defines a signal.
func	Defines a function.
static	Defines a static function or a static member variable.
const	Defines a constant.
enum	Defines an enum.
var	Defines a variable.
breakpoint	Editor helper for debugger breakpoints. Unlike breakpoints created by clicking in the gutter, `breakpoint` is stored in the script itself. This makes it persistent across different machines when using version control.

Keyword	Description
preload	Preloads a class or variable. See Classes as resources.
await	Waits for a signal or a coroutine to finish. See Awaiting for signals or coroutines.
yield	Previously used for coroutines. Kept as keyword for transition.
assert	Asserts a condition, logs error on failure. Ignored in non-debug builds. See Assert keyword.
void	Used to represent that a function does not return any value.
PI	PI constant.
TAU	TAU constant.
INF	Infinity constant. Used for comparisons and as result of calculations.
NAN	NAN (not a number) constant. Used as impossible result from calculations.

Operators

The following is the list of supported operators and their precedence.

Operator	**Description**
`( )`	Grouping (highest priority) Parentheses are not really an operator, but allow you to explicitly specify the precedence of an operation.
`x[index]`	Subscription
`x.attribute`	Attribute reference
`foo()`	Function call
`await x`	Awaiting for signals or coroutines
`x is Node`	Type checking See also is_instance_of() function.
`x ** y`	Power Multiplies `x` by itself `y` times, similar to calling pow() function. **Note:** In GDScript, the `**` operator is left-associative. See a detailed note after the table.

Operator	Description
`~x`	Bitwise NOT
`+x` `-x`	Identity / Negation

`x * y`
`x / y`
`x % y`
	Multiplication / Division / Remainder
	
	The `%` operator is additionally used for format strings.
	
	Note: These operators have the same behavior as C++, which may be unexpected for users coming from Python, JavaScript, etc. See a detailed note after the table.

`x + y`
`x - y`
	Addition (or Concatenation) / Subtraction

`x << y`
`x >> y`
	Bit shifting

`x & y`	Bitwise AND
`x ^ y`	Bitwise XOR
`x | y`	Bitwise OR

`x == y`
`x != y`
	Comparison

Operator	Description

`x < y`
`x > y`
`x <= y`
`x >= y`
	See a detailed note after the table.

`x in y`
`x not in y`
	Inclusion checking
	
	`in` is also used with the for keyword as part of the syntax.

`not x`
`!x`
	Boolean NOT and its unrecommended alias

`x and y`
`x && y`
	Boolean AND and its unrecommended alias

`x or y`
`x || y`
	Boolean OR and its unrecommended alias

`true_expr if cond else false_expr`	Ternary if/else
`x as Node`	Type casting

`x = y`
`x += y`
`x -= y`
`x *= y`
`x /= y`
`x **= y`
`x %= y`
`x &= y`
	Assignment (lowest priority)
	
	You cannot use an assignment operator inside an expression.

Operator	Description

`x |= y`
`x ^= y`
`x <<= y`
`x >>= y`	

Note

The behavior of some operators may differ from what you expect:

1. If both operands of the `/` operator are int, then integer division is performed instead of fractional. For example `5 / 2 == 2`, not `2.5`. If this is not desired, use at least one float literal (`x / 2.0`), cast (`float(x) / y`), or multiply by `1.0` (`x * 1.0 / y`).
2. The `%` operator is only available for ints, for floats use the fmod() function.
3. For negative values, the `%` operator and `fmod()` use truncation [https://en.wikipedia.org/wiki/Truncation] instead of rounding towards negative infinity. This means that the remainder has a sign. If you need the remainder in a mathematical sense, use the posmod() and fposmod() functions instead.
4. The `**` operator is left-associative [https://en.wikipedia.org/wiki/Operator_associativity]. This means that `2 ** 2 ** 3` is equal to `(2 ** 2) ** 3`. Use parentheses to explicitly specify precedence you need, for example `2 ** (2 ** 3)`.
5. The `==` and `!=` operators sometimes allow you to compare values of different types (for example, `1 == 1.0` is true), but in other cases it can cause a runtime error. If you're not sure about the types of the operands, you can safely use the is_same() function (but note that it is more strict about types and references). To compare floats, use the is_equal_approx() and is_zero_approx() functions instead.

Literals

Example(s)	Description
`null`	Null value
`false`, `true`	Boolean values
`45`	Base 10 integer
`0x8f51`	Base 16 (hexadecimal) integer
`0b101010`	Base 2 (binary) integer
`3.14`, `58.1e-10`	Floating-point number (real)
`"Hello"`, `'Hi'`	Regular strings
`"""Hello"""`, `'''Hi'''`	Triple-quoted regular strings
`r"Hello"`, `r'Hi'`	Raw strings
`r"""Hello"""`, `r'''Hi'''`	Triple-quoted raw strings
`&"name"`	StringName
`^"Node/Label"`	NodePath

There are also two constructs that look like literals, but actually are not:

Example	Description
`$NodePath`	Shorthand for `get_node("NodePath")`
`%UniqueNode`	Shorthand for `get_node("%UniqueNode")`

Integers and floats can have their numbers separated with _ to make them more readable. The following ways to write numbers are all valid:

```
12_345_678  # Equal to 12345678.
3.141_592_7  # Equal to 3.1415927.
0x8080_0000_ffff  # Equal to 0x80800000ffff.
0b11_00_11_00  # Equal to 0b11001100.
```

Regular string literals can contain the following escape sequences:

Escape sequence	Expands to
`\n`	Newline (line feed)
`\t`	Horizontal tab character
`\r`	Carriage return
`\a`	Alert (beep/bell)

`\b`	Backspace
`\f`	Formfeed page break
`\v`	Vertical tab character
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash
`\uXXXX`	UTF-16 Unicode codepoint `XXXX` (hexadecimal, case-insensitive)
`\UXXXXXX`	UTF-32 Unicode codepoint `XXXXXX` (hexadecimal, case-insensitive)

There are two ways to represent an escaped Unicode character above `0xFFFF`:

- as a UTF-16 surrogate pair [https://en.wikipedia.org/wiki/UTF-16#Code_points_from_U+010000_to_U+10FFFF] `\uXXXX\uXXXX`.
- as a single UTF-32 codepoint `\UXXXXXX`.

Also, using `\` followed by a newline inside a string will allow you to continue it in the next line, without inserting a newline character in the string itself.

A string enclosed in quotes of one type (for example `"`) can contain quotes of another type (for example `'`) without escaping. Triple-quoted strings allow you to avoid escaping up to two consecutive quotes of the same type (unless they are adjacent to the string edges).

Raw string literals always encode the string as it appears in the source code. This is especially useful for regular expressions. Raw strings do not process escape sequences, but you can "escape" a quote or backslash (they replace themselves).

```
print("\tchar=\"\\t\"")  # Prints `    char="\t"`.
print(r"\tchar=\"\\t\"") # Prints `\tchar=\"\\t\"`.
```

GDScript also supports format strings.

Annotations

There are some special tokens in GDScript that act like keywords but are not, they are *annotations* instead. Every annotation start with the `@` character and is specified by a name. A detailed description and example for each annotation can be found inside the GDScript class reference.

Annotations affect how the script is treated by external tools and usually don't change the behavior.

For instance, you can use it to export a value to the editor:

```
@export_range(1, 100, 1, "or_greater")
var ranged_var: int = 50
```

For more information about exporting properties, read the GDScript exports article.

Any constant expression compatible with the required argument type can be passed as an annotation argument value:

```
const MAX_SPEED = 120.0

@export_range(0.0, 0.5 * MAX_SPEED)
var initial_speed: float = 0.25 * MAX_SPEED
```

Annotations can be specified one per line or all in the same line. They affect the next statement that isn't an annotation. Annotations can have arguments sent between parentheses and separated by commas.

Both of these are the same:

```
@annotation_a
@annotation_b
var variable

@annotation_a @annotation_b var variable
```

`@onready` annotation

When using nodes, it's common to desire to keep references to parts of the scene in a variable. As scenes are only warranted to be configured when entering the active scene tree, the sub-nodes can only be obtained when a call to `Node._ready()` is made.

```
var my_label


func _ready():
    my_label = get_node("MyLabel")
```

This can get a little cumbersome, especially when nodes and external references pile up. For this, GDScript has the `@onready` annotation, that defers initialization of a member variable until `_ready()` is called. It can replace the above code with a single line:

```
@onready var my_label = get_node("MyLabel")
```

Warning

Applying `@onready` and any `@export` annotation to the same variable doesn't work as you might expect. The `@onready` annotation will cause the default value to be set after the `@export` takes effect and will override it:

```
@export var a = "init_value_a"
@onready @export var b = "init_value_b"

func _init():
    prints(a, b) # init_value_a <null>

func _notification(what):
    if what == NOTIFICATION_SCENE_INSTANTIATED:
        prints(a, b) # exported_value_a exported_value_b

func _ready():
    prints(a, b) # exported_value_a init_value_b
```

Therefore, the `ONREADY_WITH_EXPORT` warning is generated, which is treated as an error by default. We do not recommend disabling or ignoring it.

Comments

Anything from a `#` to the end of the line is ignored and is considered a comment.

```
# This is a comment.
```

Tip

In the Godot script editor, special keywords are highlighted within comments to bring the user's attention to specific comments:

- **Critical** *(appears in red)*: `ALERT`, `ATTENTION`, `CAUTION`, `CRITICAL`, `DANGER`, `SECURITY`
- **Warning** *(appears in yellow)*: `BUG`, `DEPRECATED`, `FIXME`, `HACK`, `TASK`, `TBD`, `TODO`, `WARNING`
- **Notice** *(appears in green)*: `INFO`, `NOTE`, `NOTICE`, `TEST`, `TESTING`

These keywords are case-sensitive, so they must be written in uppercase for them to be recognized:

```
# In the example below, "TODO" will appear in yellow by
default.
# The `:` symbol after the keyword is not required, but
```

```
it's often used.

# TODO: Add more items for the player to choose from.
```

The list of highlighted keywords and their colors can be changed in the **Text Editor > Theme > Comment Markers** section of the Editor Settings.

Code regions

Code regions are special types of comments that the script editor understands as *foldable regions*. This means that after writing code region comments, you can collapse and expand the region by clicking the arrow that appears at the left of the comment. This arrow appears within a purple square to be distinguishable from standard code folding.

The syntax is as follows:

```
# Important: There must be *no* space between the `#` and
`region` or `endregion`.

# Region without a description:
#region
...
#endregion

# Region with a description:
#region Some description that is displayed even when
collapsed
...
#endregion
```

Tip

To create a code region quickly, select several lines in the script editor, right-click the selection then choose **Create Code Region**. The region description will be selected automatically for editing.

It is possible to nest code regions within other code regions.

Here's a concrete usage example of code regions:

```
# This comment is outside the code region. It will be visible
when collapsed.
#region Terrain generation
# This comment is inside the code region. It won't be visible
when collapsed.
func generate_lakes():
    pass

func generate_hills():
    pass
#endregion

#region Terrain population
func place_vegetation():
    pass

func place_roads():
    pass
#endregion
```

This can be useful to organize large chunks of code into easier to understand sections. However, remember that external editors generally don't support this feature, so make sure your code is easy to follow even when not relying on folding code regions.

Note

Individual functions and indented sections (such as `if` and `for`) can *always* be collapsed in the script editor. This means you should avoid using a code region to contain a single function or indented section, as it won't bring much of a benefit. Code regions work best when they're used to group multiple elements together.

Line continuation

A line of code in GDScript can be continued on the next line by using a backslash (`\`). Add one at the end of a line and the code on the next line will act like it's where the backslash is. Here is an example:

```
var a = 1 + \
2
```

A line can be continued multiple times like this:

```
var a = 1 + \
4 + \
10 + \
4
```

Built-in types

Built-in types are stack-allocated. They are passed as values. This means a copy is created on each assignment or when passing them as arguments to functions. The exceptions are `Object`, `Array`, `Dictionary`, and packed arrays (such as `PackedByteArray`), which are passed by reference so they are shared. All arrays, `Dictionary`, and some objects (`Node`, `Resource`) have a `duplicate()` method that allows you to make a copy.

Basic built-in types

A variable in GDScript can be assigned to several built-in types.

null

`null` is an empty data type that contains no information and can not be assigned any other value.

bool

Short for "boolean", it can only contain `true` or `false`.

int

Short for "integer", it stores whole numbers (positive and negative). It is stored as a 64-bit value, equivalent to `int64_t` in C++.

float

Stores real numbers, including decimals, using floating-point values. It is stored as a 64-bit value, equivalent to `double` in C++. Note: Currently, data structures such as `Vector2`, `Vector3`, and `PackedFloat32Array` store 32-bit single-precision `float` values.

String

A sequence of characters in Unicode format [https://en.wikipedia.org/wiki/Unicode].

StringName

An immutable string that allows only one instance of each name. They are slower to create and may result in waiting for locks when multithreading. In exchange, they're very fast to compare, which makes them good candidates for dictionary keys.

NodePath

A pre-parsed path to a node or a node property. It can be easily assigned to, and from, a String. They are useful to interact with the tree to get a node, or affecting properties like with Tweens.

Vector built-in types

Vector2

2D vector type containing `x` and `y` fields. Can also be accessed as an array.

Vector2i

Same as a Vector2 but the components are integers. Useful for representing items in a 2D grid.

Rect2

2D Rectangle type containing two vectors fields: `position` and `size`. Also contains an `end` field which is `position + size`.

Vector3

3D vector type containing `x`, `y` and `z` fields. This can also be accessed as an array.

Vector3i

Same as Vector3 but the components are integers. Can be use for indexing items in a 3D grid.

Transform2D

3×2 matrix used for 2D transforms.

Plane

3D Plane type in normalized form that contains a `normal` vector field and a `d` scalar distance.

Quaternion

Quaternion is a datatype used for representing a 3D rotation. It's useful for interpolating rotations.

AABB

Axis-aligned bounding box (or 3D box) contains 2 vectors fields: `position` and `size`. Also contains an `end` field which is `position + size`.

Basis

3x3 matrix used for 3D rotation and scale. It contains 3 vector fields (`x`, `y` and `z`) and can also be accessed as an array of 3D vectors.

Transform3D

3D Transform contains a Basis field `basis` and a Vector3 field `origin`.

Engine built-in types

Color

Color data type contains `r`, `g`, `b`, and `a` fields. It can also be accessed as `h`, `s`, and `v` for hue/saturation/value.

RID

Resource ID (RID). Servers use generic RIDs to reference opaque data.

Object

Base class for anything that is not a built-in type.

Container built-in types

Array

Generic sequence of arbitrary object types, including other arrays or dictionaries (see below). The array can resize dynamically.

Arrays are indexed starting from index `0`. Negative indices count from the end.

```
var arr = []
arr = [1, 2, 3]
var b = arr[1] # This is 2.
var c = arr[arr.size() - 1] # This is 3.
var d = arr[-1] # Same as the previous line, but shorter.
arr[0] = "Hi!" # Replacing value 1 with "Hi!".
arr.append(4) # Array is now ["Hi!", 2, 3, 4].
```

Typed arrays

Godot 4.0 added support for typed arrays. On write operations, Godot checks that element values match the specified type, so the array cannot contain invalid values. The GDScript static analyzer takes typed arrays into account, however array methods like `front()` and `back()` still have the `Variant` return type.

Typed arrays have the syntax `Array[Type]`, where `Type` can be any `Variant` type, native or user class, or enum. Nested array types (like `Array[Array[int]]`) are not supported.

```
var a: Array[int]
var b: Array[Node]
var c: Array[MyClass]
var d: Array[MyEnum]
var e: Array[Variant]
```

`Array` and `Array[Variant]` are the same thing.

Note

Arrays are passed by reference, so the array element type is also an attribute of the in-memory structure referenced by a variable in runtime. The static type of a variable restricts the structures that it can reference to. Therefore, you **cannot** assign an array with a different element type, even if the type is a subtype of the required type.

If you want to *convert* a typed array, you can create a new array and use the Array.assign() method:

```
var a: Array[Node2D] = [Node2D.new()]

# (OK) You can add the value to the array because `Node2D`
extends `Node`.
var b: Array[Node] = [a[0]]

# (Error) You cannot assign an `Array[Node2D]` to an
`Array[Node]` variable.
b = a

# (OK) But you can use the `assign()` method instead.
Unlike the `=` operator,
# the `assign()` method copies the contents of the array,
not the reference.
b.assign(a)
```

The only exception was made for the `Array` (`Array[Variant]`) type, for user convenience and compatibility with old code. However, operations on untyped arrays are considered unsafe.

Packed arrays

GDScript arrays are allocated linearly in memory for speed. Large arrays (more than tens of thousands of elements) may however cause memory fragmentation. If this is a concern, special types of arrays are available. These only accept a single data type. They avoid memory fragmentation and use less memory, but are atomic and tend to run slower than generic arrays. They are therefore only recommended to use for large data sets:

- PackedByteArray: An array of bytes (integers from 0 to 255).
- PackedInt32Array: An array of 32-bit integers.
- PackedInt64Array: An array of 64-bit integers.
- PackedFloat32Array: An array of 32-bit floats.
- PackedFloat64Array: An array of 64-bit floats.
- PackedStringArray: An array of strings.
- PackedVector2Array: An array of Vector2 values.

- PackedVector3Array: An array of Vector3 values.
- PackedColorArray: An array of Color values.

Dictionary

Associative container which contains values referenced by unique keys.

```
var d = {4: 5, "A key": "A value", 28: [1, 2, 3]}
d["Hi!"] = 0
d = {
    22: "value",
    "some_key": 2,
    "other_key": [2, 3, 4],
    "more_key": "Hello"
}
```

Lua-style table syntax is also supported. Lua-style uses `=` instead of `:` and doesn't use quotes to mark string keys (making for slightly less to write). However, keys written in this form can't start with a digit (like any GDScript identifier), and must be string literals.

```
var d = {
    test22 = "value",
    some_key = 2,
    other_key = [2, 3, 4],
    more_key = "Hello"
}
```

To add a key to an existing dictionary, access it like an existing key and assign to it:

```
var d = {} # Create an empty Dictionary.
d.waiting = 14 # Add String "waiting" as a key and assign the
value 14 to it.
d[4] = "hello" # Add integer 4 as a key and assign the String
"hello" as its value.
d["Godot"] = 3.01 # Add String "Godot" as a key and assign
the value 3.01 to it.

var test = 4
# Prints "hello" by indexing the dictionary with a dynamic
key.
```

```
# This is not the same as `d.test`. The bracket syntax equivalent to
# `d.test` is `d["test"]`.
print(d[test])
```

Note

The bracket syntax can be used to access properties of any Object, not just Dictionaries. Keep in mind it will cause a script error when attempting to index a non-existing property. To avoid this, use the Object.get() and Object.set() methods instead.

Signal

A signal is a message that can be emitted by an object to those who want to listen to it. The Signal type can be used for passing the emitter around.

Signals are better used by getting them from actual objects, e.g. `$Button.button_up`.

Callable

Contains an object and a function, which is useful for passing functions as values (e.g. when connecting to signals).

Getting a method as a member returns a callable. `var x = $Sprite2D.rotate` will set the value of `x` to a callable with `$Sprite2D` as the object and `rotate` as the method.

You can call it using the `call` method: `x.call(PI)`.

Data

Variables

Variables can exist as class members or local to functions. They are created with the `var` keyword and may, optionally, be assigned a value upon initialization.

```
var a # Data type is 'null' by default.
var b = 5
var c = 3.8
var d = b + c # Variables are always initialized in order.
```

Variables can optionally have a type specification. When a type is specified, the variable will be forced to have always that same type, and trying to assign an incompatible value will raise an error.

Types are specified in the variable declaration using a `:` (colon) symbol after the variable name, followed by the type.

```
var my_vector2: Vector2
var my_node: Node = Sprite2D.new()
```

If the variable is initialized within the declaration, the type can be inferred, so it's possible to omit the type name:

```
var my_vector2 := Vector2() # 'my_vector2' is of type
'Vector2'.
var my_node := Sprite2D.new() # 'my_node' is of type
'Sprite2D'.
```

Type inference is only possible if the assigned value has a defined type, otherwise it will raise an error.

Valid types are:

- Built-in types (Array, Vector2, int, String, etc.).
- Engine classes (Node, Resource, Reference, etc.).
- Constant names if they contain a script resource (`MyScript` if you declared `const MyScript = preload("res://my_script.gd")`).
- Other classes in the same script, respecting scope (`InnerClass.NestedClass` if you declared `class NestedClass` inside the `class InnerClass` in the same scope).
- Script classes declared with the `class_name` keyword.

- Autoloads registered as singletons.

Note

While `Variant` is a valid type specification, it's not an actual type. It only means there's no set type and is equivalent to not having a static type at all. Therefore, inference is not allowed by default for `Variant`, since it's likely a mistake.

You can turn off this check, or make it only a warning, by changing it in the project settings. See GDScript warning system for details.

Static variables

A class member variable can be declared static:

```
static var a
```

Static variables belong to the class, not instances. This means that static variables share values between multiple instances, unlike regular member variables.

From inside a class, you can access static variables from any function, both static and non-static. From outside the class, you can access static variables using the class or an instance (the second is not recommended as it is less readable).

Note

The `@export` and `@onready` annotations cannot be applied to a static variable. Local variables cannot be static.

The following example defines a `Person` class with a static variable named `max_id`. We increment the `max_id` in the `_init()` function. This makes it easy to keep track of the number of `Person` instances in our game.

```
# person.gd
class_name Person

static var max_id = 0

var id
var name

func _init(p_name):
    max_id += 1
    id = max_id
    name = p_name
```

In this code, we create two instances of our `Person` class and check that the class and every instance have the same `max_id` value, because the variable is static and accessible to every instance.

```
# test.gd
extends Node

func _ready():
    var person1 = Person.new("John Doe")
    var person2 = Person.new("Jane Doe")

    print(person1.id) # 1
    print(person2.id) # 2

    print(Person.max_id)  # 2
    print(person1.max_id) # 2
    print(person2.max_id) # 2
```

Static variables can have type hints, setters and getters:

```
static var balance: int = 0

static var debt: int:
    get:
        return -balance
    set(value):
        balance = -value
```

A base class static variable can also be accessed via a child class:

```
class A:
    static var x = 1
```

```
class B extends A:
    pass

func _ready():
    prints(A.x, B.x) # 1 1
    A.x = 2
    prints(A.x, B.x) # 2 2
    B.x = 3
    prints(A.x, B.x) # 3 3
```

`@static_unload` annotation

Since GDScript classes are resources, having static variables in a script prevents it from being unloaded even if there are no more instances of that class and no other references left. This can be important if static variables store large amounts of data or hold references to other project resources, such as scenes. You should clean up this data manually, or use the @static_unload annotation if static variables don't store important data and can be reset.

Warning

Currently, due to a bug, scripts are never freed, even if `@static_unload` annotation is used.

Note that `@static_unload` applies to the entire script (including inner classes) and must be placed at the top of the script, before `class_name` and `extends`:

```
@static_unload
class_name MyNode
extends Node
```

See also Static functions and Static constructor.

Casting

Values assigned to typed variables must have a compatible type. If it's needed to coerce a value to be of a certain type, in particular for object types, you can use the casting operator `as`.

Casting between object types results in the same object if the value is of the same type or a subtype of the cast type.

```
var my_node2D: Node2D
my_node2D = $Sprite2D as Node2D # Works since Sprite2D is a
subtype of Node2D.
```

If the value is not a subtype, the casting operation will result in a `null` value.

```
var my_node2D: Node2D
my_node2D = $Button as Node2D # Results in 'null' since a
Button is not a subtype of Node2D.
```

For built-in types, they will be forcibly converted if possible, otherwise the engine will raise an error.

```
var my_int: int
my_int = "123" as int # The string can be converted to int.
my_int = Vector2() as int # A Vector2 can't be converted to
int, this will cause an error.
```

Casting is also useful to have better type-safe variables when interacting with the scene tree:

```
# Will infer the variable to be of type Sprite2D.
var my_sprite := $Character as Sprite2D

# Will fail if $AnimPlayer is not an AnimationPlayer, even if
it has the method 'play()'.
($AnimPlayer as AnimationPlayer).play("walk")
```

Constants

Constants are values you cannot change when the game is running. Their value must be known at compile-time. Using the `const` keyword allows you to give a constant value a name. Trying to assign a value to a constant after it's declared will give you an error.

We recommend using constants whenever a value is not meant to change.

```
const A = 5
const B = Vector2(20, 20)
const C = 10 + 20 # Constant expression.
const D = Vector2(20, 30).x # Constant expression: 20.
const E = [1, 2, 3, 4][0] # Constant expression: 1.
const F = sin(20) # 'sin()' can be used in constant
expressions.
const G = x + 20 # Invalid; this is not a constant
expression!
const H = A + 20 # Constant expression: 25 (`A` is a
constant).
```

Although the type of constants is inferred from the assigned value, it's also possible to add explicit type specification:

```
const A: int = 5
const B: Vector2 = Vector2()
```

Assigning a value of an incompatible type will raise an error.

You can also create constants inside a function, which is useful to name local magic values.

Note

Since objects, arrays and dictionaries are passed by reference, constants are "flat". This means that if you declare a constant array or dictionary, it can still be modified afterwards. They can't be reassigned with another value though.

Enums

Enums are basically a shorthand for constants, and are pretty useful if you want to assign consecutive integers to some constant.

```
enum {TILE_BRICK, TILE_FLOOR, TILE_SPIKE, TILE_TELEPORT}

# Is the same as:
const TILE_BRICK = 0
const TILE_FLOOR = 1
const TILE_SPIKE = 2
const TILE_TELEPORT = 3
```

If you pass a name to the enum, it will put all the keys inside a constant Dictionary of that name. This means all constant methods of a dictionary can also be used with a named enum.

Important

Keys in a named enum are not registered as global constants. They should be accessed prefixed by the enum's name (`Name.KEY`).

```
enum State {STATE_IDLE, STATE_JUMP = 5, STATE_SHOOT}

# Is the same as:
const State = {STATE_IDLE = 0, STATE_JUMP = 5, STATE_SHOOT =
6}
# Access values with State.STATE_IDLE, etc.

func _ready():
    # Access values with Name.KEY, prints '5'
    print(State.STATE_JUMP)
    # Use dictionary methods:
    # prints '["STATE_IDLE", "STATE_JUMP", "STATE_SHOOT"]'
    print(State.keys())
    # prints '{ "STATE_IDLE": 0, "STATE_JUMP": 5,
"STATE_SHOOT": 6 }'
    print(State)
    # prints '[0, 5, 6]'
    print(State.values())
```

Functions

Functions always belong to a class. The scope priority for variable look-up is: local → class member → global. The `self` variable is always available and is provided as an option for accessing class members, but is not always required (and should *not* be sent as the function's first argument, unlike Python).

```
func my_function(a, b):
    print(a)
    print(b)
    return a + b  # Return is optional; without it 'null' is
returned.
```

A function can `return` at any point. The default return value is `null`.

If a function contains only one line of code, it can be written on one line:

```
func square(a): return a * a

func hello_world(): print("Hello World")

func empty_function(): pass
```

Functions can also have type specification for the arguments and for the return value. Types for arguments can be added in a similar way to variables:

```
func my_function(a: int, b: String):
    pass
```

If a function argument has a default value, it's possible to infer the type:

```
func my_function(int_arg := 42, String_arg := "string"):
    pass
```

The return type of the function can be specified after the arguments list using the arrow token (->):

```
func my_int_function() -> int:
    return 0
```

Functions that have a return type **must** return a proper value. Setting the type as `void` means the function doesn't return anything. Void functions can return early with the `return` keyword, but they can't return any value.

```
func void_function() -> void:
    return # Can't return a value.
```

Note

Non-void functions must **always** return a value, so if your code has branching statements (such as an `if`/`else` construct), all the possible paths must have a return. E.g., if you have a `return` inside an `if` block but not after it, the editor will raise an error because if the block is not executed, the function won't have a valid value to return.

Referencing functions

Functions are first-class items in terms of the Callable object. Referencing a function by name without calling it will automatically generate the proper callable. This can be used to pass functions as arguments.

```
func map(arr: Array, function: Callable) -> Array:
    var result = []
    for item in arr:
        result.push_back(function.call(item))
    return result

func add1(value: int) -> int:
    return value + 1;

func _ready() -> void:
    var my_array = [1, 2, 3]
    var plus_one = map(my_array, add1)
    print(plus_one) # Prints [2, 3, 4].
```

Note

Callables **must** be called with the `call` method. You cannot use the `()` operator directly. This behavior is implemented to avoid performance issues on direct function calls.

Lambda functions

Lambda functions allow you to declare functions that do not belong to a class. Instead a Callable object is created and assigned to a variable directly. This can be useful to create Callables to pass around without polluting the class scope.

```
var lambda = func(x): print(x)
lambda.call(42) # Prints "42"
```

Lambda functions can be named for debugging purposes:

```
var lambda = func my_lambda(x):
    print(x)
```

Lambda functions capture the local environment. Local variables are passed by value, so they won't be updated in the lambda if changed in the local function:

```
var x = 42
var my_lambda = func(): print(x)
my_lambda.call() # Prints "42"
x = "Hello"
my_lambda.call() # Prints "42"
```

Note

The values of the outer scope behave like constants. Therefore, if you declare an array or dictionary, it can still be modified afterwards.

Static functions

A function can be declared static. When a function is static, it has no access to the instance member variables or `self`. A static function has access to static variables. Also static functions are useful to make libraries of helper functions:

```
static func sum2(a, b):
    return a + b
```

Lambdas cannot be declared static.

See also Static variables and Static constructor.

Statements and control flow

Statements are standard and can be assignments, function calls, control flow structures, etc (see below). `;` as a statement separator is entirely optional.

Expressions

Expressions are sequences of operators and their operands in orderly fashion. An expression by itself can be a statement too, though only calls are reasonable to use as statements since other expressions don't have side effects.

Expressions return values that can be assigned to valid targets. Operands to some operator can be another expression. An assignment is not an expression and thus does not return any value.

Here are some examples of expressions:

```
2 + 2 # Binary operation.
-5 # Unary operation.
"okay" if x > 4 else "not okay" # Ternary operation.
x # Identifier representing variable or constant.
x.a # Attribute access.
x[4] # Subscript access.
x > 2 or x < 5 # Comparisons and logic operators.
x == y + 2 # Equality test.
do_something() # Function call.
[1, 2, 3] # Array definition.
{A = 1, B = 2} # Dictionary definition.
```

```
preload("res://icon.png") # Preload builtin function.
self # Reference to current instance.
```

Identifiers, attributes, and subscripts are valid assignment targets. Other expressions cannot be on the left side of an assignment.

if/else/elif

Simple conditions are created by using the `if`/`else`/`elif` syntax. Parenthesis around conditions are allowed, but not required. Given the nature of the tab-based indentation, `elif` can be used instead of `else`/`if` to maintain a level of indentation.

```
if (expression):
    statement(s)
elif (expression):
    statement(s)
else:
    statement(s)
```

Short statements can be written on the same line as the condition:

```
if 1 + 1 == 2: return 2 + 2
else:
    var x = 3 + 3
    return x
```

Sometimes, you might want to assign a different initial value based on a boolean expression. In this case, ternary-if expressions come in handy:

```
var x = (value) if (expression) else (value)
y += 3 if y < 10 else -1
```

Ternary-if expressions can be nested to handle more than 2 cases. When nesting ternary-if expressions, it is recommended to wrap the complete expression over multiple lines to preserve readability:

```
var count = 0

var fruit = (
        "apple" if count == 2
        else "pear" if count == 1
        else "banana" if count == 0
        else "orange"
)
print(fruit)  # banana

# Alternative syntax with backslashes instead of parentheses
(for multi-line expressions).
# Less lines required, but harder to refactor.
var fruit_alt = \
        "apple" if count == 2 \
        else "pear" if count == 1 \
        else "banana" if count == 0 \
        else "orange"
print(fruit_alt)  # banana
```

You may also wish to check if a value is contained within something. You can use an `if` statement combined with the `in` operator to accomplish this:

```
# Check if a letter is in a string.
var text = "abc"
if 'b' in text: print("The string contains b")

# Check if a variable is contained within a node.
if "varName" in get_parent(): print("varName is defined in
parent!")
```

while

Simple loops are created by using `while` syntax. Loops can be broken using `break` or continued using `continue` (which skips to the next iteration of the loop without executing any further code in the current iteration):

```
while (expression):
    statement(s)
```

for

To iterate through a range, such as an array or table, a *for* loop is used. When iterating over an array, the current array element is stored in the loop variable. When iterating over a dictionary, the *key* is stored in the loop variable.

```
for x in [5, 7, 11]:
    statement # Loop iterates 3 times with 'x' as 5, then 7
and finally 11.

var dict = {"a": 0, "b": 1, "c": 2}
for i in dict:
    print(dict[i]) # Prints 0, then 1, then 2.

for i in range(3):
    statement # Similar to [0, 1, 2] but does not allocate an
array.

for i in range(1, 3):
    statement # Similar to [1, 2] but does not allocate an
array.

for i in range(2, 8, 2):
    statement # Similar to [2, 4, 6] but does not allocate an
array.

for i in range(8, 2, -2):
    statement # Similar to [8, 6, 4] but does not allocate an
array.

for c in "Hello":
    print(c) # Iterate through all characters in a String,
print every letter on new line.

for i in 3:
    statement # Similar to range(3).

for i in 2.2:
    statement # Similar to range(ceil(2.2)).
```

If you want to assign values on an array as it is being iterated through, it is best to use `for i in array.size()`.

```
for i in array.size():
        array[i] = "Hello World"
```

The loop variable is local to the for-loop and assigning to it will not change the value on the array. Objects passed by reference (such as nodes) can still be manipulated by calling methods on the loop variable.

```
for string in string_array:
    string = "Hello World" # This has no effect

for node in node_array:
    node.add_to_group("Cool_Group") # This has an effect
```

match

A `match` statement is used to branch execution of a program. It's the equivalent of the `switch` statement found in many other languages, but offers some additional features.

Warning

`match` is more type strict than the `==` operator. For example `1` will **not** match `1.0`. The only exception is `String` vs `StringName` matching: for example, the String `"hello"` is considered equal to the StringName `&"hello"`.

Basic syntax

```
match <expression>:
    <pattern(s)>:
        <block>
    <pattern(s)> when <guard expression>:
        <block>
    <...>
```

Crash-course for people who are familiar with switch statements

1. Replace `switch` with `match`.
2. Remove `case`.
3. Remove any `break`s.
4. Change `default` to a single underscore.

The patterns are matched from top to bottom. If a pattern matches, the first corresponding block will be executed. After that, the execution continues below the `match` statement.

Note

The special `continue` behavior in `match` supported in 3.x was removed in Godot 4.0.

The following pattern types are available:

- Literal pattern

  Matches a literal:

  ```
  match x:
      1:
          print("We are number one!")
      2:
          print("Two are better than one!")
      "test":
          print("Oh snap! It's a string!")
  ```

- Expression pattern

  Matches a constant expression, an identifier, or an attribute access (`A.B`):

  ```
  match typeof(x):
      TYPE_FLOAT:
          print("float")
      TYPE_STRING:
          print("text")
      TYPE_ARRAY:
          print("array")
  ```

- Wildcard pattern

  This pattern matches everything. It's written as a single underscore.

It can be used as the equivalent of the `default` in a `switch` statement in other languages:

```
match x:
    1:
        print("It's one!")
    2:
        print("It's one times two!")
    _:
        print("It's not 1 or 2. I don't care to be
honest.")
```

- Binding pattern

  A binding pattern introduces a new variable. Like the wildcard pattern, it matches everything - and also gives that value a name. It's especially useful in array and dictionary patterns:

```
match x:
    1:
        print("It's one!")
    2:
        print("It's one times two!")
    var new_var:
        print("It's not 1 or 2, it's ", new_var)
```

- Array pattern

  Matches an array. Every single element of the array pattern is a pattern itself, so you can nest them.

  The length of the array is tested first, it has to be the same size as the pattern, otherwise the pattern doesn't match.

  Open-ended array: An array can be bigger than the pattern by making the last subpattern `..`

  Every subpattern has to be comma-separated.

```
match x:
    []:
        print("Empty array")
    [1, 3, "test", null]:
        print("Very specific array")
    [var start, _, "test"]:
        print("First element is ", start, ", and the
last is \"test\"")
    [42, ..]:
        print("Open ended array")
```

- Dictionary pattern

  Works in the same way as the array pattern. Every key has to be a constant pattern.

  The size of the dictionary is tested first, it has to be the same size as the pattern, otherwise the pattern doesn't match.

  Open-ended dictionary: A dictionary can be bigger than the pattern by making the last subpattern `..`.

  Every subpattern has to be comma separated.

  If you don't specify a value, then only the existence of the key is checked.

  A value pattern is separated from the key pattern with a `:`.

```
match x:
    {}:
        print("Empty dict")
    {"name": "Dennis"}:
        print("The name is Dennis")
    {"name": "Dennis", "age": var age}:
        print("Dennis is ", age, " years old.")
    {"name", "age"}:
        print("Has a name and an age, but it's not
Dennis :(")
    {"key": "godotisawesome", ..}:
        print("I only checked for one entry and
ignored the rest")
```

- Multiple patterns

  You can also specify multiple patterns separated by a comma. These patterns aren't allowed to have any bindings in them.

```
match x:
    1, 2, 3:
        print("It's 1 - 3")
    "Sword", "Splash potion", "Fist":
        print("Yep, you've taken damage")
```

Pattern guards

Only one branch can be executed per `match`. Once a branch is chosen, the rest are not checked. If you want to use the same pattern for multiple branches or to prevent choosing a branch with too general pattern, you can specify a guard expression after the list of patterns with the `when` keyword:

```
match point:
    [0, 0]:
        print("Origin")
    [_, 0]:
        print("Point on X-axis")
    [0, _]:
        print("Point on Y-axis")
    [var x, var y] when y == x:
        print("Point on line y = x")
    [var x, var y] when y == -x:
        print("Point on line y = -x")
    [var x, var y]:
        print("Point (%s, %s)" % [x, y])
```

- If there is no matching pattern for the current branch, the guard expression is **not** evaluated and the patterns of the next branch are checked.
- If a matching pattern is found, the guard expression is evaluated.
    - If it's true, then the body of the branch is executed and `match` ends.
    - If it's false, then the patterns of the next branch are checked.

Classes

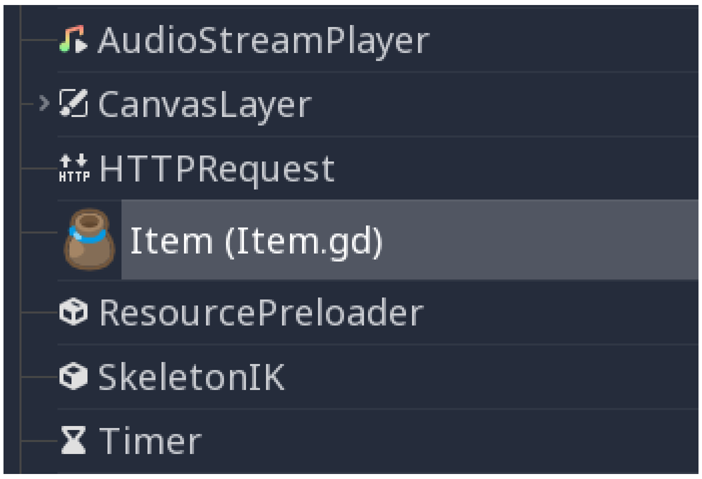
By default, all script files are unnamed classes. In this case, you can only reference them using the file's path, using either a relative or an absolute path. For example, if you name a script file `character.gd`:

```
# Inherit from 'character.gd'.

extends "res://path/to/character.gd"

# Load character.gd and create a new node instance from it.

var Character = load("res://path/to/character.gd")
var character_node = Character.new()
```

Registering named classes

You can give your class a name to register it as a new type in Godot's editor. For that, you use the `class_name` keyword. You can optionally use the `@icon` annotation with a path to an image, to use it as an icon. Your class will then appear with its new icon in the editor:

```
# item.gd

@icon("res://interface/icons/item.png")
class_name Item
extends Node
```



Here's a class file example:

```
# Saved as a file named 'character.gd'.

class_name Character
```

```
var health = 5


func print_health():
    print(health)


func print_this_script_three_times():
    print(get_script())
    print(ResourceLoader.load("res://character.gd"))
    print(Character)
```

If you want to use `extends` too, you can keep both on the same line:

```
class_name MyNode extends Node
```

Note

Godot initializes non-static variables every time you create an instance, and this includes arrays and dictionaries. This is in the spirit of thread safety, since scripts can be initialized in separate threads without the user knowing.

Inheritance

A class (stored as a file) can inherit from:

- A global class.
- Another class file.
- An inner class inside another class file.

Multiple inheritance is not allowed.

Inheritance uses the `extends` keyword:

```
# Inherit/extend a globally available class.
extends SomeClass

# Inherit/extend a named class file.
```

```
extends "somefile.gd"

# Inherit/extend an inner class in another file.
extends "somefile.gd".SomeInnerClass
```

Note

If inheritance is not explicitly defined, the class will default to inheriting RefCounted.

To check if a given instance inherits from a given class, the `is` keyword can be used:

```
# Cache the enemy class.
const Enemy = preload("enemy.gd")

# [...]

# Use 'is' to check inheritance.
if entity is Enemy:
    entity.apply_damage()
```

To call a function in a *super class* (i.e. one `extend`-ed in your current class), use the `super` keyword:

```
super(args)
```

This is especially useful because functions in extending classes replace functions with the same name in their super classes. If you still want to call them, you can use `super`:

```
func some_func(x):
    super(x) # Calls the same function on the super class.
```

If you need to call a different function from the super class, you can specify the function name with the attribute operator:

```
func overriding():
    return 0 # This overrides the method in the base class.

func dont_override():
    return super.overriding() # This calls the method as
defined in the base class.
```

Warning

One of the common misconceptions is trying to override *non-virtual* engine methods such as `get_class()`, `queue_free()`, etc. This is not supported for technical reasons.

In Godot 3, you can *shadow* engine methods in GDScript, and it will work if you call this method in GDScript. However, the engine will **not** execute your code if the method is called inside the engine on some event.

In Godot 4, even shadowing may not always work, as GDScript optimizes native method calls. Therefore, we added the `NATIVE_METHOD_OVERRIDE` warning, which is treated as an error by default. We strongly advise against disabling or ignoring the warning.

Note that this does not apply to virtual methods such as `_ready()`, `_process()` and others (marked with the `virtual` qualifier in the documentation and the names start with an underscore). These methods are specifically for customizing engine behavior and can be overridden in GDScript. Signals and notifications can also be useful for these purposes.

Class constructor

The class constructor, called on class instantiation, is named `_init`. If you want to call the base class constructor, you can also use the `super` syntax. Note that every class has an implicit constructor that it's always called (defining the default values of class variables). `super` is used to call the explicit constructor:

```
func _init(arg):
   super("some_default", arg) # Call the custom base
constructor.
```

This is better explained through examples. Consider this scenario:

```
# state.gd (inherited class).
var entity = null
var message = null


func _init(e=null):
    entity = e


func enter(m):
    message = m


# idle.gd (inheriting class).
extends "state.gd"


func _init(e=null, m=null):
    super(e)
    # Do something with 'e'.
    message = m
```

There are a few things to keep in mind here:

1. If the inherited class (`state.gd`) defines a `_init` constructor that takes arguments (`e` in this case), then the inheriting class (`idle.gd`) *must* define `_init` as well and pass appropriate parameters to `_init` from `state.gd`.

2. `idle.gd` can have a different number of arguments than the base class `state.gd`.

3. In the example above, `e` passed to the `state.gd` constructor is the same `e` passed in to `idle.gd`.

4. If `idle.gd`'s `_init` constructor takes 0 arguments, it still needs to pass some value to the `state.gd` base class, even if it does nothing. This brings us to the fact that you can pass expressions to the base constructor as well, not just variables, e.g.:

```
# idle.gd

func _init():
    super(5)
```

Static constructor

A static constructor is a static function `_static_init` that is called automatically when the class is loaded, after the static variables have been initialized:

```
static var my_static_var = 1

static func _static_init():
    my_static_var = 2
```

A static constructor cannot take arguments and must not return any value.

Inner classes

A class file can contain inner classes. Inner classes are defined using the `class` keyword. They are instanced using the `ClassName.new()` function.

```
# Inside a class file.

# An inner class in this class file.
class SomeInnerClass:
    var a = 5


    func print_value_of_a():
        print(a)


# This is the constructor of the class file's main class.
func _init():
    var c = SomeInnerClass.new()
    c.print_value_of_a()
```

Classes as resources

Classes stored as files are treated as resources. They must be loaded from disk to access them in other classes. This is done using either the `load` or `preload` functions (see below). Instancing of a loaded class resource is done by calling the `new` function on the class object:

```
# Load the class resource when calling load().
var MyClass = load("myclass.gd")

# Preload the class only once at compile time.
const MyClass = preload("myclass.gd")


func _init():
    var a = MyClass.new()
    a.some_function()
```

Exports

Note

Documentation about exports has been moved to GDScript exported properties.

Properties (setters and getters)

Sometimes, you want a class' member variable to do more than just hold data and actually perform some validation or computation whenever its value changes. It may also be desired to encapsulate its access in some way.

For this, GDScript provides a special syntax to define properties using the `set` and `get` keywords after a variable declaration. Then you can define a code block that will be executed when the variable is accessed or assigned.

Example:

```
var milliseconds: int = 0
var seconds: int:
    get:
        return milliseconds / 1000
    set(value):
        milliseconds = value * 1000
```

Note

Unlike `setget` in previous Godot versions, the properties setter and getter are **always** called (except as noted below), even when accessed inside the same class (with or without prefixing with `self.`). This makes the behavior consistent. If you need direct access to the value, use another variable for direct access and make the property code use that name.

Alternative syntax

Also there is another notation to use existing class functions if you want to split the code from the variable declaration or you need to reuse the code across multiple properties (but you can't distinguish which property the setter/getter is being called for):

```
var my_prop:
    get = get_my_prop, set = set_my_prop
```

This can also be done in the same line:

```
var my_prop: get = get_my_prop, set = set_my_prop
```

The setter and getter must use the same notation, mixing styles for the same variable is not allowed.

Note

You cannot specify type hints for *inline* setters and getters. This is done on purpose to reduce the boilerplate. If the variable is typed, then the setter's argument is automatically of the same type, and the getter's return value must match it. Separated setter/getter functions can have type hints, and the type must match the variable's type or be a wider type.

When setter/getter is not called

When a variable is initialized, the value of the initializer will be written directly to the variable. Including if the `@onready` annotation is applied to the variable.

Using the variable's name to set it inside its own setter or to get it inside its own getter will directly access the underlying member, so it won't generate infinite recursion and saves you from explicitly declaring another variable:

```
signal changed(new_value)
var warns_when_changed = "some value":
    get:
        return warns_when_changed
    set(value):
        changed.emit(value)
        warns_when_changed = value
```

This also applies to the alternative syntax:

```
var my_prop: set = set_my_prop

func set_my_prop(value):
    my_prop = value # No infinite recursion.
```

Warning

The exception does **not** propagate to other functions called in the setter/getter. For example, the following code **will** cause an infinite recursion:

```
var my_prop:
    set(value):
        set_my_prop(value)

func set_my_prop(value):
    my_prop = value # Infinite recursion, since
`set_my_prop()` is not the setter.
```

Tool mode

By default, scripts don't run inside the editor and only the exported properties can be changed. In some cases, it is desired that they do run inside the editor (as long as they don't execute game code or manually avoid doing so). For this, the `@tool` annotation exists and must be placed at the top of the file:

```
@tool
extends Button

func _ready():
    print("Hello")
```

See Running code in the editor for more information.

Warning

Be cautious when freeing nodes with `queue_free()` or `free()` in a tool script (especially the script's owner itself). As tool scripts run their code in the editor, misusing them may lead to crashing the editor.

Memory management

Godot implements reference counting to free certain instances that are no longer used, instead of a garbage collector, or requiring purely manual management. Any instance of the RefCounted class (or any class that inherits it, such as Resource) will be freed automatically when no longer in use. For an instance of any class that is not a RefCounted (such as Node or the base Object type), it will remain in memory until it is deleted with `free()` (or `queue_free()` for Nodes).

Note

If a Node is deleted via `free()` or `queue_free()`, all of its children will also recursively be deleted.

To avoid reference cycles that can't be freed, a WeakRef function is provided for creating weak references, which allow access to the object without preventing a RefCounted from freeing. Here is an example:

```
extends Node

var my_file_ref

func _ready():
    var f = FileAccess.open("user://example_file.json", 
FileAccess.READ)
    my_file_ref = weakref(f)
    # the FileAccess class inherits RefCounted, so it will be 
freed when not in use

    # the WeakRef will not prevent f from being freed when 
other_node is finished
    other_node.use_file(f)

func _this_is_called_later():
    var my_file = my_file_ref.get_ref()
    if my_file:
        my_file.close()
```

Alternatively, when not using references, the `is_instance_valid(instance)` can be used to check if an object has been freed.

Signals

Signals are a tool to emit messages from an object that other objects can react to. To create custom signals for a class, use the `signal` keyword.

```
extends Node


# A signal named health_depleted.
signal health_depleted
```

Note

Signals are a Callback [https://en.wikipedia.org/wiki/Callback_(computer_programming)] mechanism. They also fill the role of Observers, a common programming pattern. For more information, read the Observer tutorial [https://gameprogrammingpatterns.com/observer.html] in the Game Programming Patterns ebook.

You can connect these signals to methods the same way you connect built-in signals of nodes like Button or RigidBody3D.

In the example below, we connect the `health_depleted` signal from a `Character` node to a `Game` node. When the `Character` node emits the signal, the game node's `_on_character_health_depleted` is called:

```
# game.gd

func _ready():
    var character_node = get_node('Character')

character_node.health_depleted.connect(_on_character_health_d
epleted)


func _on_character_health_depleted():
    get_tree().reload_current_scene()
```

You can emit as many arguments as you want along with a signal.

Here is an example where this is useful. Let's say we want a life bar on screen to react to health changes with an animation, but we want to keep the user interface separate from the player in our scene tree.

In our `character.gd` script, we define a `health_changed` signal and emit it with Signal.emit(), and from a `Game` node higher up our scene tree, we connect it to the `Lifebar` using the Signal.connect() method:

```
# character.gd

...
signal health_changed


func take_damage(amount):
    var old_health = health
    health -= amount

    # We emit the health_changed signal every time the
    # character takes damage.
    health_changed.emit(old_health, health)
...
```

```
# lifebar.gd

# Here, we define a function to use as a callback when the
# character's health_changed signal is emitted.

...
func _on_Character_health_changed(old_value, new_value):
    if old_value > new_value:
        progress_bar.modulate = Color.RED
    else:
        progress_bar.modulate = Color.GREEN

    # Imagine that `animate` is a user-defined function that
animates the
    # bar filling up or emptying itself.
    progress_bar.animate(old_value, new_value)
...
```

In the `Game` node, we get both the `Character` and `Lifebar` nodes, then connect the character, that emits the signal, to the receiver, the `Lifebar` node in this case.

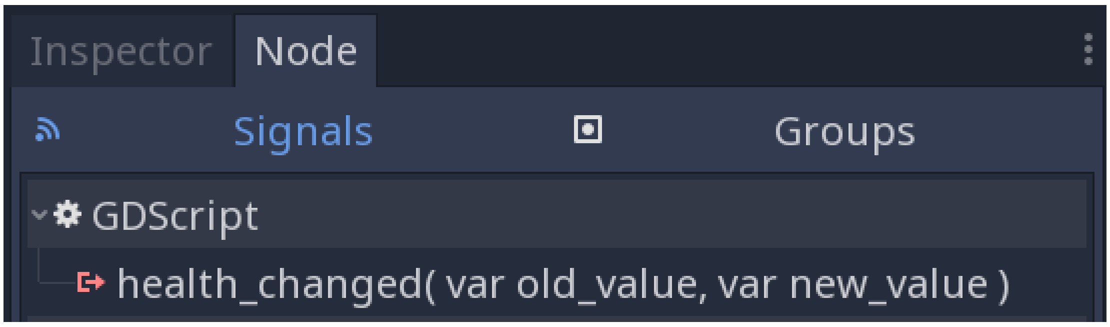
```
# game.gd

func _ready():
    var character_node = get_node('Character')
    var lifebar_node = get_node('UserInterface/Lifebar')


character_node.health_changed.connect(lifebar_node._on_Charac
ter_health_changed)
```

This allows the `Lifebar` to react to health changes without coupling it to the `Character` node.

You can write optional argument names in parentheses after the signal's definition:

```
# Defining a signal that forwards two arguments.
signal health_changed(old_value, new_value)
```

These arguments show up in the editor's node dock, and Godot can use them to generate callback functions for you. However, you can still emit any number of arguments when you emit signals; it's up to you to emit the correct values.



health_changed(var old_value, var new_value)

GDScript can bind an array of values to connections between a signal and a method. When the signal is emitted, the callback method receives the bound values. These bound arguments are unique to each connection, and the values will stay the same.

You can use this array of values to add extra constant information to the connection if the emitted signal itself doesn't give you access to all the data that you need.

Building on the example above, let's say we want to display a log of the damage taken by each character on the screen, like `Player1 took 22 damage.`. The `health_changed` signal doesn't give us the name of the character that took damage. So when we connect the signal to the in-game console, we can add the character's name in the binds array argument:

```
# game.gd

func _ready():
    var character_node = get_node('Character')
    var battle_log_node = get_node('UserInterface/BattleLog')


character_node.health_changed.connect(battle_log_node._on_Cha
racter_health_changed, [character_node.name])
```

Our `BattleLog` node receives each element in the binds array as an extra argument:

```
# battle_log.gd

func _on_Character_health_changed(old_value, new_value,
character_name):
    if not new_value <= old_value:
        return

    var damage = old_value - new_value
    label.text += character_name + " took " + str(damage) +
" damage."
```

Awaiting for signals or coroutines

The `await` keyword can be used to create coroutines [https://en.wikipedia.org/wiki/Coroutine] which wait until a signal is emitted before continuing execution. Using the `await` keyword with a signal or a call to a function that is also a coroutine will immediately return the control to the caller. When the signal is emitted (or the called coroutine finishes), it will resume execution from the point on where it stopped.

For example, to stop execution until the user presses a button, you can do something like this:

```
func wait_confirmation():
    print("Prompting user")
    await $Button.button_up # Waits for the button_up signal
from Button node.
    print("User confirmed")
    return true
```

In this case, the `wait_confirmation` becomes a coroutine, which means that the caller also needs to await for it:

```
func request_confirmation():
    print("Will ask the user")
    var confirmed = await wait_confirmation()
    if confirmed:
        print("User confirmed")
    else:
        print("User cancelled")
```

Note that requesting a coroutine's return value without `await` will trigger an error:

```
func wrong():
    var confirmed = wait_confirmation() # Will give an error.
```

However, if you don't depend on the result, you can just call it asynchronously, which won't stop execution and won't make the current function a coroutine:

```
func okay():
    wait_confirmation()
    print("This will be printed immediately, before the user
press the button.")
```

If you use await with an expression that isn't a signal nor a coroutine, the value will be returned immediately and the function won't give the control back to the caller:

```
func no_wait():
    var x = await get_five()
    print("This doesn't make this function a coroutine.")

func get_five():
    return 5
```

This also means that returning a signal from a function that isn't a coroutine will make the caller await on that signal:

```
func get_signal():
    return $Button.button_up

func wait_button():
    await get_signal()
    print("Button was pressed")
```

Note

Unlike `yield` in previous Godot versions, you cannot obtain the function state object. This is done to ensure type safety. With this type safety in place, a function cannot say that it returns an `int` while it actually returns a function state object during runtime.

Assert keyword

The `assert` keyword can be used to check conditions in debug builds. These assertions are ignored in non-debug builds. This means that the expression passed as argument won't be evaluated in a project exported in release mode. Due to this, assertions must **not** contain expressions that have side effects. Otherwise, the behavior of the script would vary depending on whether the project is run in a debug build.

```
# Check that 'i' is 0. If 'i' is not 0, an assertion error
will occur.
assert(i == 0)
```

When running a project from the editor, the project will be paused if an assertion error occurs.

You can optionally pass a custom error message to be shown if the assertion fails:

```
assert(enemy_power < 256, "Enemy is too powerful!")
```

GDScript: An introduction to dynamic languages

About

This tutorial aims to be a quick reference for how to use GDScript more efficiently. It focuses on common cases specific to the language, but also covers a lot of information on dynamically typed languages.

It's meant to be especially useful for programmers with little or no previous experience with dynamically typed languages.

Dynamic nature

Pros & cons of dynamic typing

GDScript is a Dynamically Typed language. As such, its main advantages are that:

- The language is easy to get started with.
- Most code can be written and changed quickly and without hassle.
- Less code written means less errors & mistakes to fix.
- The code is easy to read (little clutter).
- No compilation is required to test.
- Runtime is tiny.
- It has duck-typing and polymorphism by nature.

While the main disadvantages are:

- Less performance than statically typed languages.
- More difficult to refactor (symbols can't be traced).

- Some errors that would typically be detected at compile time in statically typed languages only appear while running the code (because expression parsing is more strict).
- Less flexibility for code-completion (some variable types are only known at run-time).

This, translated to reality, means that Godot used with GDScript is a combination designed to create games quickly and efficiently. For games that are very computationally intensive and can't benefit from the engine built-in tools (such as the Vector types, Physics Engine, Math library, etc), the possibility of using C++ is present too. This allows you to still create most of the game in GDScript and add small bits of C++ in the areas that need a performance boost.

Variables & assignment

All variables in a dynamically typed language are "variant"-like. This means that their type is not fixed, and is only modified through assignment. Example:

Static:

```
int a; // Value uninitialized.
a = 5; // This is valid.
a = "Hi!"; // This is invalid.
```

Dynamic:

```
var a # 'null' by default.
a = 5 # Valid, 'a' becomes an integer.
a = "Hi!" # Valid, 'a' changed to a string.
```

As function arguments:

Functions are of dynamic nature too, which means they can be called with different arguments, for example:

Static:

```
void print_value(int value) {

    printf("value is %i\n", value);
}

[..]

print_value(55); // Valid.
print_value("Hello"); // Invalid.
```

Dynamic:

```
func print_value(value):
    print(value)

[..]

print_value(55) # Valid.
print_value("Hello") # Valid.
```

Pointers & referencing:

In static languages, such as C or C++ (and to some extent Java and C#), there is a distinction between a variable and a pointer/reference to a variable. The latter allows the object to be modified by other functions by passing a reference to the original one.

In C# or Java, everything not a built-in type (int, float, sometimes String) is always a pointer or a reference. References are also garbage-collected automatically, which means they are erased when no longer used. Dynamically typed languages tend to use this memory model, too. Some Examples:

- C++:

```
void use_class(SomeClass *instance) {

    instance->use();
}

void do_something() {
    SomeClass *instance = new SomeClass; // Created as
pointer.
    use_class(instance); // Passed as pointer.
    delete instance; // Otherwise it will leak memory.
}
```

- Java:

```
@Override
public final void use_class(SomeClass instance) {

    instance.use();
}

public final void do_something() {

    SomeClass instance = new SomeClass(); // Created as
reference.
    use_class(instance); // Passed as reference.
    // Garbage collector will get rid of it when not in
    // use and freeze your game randomly for a second.
}
```

- GDScript:

```
func use_class(instance): # Does not care about class type
    instance.use() # Will work with any class that has a
".use()" method.

func do_something():
    var instance = SomeClass.new() # Created as reference.
    use_class(instance) # Passed as reference.
    # Will be unreferenced and deleted.
```

In GDScript, only base types (int, float, string and the vector types) are passed by value to functions (value is copied). Everything else (instances, arrays, dictionaries, etc) is passed as reference. Classes that inherit RefCounted (the default if nothing is specified) will be freed when not used, but manual memory management is allowed too if inheriting manually from Object.

Arrays

Arrays in dynamically typed languages can contain many different mixed datatypes inside and are always dynamic (can be resized at any time). Compare for example arrays in statically typed languages:

```
int *array = new int[4]; // Create array.
array[0] = 10; // Initialize manually.
array[1] = 20; // Can't mix types.
array[2] = 40;
array[3] = 60;
// Can't resize.
use_array(array); // Passed as pointer.
delete[] array; // Must be freed.

// or

std::vector<int> array;
array.resize(4);
array[0] = 10; // Initialize manually.
array[1] = 20; // Can't mix types.
array[2] = 40;
array[3] = 60;
array.resize(3); // Can be resized.
use_array(array); // Passed reference or value.
// Freed when stack ends.
```

And in GDScript:

```
var array = [10, "hello", 40, 60] # You can mix types.
array.resize(3) # Can be resized.
use_array(array) # Passed as reference.
# Freed when no longer in use.
```

In dynamically typed languages, arrays can also double as other datatypes, such as lists:

```
var array = []
array.append(4)
array.append(5)
array.pop_front()
```

Or unordered sets:

```
var a = 20
if a in [10, 20, 30]:
    print("We have a winner!")
```

Dictionaries

Dictionaries are a powerful tool in dynamically typed languages. Most programmers that come from statically typed languages (such as C++ or C#) ignore their existence and make their life unnecessarily more difficult. This datatype is generally not present in such languages (or only in limited form).

Dictionaries can map any value to any other value with complete disregard for the datatype used as either key or value. Contrary to popular belief, they are efficient because they can be implemented with hash tables. They are, in fact, so efficient that some languages will go as far as implementing arrays as dictionaries.

Example of Dictionary:

```
var d = {"name": "John", "age": 22}
print("Name: ", d["name"], " Age: ", d["age"])
```

Dictionaries are also dynamic, keys can be added or removed at any point at little cost:

```
d["mother"] = "Rebecca" # Addition.
d["age"] = 11 # Modification.
d.erase("name") # Removal.
```

In most cases, two-dimensional arrays can often be implemented more easily with dictionaries. Here's a battleship game example:

```
# Battleship Game

const SHIP = 0
const SHIP_HIT = 1
const WATER_HIT = 2

var board = {}

```

```
func initialize():
    board[Vector2(1, 1)] = SHIP
    board[Vector2(1, 2)] = SHIP
    board[Vector2(1, 3)] = SHIP

func missile(pos):
    if pos in board: # Something at that position.
        if board[pos] == SHIP: # There was a ship! hit it.
            board[pos] = SHIP_HIT
        else:
            print("Already hit here!") # Hey dude you already
hit here.
    else: # Nothing, mark as water.
        board[pos] = WATER_HIT

func game():
    initialize()
    missile(Vector2(1, 1))
    missile(Vector2(5, 8))
    missile(Vector2(2, 3))
```

Dictionaries can also be used as data markup or quick structures. While GDScript's dictionaries resemble python dictionaries, it also supports Lua style syntax and indexing, which makes it useful for writing initial states and quick structs:

```
# Same example, lua-style support.
# This syntax is a lot more readable and usable.
# Like any GDScript identifier, keys written in this form
cannot start
# with a digit.

var d = {
    name = "John",
    age = 22
}

print("Name: ", d.name, " Age: ", d.age) # Used "." based
indexing.

# Indexing

d["mother"] = "Rebecca"
d.mother = "Caroline" # This would work too to create a new
key.
```

For & while

Iterating using the C-style for loop in C-derived languages can be quite complex:

```
const char** strings = new const char*[50];

[..]

for (int i = 0; i < 50; i++) {
            printf("Value: %c Index: %d\n", strings[i], i);
    }

// Even in STL:
std::list<std::string> strings;

[..]

    for (std::string::const_iterator it = strings.begin(); it
!= strings.end(); it++) {
            std::cout << *it << std::endl;
    }
```

Because of this, GDScript makes the opinionated decision to have a for-in loop over iterables instead:

```
for s in strings:
    print(s)
```

Container datatypes (arrays and dictionaries) are iterable. Dictionaries allow iterating the keys:

```
for key in dict:
    print(key, " -> ", dict[key])
```

Iterating with indices is also possible:

```
for i in range(strings.size()):
    print(strings[i])
```

The range() function can take 3 arguments:

```
range(n) # Will count from 0 to n in steps of 1. The
parameter n is exclusive.
range(b, n) # Will count from b to n in steps of 1. The
parameters b is inclusive. The parameter n is exclusive.
range(b, n, s) # Will count from b to n, in steps of s. The
parameters b is inclusive. The parameter n is exclusive.
```

Some examples involving C-style for loops:

```
for (int i = 0; i < 10; i++) {}

for (int i = 5; i < 10; i++) {}

for (int i = 5; i < 10; i += 2) {}
```

Translate to:

```
for i in range(10):
    pass

for i in range(5, 10):
    pass

for i in range(5, 10, 2):
    pass
```

And backwards looping done through a negative counter:

```
for (int i = 10; i > 0; i--) {}
```

Becomes:

```
for i in range(10, 0, -1):
    pass
```

While

while() loops are the same everywhere:

```
var i = 0

while i < strings.size():
    print(strings[i])
    i += 1
```

Custom iterators

You can create custom iterators in case the default ones don't quite meet your needs by overriding the Variant class's `_iter_init`, `_iter_next`, and `_iter_get` functions in your script. An example implementation of a forward iterator follows:

```
class ForwardIterator:
    var start
    var current
    var end
    var increment

    func _init(start, stop, increment):
        self.start = start
        self.current = start
        self.end = stop
        self.increment = increment

    func should_continue():
        return (current < end)

    func _iter_init(arg):
        current = start
        return should_continue()

    func _iter_next(arg):
        current += increment
        return should_continue()

    func _iter_get(arg):
        return current
```

And it can be used like any other iterator:

```
var itr = ForwardIterator.new(0, 6, 2)
for i in itr:
    print(i) # Will print 0, 2, and 4.
```

Make sure to reset the state of the iterator in `_iter_init`, otherwise nested for-loops that use custom iterators will not work as expected.

Duck typing

One of the most difficult concepts to grasp when moving from a statically typed language to a dynamic one is duck typing. Duck typing makes overall code design much simpler and straightforward to write, but it's not obvious how it works.

As an example, imagine a situation where a big rock is falling down a tunnel, smashing everything on its way. The code for the rock, in a statically typed language would be something like:

```
void BigRollingRock::on_object_hit(Smashable *entity) {

    entity->smash();
}
```

This way, everything that can be smashed by a rock would have to inherit Smashable. If a character, enemy, piece of furniture, small rock were all smashable, they would need to inherit from the class Smashable, possibly requiring multiple inheritance. If multiple inheritance was undesired, then they would have to inherit a common class like Entity. Yet, it would not be very elegant to add a virtual method `smash()` to Entity only if a few of them can be smashed.

With dynamically typed languages, this is not a problem. Duck typing makes sure you only have to define a `smash()` function where required and that's it. No need to consider inheritance, base classes, etc.

```
func _on_object_hit(object):
    object.smash()
```

And that's it. If the object that hit the big rock has a smash() method, it will be called. No need for inheritance or polymorphism.

Dynamically typed languages only care about the instance having the desired method or member, not what it inherits or the class type. The definition of Duck Typing should make this clearer:

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"

In this case, it translates to:

"If the object can be smashed, don't care what it is, just smash it."

Yes, we should call it Hulk typing instead.

It's possible that the object being hit doesn't have a smash() function. Some dynamically typed languages simply ignore a method call when it doesn't exist, but GDScript is stricter, so checking if the function exists is desirable:

```
func _on_object_hit(object):
    if object.has_method("smash"):
        object.smash()
```

Then, simply define that method and anything the rock touches can be smashed.

GDScript exported properties

In Godot, class members can be exported. This means their value gets saved along with the resource (such as the scene) they're attached to. They will also be available for editing in the property editor. Exporting is done by using the `@export` annotation.

```
@export var number: int = 5
```

In that example the value `5` will be saved and visible in the property editor.

An exported variable must be initialized to a constant expression or have a type specifier in the variable. Some of the export annotations have a specific type and don't need the variable to be typed (see the *Examples* section below).

One of the fundamental benefits of exporting member variables is to have them visible and editable in the editor. This way, artists and game designers can modify values that later influence how the program runs. For this, a special export syntax is provided.

Note

Exporting properties can also be done in other languages such as C#. The syntax varies depending on the language. See C# exported properties for information on C# exports.

Basic use

If the exported value assigns a constant or constant expression, the type will be inferred and used in the editor.

```
@export var number = 5
```

If there's no default value, you can add a type to the variable.

```
@export var number: int
```

Resources and nodes can be exported.

```
@export var resource: Resource
@export var node: Node
```

Grouping Exports

It is possible to group your exported properties inside the Inspector with the @export_group annotation. Every exported property after this annotation will be added to the group. Start a new group or use `@export_group("")` to break out.

```
@export_group("My Properties")
@export var number = 3
```

The second argument of the annotation can be used to only group properties with the specified prefix.

Groups cannot be nested, use @export_subgroup to create subgroups within a group.

```
@export_subgroup("Extra Properties")
@export var string = ""
@export var flag = false
```

You can also change the name of your main category, or create additional categories in the property list with the @export_category annotation.

```
@export_category("Main Category")
@export var number = 3
@export var string = ""

@export_category("Extra Category")
@export var flag = false
```

Note

The list of properties is organized based on the class inheritance and new categories break that expectation. Use them carefully, especially when creating projects for public use.

Strings as paths

String as a path to a file.

```
@export_file var f
```

String as a path to a directory.

```
@export_dir var f
```

String as a path to a file, custom filter provided as hint.

```
@export_file("*.txt") var f
```

Using paths in the global filesystem is also possible, but only in scripts in tool mode.

String as a path to a PNG file in the global filesystem.

```
@export_global_file("*.png") var tool_image
```

String as a path to a directory in the global filesystem.

```
@export_global_dir var tool_dir
```

The multiline annotation tells the editor to show a large input field for editing over multiple lines.

```
@export_multiline var text
```

Limiting editor input ranges

Allow integer values from 0 to 20.

```
@export_range(0, 20) var i
```

Allow integer values from -10 to 20.

```
@export_range(-10, 20) var j
```

Allow floats from -10 to 20 and snap the value to multiples of 0.2.

```
@export_range(-10, 20, 0.2) var k: float
```

The limits can be only for the slider if you add the hints "or_greater" and/or "or_less".

```
@export_range(0, 100, 1, "or_greater", "or_less")
```

Floats with easing hint

Display a visual representation of the 'ease()' function when editing.

```
@export_exp_easing var transition_speed
```

Colors

Regular color given as red-green-blue-alpha value.

```
@export var col: Color
```

Color given as red-green-blue value (alpha will always be 1).

```
@export_color_no_alpha var col: Color
```

Nodes

Since Godot 4.0, nodes can be directly exported as properties in a script without having to use NodePaths:

```
# Allows any node.
@export var node: Node
```

```
# Allows any node that inherits from BaseButton.
# Custom classes declared with `class_name` can also be used.
@export var some_button: BaseButton
```

Exporting NodePaths like in Godot 3.x is still possible, in case you need it:

```
@export var node_path: NodePath
var node = get_node(node_path)
```

If you want to limit the types of nodes for NodePaths, you can use the @export_node_path annotation:

```
@export_node_path("Button", "TouchScreenButton") var
some_button
```

Resources

```
@export var resource: Resource
```

In the Inspector, you can then drag and drop a resource file from the FileSystem dock into the variable slot.

Opening the inspector dropdown may result in an extremely long list of possible classes to create, however. Therefore, if you specify an extension of Resource such as:

```
@export var resource: AnimationNode
```

The drop-down menu will be limited to AnimationNode and all its inherited classes.

It must be noted that even if the script is not being run while in the editor, the exported properties are still editable. This can be used in conjunction with a script in "tool" mode.

Exporting bit flags

Integers used as bit flags can store multiple `true`/`false` (boolean) values in one property. By using the `@export_flags` annotation, they can be set from the editor:

```
# Set any of the given flags from the editor.
@export_flags("Fire", "Water", "Earth", "Wind") var
spell_elements = 0
```

You must provide a string description for each flag. In this example, `Fire` has value 1, `Water` has value 2, `Earth` has value 4 and `Wind` corresponds to value 8. Usually, constants should be defined accordingly (e.g. `const ELEMENT_WIND = 8` and so on).

You can add explicit values using a colon:

```
@export_flags("Self:4", "Allies:8", "Foes:16") var
spell_targets = 0
```

Only power of 2 values are valid as bit flags options. The lowest allowed value is 1, as 0 means that nothing is selected. You can also add options that are a combination of other flags:

```
@export_flags("Self:4", "Allies:8", "Self and Allies:12",
"Foes:16")
var spell_targets = 0
```

Export annotations are also provided for the physics, render, and navigation layers defined in the project settings:

```
@export_flags_2d_physics var layers_2d_physics
@export_flags_2d_render var layers_2d_render
@export_flags_2d_navigation var layers_2d_navigation
@export_flags_3d_physics var layers_3d_physics
@export_flags_3d_render var layers_3d_render
@export_flags_3d_navigation var layers_3d_navigation
```

Using bit flags requires some understanding of bitwise operations. If in doubt, use boolean variables instead.

Exporting enums

Properties can be exported with a type hint referencing an enum to limit their values to the values of the enumeration. The editor will create a widget in the Inspector, enumerating the following as "Thing 1", "Thing 2", "Another Thing". The value will be stored as an integer.

```
enum NamedEnum {THING_1, THING_2, ANOTHER_THING = -1}
@export var x: NamedEnum
```

Integer and string properties can also be limited to a specific list of values using the @export_enum annotation. The editor will create a widget in the Inspector, enumerating the following as Warrior, Magician, Thief. The value will be stored as an integer, corresponding to the index of the selected option (i.e. `0`, `1`, or `2`).

```
@export_enum("Warrior", "Magician", "Thief") var
character_class: int
```

You can add explicit values using a colon:

```
@export_enum("Slow:30", "Average:60", "Very Fast:200") var
character_speed: int
```

If the type is String, the value will be stored as a string.

```
@export_enum("Rebecca", "Mary", "Leah") var character_name:
String
```

If you want to set an initial value, you must specify it explicitly:

```
@export_enum("Rebecca", "Mary", "Leah") var character_name:
String = "Rebecca"
```

Exporting arrays

Exported arrays can have initializers, but they must be constant expressions.

If the exported array specifies a type which inherits from Resource, the array values can be set in the inspector by dragging and dropping multiple files from the FileSystem dock at once.

The default value **must** be a constant expression.

```
@export var a = [1, 2, 3]
```

Exported arrays can specify type (using the same hints as before).

```
@export var ints: Array[int] = [1, 2, 3]

# Nested typed arrays such as `Array[Array[float]]` are not
supported yet.
@export var two_dimensional: Array[Array] = [[1.0, 2.0],
[3.0, 4.0]]
```

You can omit the default value, but it would then be `null` if not assigned.

```
@export var b: Array
@export var scenes: Array[PackedScene]
```

Arrays with specified types which inherit from resource can be set by drag-and-dropping multiple files from the FileSystem dock.

```
@export var textures: Array[Texture] = []
@export var scenes: Array[PackedScene] = []
```

Packed type arrays also work, but only initialized empty:

```
@export var vector3s = PackedVector3Array()
@export var strings = PackedStringArray()
```

Setting exported variables from a tool script

When changing an exported variable's value from a script in Tool mode, the value in the inspector won't be updated automatically. To update it, call notify_property_list_changed() after setting the exported variable's value.

Advanced exports

Not every type of export can be provided on the level of the language itself to avoid unnecessary design complexity. The following describes some more or less common exporting features which can be implemented with a low-level API.

Before reading further, you should get familiar with the way properties are handled and how they can be customized with _set(), _get(), and _get_property_list() methods as described in Accessing data or logic from an object.

See also

For binding properties using the above methods in C++, see Binding properties using _set/_get/_get_property_list.

Warning

The script must operate in the `tool` mode so the above methods can work from within the editor.

GDScript documentation comments

In GDScript, comments can be used to document your code and add descriptions to the members of a script. There are two differences between a normal comment and a documentation comment. Firstly, a documentation comment should start with double hash symbols `##`. Secondly, it must immediately precede a script member, or for script descriptions, be placed at the top of the script. If an exported variable is documented, its description is used as a tooltip in the editor. This documentation can be generated as XML files by the editor.

Documenting a script

Comments documenting a script must come before any member documentation. A suggested format for script documentation can be divided into three parts.

- A brief description of the script.
- Detailed description.
- Tutorials and deprecated/experimental marks.

To separate these from each other, the documentation comments use special tags. The tag must be at the beginning of a line (ignoring preceding white space) and must have the format `@`, followed by the keyword.

Tags

Brief description	No tag. Lives at the very beginning of the documentation section.
Description	No tag. Use one blank line to separate the description from the brief.

Tutorial	`@tutorial:` `@tutorial(The Title Here):`
Deprecated	`@deprecated`
Experimental	`@experimental`

For example:

```
extends Node2D
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
##
## @tutorial:            https://the/tutorial1/url.com
## @tutorial(Tutorial2): https://the/tutorial2/url.com
## @experimental
```

Warning

If there is any space in between the tag name and colon, for example `@tutorial  :`, it won't be treated as a valid tag and will be ignored.

Note

When the description spans multiple lines, the preceding and trailing white spaces will be stripped and joined with a single space. To preserve the line break use `[br]`. See also BBCode and class reference below.

Documenting script members

Members that are applicable for documentation:

- Inner class
- Constant
- Function
- Signal
- Variable
- Enum
- Enum value

Documentation of a script member must immediately precede the member or its annotations if it has any. The description can have more than one line but every line must start with the double hash symbol ## to be considered as part of the documentation.

Tags

Description	No tag.
Deprecated	`@deprecated`
Experimental	`@experimental`

For example:

```
## The description of the variable.
## @deprecated
var my_var
```

Alternatively, you can use inline documentation comments:

```
enum MyEnum { ## My enum.
    VALUE_A = 0, ## Value A.
    VALUE_B = 1, ## Value B.
}

const MY_CONST = 1 ## My constant.

var my_var ## My variable.

signal my_signal ## My signal.
```

```
func my_func(): ## My func.
    pass

class MyClass: ## My class.
    pass
```

The script documentation will update in the editor help window every time the script is updated. If any member variable or function name starts with an underscore, it will be treated as private. It will not appear in the documentation and will be ignored in the help window.

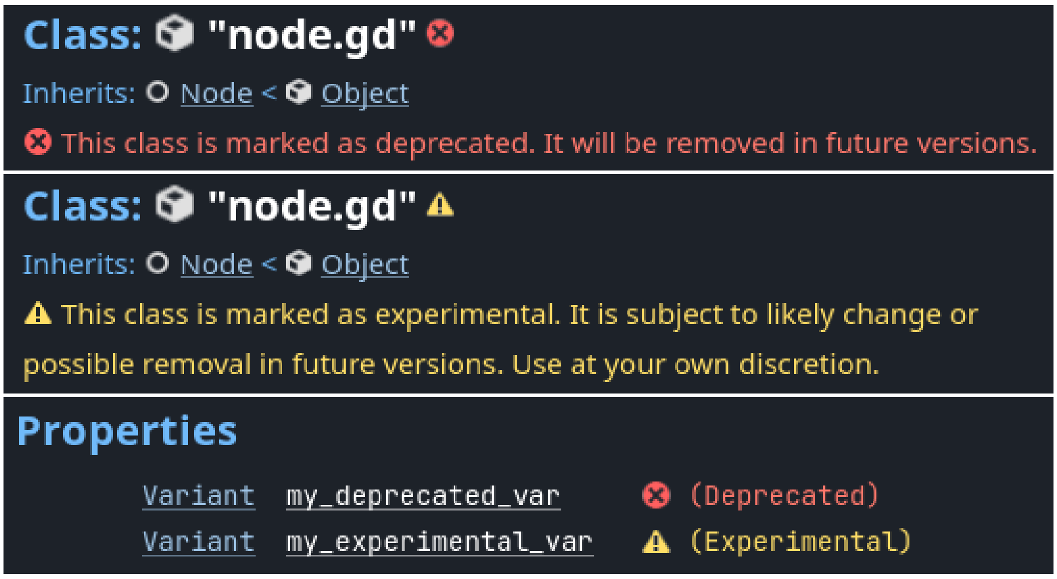
Complete script example

```
extends Node2D
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
##
## @tutorial:            https://the/tutorial1/url.com
## @tutorial(Tutorial2): https://the/tutorial2/url.com
## @experimental

## The description of a constant.
const GRAVITY = 9.8

## The description of a signal.
signal my_signal

## This is a description of the below enum.
enum Direction {
    ## Direction up.
    UP = 0,
    ## Direction down.
    DOWN = 1,
    ## Direction left.
    LEFT = 2,
    ## Direction right.
    RIGHT = 3,
}

## The description of the variable v1.
var v1
```

```
## This is a multiline description of the variable v2.[br]
## The type information below will be extracted for the
documentation.
var v2: int

## If the member has any annotation, the annotation should
## immediately precede it.
@export
var v3 := some_func()


## As the following function is documented, even though its
name starts with
## an underscore, it will appear in the help window.
func _fn(p1: int, p2: String) -> int:
    return 0


# The below function isn't documented and its name starts with
an underscore
# so it will treated as private and will not be shown in the
help window.
func _internal() -> void:
    pass


## Documenting an inner class.
##
## The same rules apply here. The documentation must
## immediately precede the class definition.
##
## @tutorial: https://the/tutorial/url.com
## @experimental
class Inner:

    ## Inner class variable v4.
    var v4


    ## Inner class function fn.
    func fn(): pass
```

`@deprecated` and `@experimental` tags

You can mark a class or any of its members as deprecated or experimental. This will add the corresponding indicator in the built-in documentation viewer. This can be especially useful for plugin and library creators.



Class: "node.gd"

Inherits: Node < Object

This class is marked as deprecated. It will be removed in future versions.

Class: "node.gd"

Inherits: Node < Object

This class is marked as experimental. It is subject to likely change or possible removal in future versions. Use at your own discretion.

Properties

```
Variant  my_deprecated_var      (Deprecated)
Variant  my_experimental_var    (Experimental)
```

- **Deprecated** marks a non-recommended API that is subject to removal or incompatible change in a future major release. Usually the API is kept for backwards compatibility.
- **Experimental** marks a new unstable API that may be changed or removed in the current major branch. Using this API is not recommended in production code.

Note

While technically you can use both `@deprecated` and `@experimental` tags on the same class/member, this is not recommended as it is against common conventions.

BBCode and class reference

The editor help window which renders the documentation supports bbcode. As a result it's possible to align and format the documentation. Color texts, images, fonts, tables, URLs, animation effects, etc. can be added with the bbcode.

Godot's class reference supports BBCode-like tags. They add nice formatting to the text which could also be used in the documentation.

See also class reference bbcode.

Whenever you link to a member of another class, you need to specify the class name. For links to the same class, the class name is optional and can be omitted.

Here's the list of available tags:

Tag and Description	Example	Result
`[Class]` Link to class	`Move the [Sprite2D].`	Move the Sprite2D.
`[annotation Class.name]` Link to annotation	`See [annotation @GDScript.@export].`	See @GDScript.@export.
`[constant Class.name]` Link to constant	`See [constant @GlobalScope.KEY_F1].`	See @GlobalScope.KEY_F1.
`[enum Class.name]` Link to enum	`See [enum Mesh.ArrayType].`	See Mesh.ArrayType.
`[method Class.name]` Link to method	`Call [method Node3D.hide].`	Call Node3D.hide().
`[member Class.name]` Link to member	`Get [member Node2D.scale].`	Get Node2D.scale.

Tag and Description	Example	Result
`[signal Class.name]` Link to signal	`Emit [signal Node.renamed].`	Emit Node.renamed.
`[theme_item Class.name]` Link to theme item	`See [theme_item Label.font].`	See Label.font.
`[param name]` Formats a parameter name (as code)	`Takes [param size] for the size.`	Takes `size` for the size.
`[br]` Line break	`Line 1.[br] Line 2.`	Line 1. Line 2.
`[b] [/b]` Bold	`Some [b]bold[/b] text.`	Some **bold** text.
`[i] [/i]` Italic	`Some [i]italic[/i] text.`	Some *italic* text.
`[kbd] [/kbd]` Keyboard/mouse shortcut	`Some [kbd]Ctrl + C[/kbd] key.`	Some `Ctrl + C` key.
`[code] [/code]` Monospace	`Some [code]monospace[/code] text.`	Some `monospace` text.

Tag and Description	Example	Result
`[codeblock]` `[/codeblock]` Multiline preformatted block	*See below.*	*See below.*

Note

1. Currently only [@GDScript] has annotations.
2. `[code]` disables BBCode until the parser encounters `[/code]`.
3. `[codeblock]` disables BBCode until the parser encounters `[/codeblock]`.

Warning

Use `[codeblock]` for pre-formatted code blocks. Inside `[codeblock]`, always use **four spaces** for indentation (the parser will delete tabs).

```
## Do something for this plugin. Before using the method
## you first have to [method initialize] [MyPlugin].[br]
## [color=yellow]Warning:[/color] Always [method clean] after
use.[br]
## Usage:
## [codeblock]
## func _ready():
##     the_plugin.initialize()
##     the_plugin.do_something()
##     the_plugin.clean()
## [/codeblock]
func do_something():
    pass
```

GDScript style guide

This style guide lists conventions to write elegant GDScript. The goal is to encourage writing clean, readable code and promote consistency across projects, discussions, and tutorials. Hopefully, this will also support the development of auto-formatting tools.

Since GDScript is close to Python, this guide is inspired by Python's PEP 8 [https://www.python.org/dev/peps/pep-0008/] programming style guide.

Style guides aren't meant as hard rulebooks. At times, you may not be able to apply some of the guidelines below. When that happens, use your best judgment, and ask fellow developers for insights.

In general, keeping your code consistent in your projects and within your team is more important than following this guide to a tee.

Note

Godot's built-in script editor uses a lot of these conventions by default. Let it help you.

Here is a complete class example based on these guidelines:

```
class_name StateMachine
extends Node
## Hierarchical State machine for the player.
##
## Initializes states and delegates engine callbacks ([method
Node._physics_process],
## [method Node._unhandled_input]) to the state.


signal state_changed(previous, new)

@export var initial_state: Node
```

```
var is_active = true:
    set = set_is_active

@onready var _state = initial_state:
    set = set_state
@onready var _state_name = _state.name


func _init():
    add_to_group("state_machine")


func _enter_tree():
    print("this happens before the ready method!")


func _ready():
    state_changed.connect(_on_state_changed)
    _state.enter()


func _unhandled_input(event):
    _state.unhandled_input(event)


func _physics_process(delta):
    _state.physics_process(delta)


func transition_to(target_state_path, msg={}):
    if not has_node(target_state_path):
        return

    var target_state = get_node(target_state_path)
    assert(target_state.is_composite == false)

    _state.exit()
    self._state = target_state
    _state.enter(msg)
    Events.player_state_changed.emit(_state.name)


func set_is_active(value):
    is_active = value
    set_physics_process(value)
    set_process_unhandled_input(value)
    set_block_signals(not value)


func set_state(value):
    _state = value
    _state_name = _state.name


func _on_state_changed(previous, new):
    print("state changed")
    state_changed.emit()


class State:
    var foo = 0

    func _init():
        print("Hello!")
```

Formatting

Encoding and special characters

- Use line feed (**LF**) characters to break lines, not CRLF or CR. *(editor default)*
- Use one line feed character at the end of each file. *(editor default)*
- Use **UTF-8** encoding without a byte order mark [https://en.wikipedia.org/wiki/Byte_order_mark]. *(editor default)*
- Use **Tabs** instead of spaces for indentation. *(editor default)*

Indentation

Each indent level should be one greater than the block containing it.

Good:

```
for i in range(10):
    print("hello")
```

Bad:

```
for i in range(10):
  print("hello")

for i in range(10):
        print("hello")
```

Use 2 indent levels to distinguish continuation lines from regular code blocks.

Good:

```
effect.interpolate_property(sprite, "transform/scale",
        sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
        Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Bad:

```
effect.interpolate_property(sprite, "transform/scale",
    sprite.get_scale(), Vector2(2.0, 2.0), 0.3,
    Tween.TRANS_QUAD, Tween.EASE_OUT)
```

Exceptions to this rule are arrays, dictionaries, and enums. Use a single indentation level to distinguish continuation lines:

Good:

```
var party = [
    "Godot",
    "Godette",
    "Steve",
]

var character_dict = {
    "Name": "Bob",
    "Age": 27,
    "Job": "Mechanic",
}

enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT,
}
```

Bad:

```
var party = [
        "Godot",
        "Godette",
        "Steve",
]

var character_dict = {
        "Name": "Bob",
        "Age": 27,
        "Job": "Mechanic",
}

enum Tiles {
        TILE_BRICK,
        TILE_FLOOR,
        TILE_SPIKE,
        TILE_TELEPORT,
}
```

Trailing comma

Use a trailing comma on the last line in arrays, dictionaries, and enums. This results in easier refactoring and better diffs in version control as the last line doesn't need to be modified when adding new elements.

Good:

```
enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT,
}
```

Bad:

```
enum Tiles {
    TILE_BRICK,
    TILE_FLOOR,
    TILE_SPIKE,
    TILE_TELEPORT
}
```

Trailing commas are unnecessary in single-line lists, so don't add them in this case.

Good:

```
enum Tiles {TILE_BRICK, TILE_FLOOR, TILE_SPIKE,
TILE_TELEPORT}
```

Bad:

```
enum Tiles {TILE_BRICK, TILE_FLOOR, TILE_SPIKE,
TILE_TELEPORT,}
```

Blank lines

Surround functions and class definitions with two blank lines:

```
func heal(amount):
    health += amount
    health = min(health, max_health)
    health_changed.emit(health)


func take_damage(amount, effect=null):
    health -= amount
    health = max(0, health)
    health_changed.emit(health)
```

Use one blank line inside functions to separate logical sections.

Note

We use a single line between classes and function definitions in the class reference and in short code snippets in this documentation.

Line length

Keep individual lines of code under 100 characters.

If you can, try to keep lines under 80 characters. This helps to read the code on small displays and with two scripts opened side-by-side in an external text editor. For example, when looking at a differential revision.

One statement per line

Never combine multiple statements on a single line. No, C programmers, not even with a single line conditional statement.

Good:

```
if position.x > width:
    position.x = 0

if flag:
    print("flagged")
```

Bad:

```
if position.x > width: position.x = 0

if flag: print("flagged")
```

The only exception to that rule is the ternary operator:

```
next_state = "idle" if is_on_floor() else "fall"
```

Format multiline statements for readability

When you have particularly long `if` statements or nested ternary expressions, wrapping them over multiple lines improves readability. Since continuation lines are still part of the same expression, 2 indent levels should be used instead of one.

GDScript allows wrapping statements using multiple lines using parentheses or backslashes. Parentheses are favored in this style guide since they make for easier refactoring. With backslashes, you have to ensure that the last line never contains a backslash at the end. With parentheses, you don't have to worry about the last line having a backslash at the end.

When wrapping a conditional expression over multiple lines, the `and`/`or` keywords should be placed at the beginning of the line continuation, not at the end of the previous line.

Good:

```
var angle_degrees = 135
var quadrant = (
        "northeast" if angle_degrees <= 90
        else "southeast" if angle_degrees <= 180
        else "southwest" if angle_degrees <= 270
        else "northwest"
)

var position = Vector2(250, 350)
if (
        position.x > 200 and position.x < 400
        and position.y > 300 and position.y < 400
):
    pass
```

Bad:

```
var angle_degrees = 135
var quadrant = "northeast" if angle_degrees <= 90 else
"southeast" if angle_degrees <= 180 else "southwest" if
angle_degrees <= 270 else "northwest"

var position = Vector2(250, 350)
if position.x > 200 and position.x < 400 and position.y >
```

```
300 and position.y < 400:
    pass
```

Avoid unnecessary parentheses

Avoid parentheses in expressions and conditional statements. Unless necessary for order of operations or wrapping over multiple lines, they only reduce readability.

Good:

```
if is_colliding():
    queue_free()
```

Bad:

```
if (is_colliding()):
    queue_free()
```

Boolean operators

Prefer the plain English versions of boolean operators, as they are the most accessible:

- Use `and` instead of `&&`.
- Use `or` instead of `||`.
- Use `not` instead of `!`.

You may also use parentheses around boolean operators to clear any ambiguity. This can make long expressions easier to read.

Good:

```
if (foo and bar) or not baz:
    print("condition is true")
```

Bad:

```
if foo && bar || !baz:
    print("condition is true")
```

Comment spacing

Regular comments (`#`) and documentation comments (`##`) should start with a space, but not code that you comment out. Additionally, code region comments (`#region`/`#endregion`) must follow that precise syntax, so they should not start with a space.

Using a space for regular and documentation comments helps differentiate text comments from disabled code.

Good:

```
# This is a comment.
#print("This is disabled code")
```

Bad:

```
#This is a comment.
# print("This is disabled code")
```

Note

In the script editor, to toggle the selected code commented, press `Ctrl + K`. This feature adds a single `#` sign at the start of the selected lines.

Whitespace

Always use one space around operators and after commas. Also, avoid extra spaces in dictionary references and function calls.

Good:

```
position.x = 5
position.y = target_position.y + 10
dict["key"] = 5
my_array = [4, 5, 6]
print("foo")
```

Bad:

```
position.x=5
position.y = mpos.y+10
dict ["key"] = 5
myarray = [4,5,6]
print ("foo")
```

Don't use spaces to align expressions vertically:

```
x        = 100
y        = 100
velocity = 500
```

Quotes

Use double quotes unless single quotes make it possible to escape fewer characters in a given string. See the examples below:

```
# Normal string.
print("hello world")

# Use double quotes as usual to avoid escapes.
print("hello 'world'")

# Use single quotes as an exception to the rule to avoid
escapes.
print('hello "world"')

# Both quote styles would require 2 escapes; prefer double
quotes if it's a tie.
print("'hello' \"world\"")
```

Numbers

Don't omit the leading or trailing zero in floating-point numbers. Otherwise, this makes them less readable and harder to distinguish from integers at a glance.

Good:

```
var float_number = 0.234
var other_float_number = 13.0
```

Bad:

```
var float_number = .234
var other_float_number = 13.
```

Use lowercase for letters in hexadecimal numbers, as their lower height makes the number easier to read.

Good:

```
var hex_number = 0xfb8c0b
```

Bad:

```
var hex_number = 0xFB8C0B
```

Take advantage of GDScript's underscores in literals to make large numbers more readable.

Good:

```
var large_number = 1_234_567_890
var large_hex_number = 0xffff_f8f8_0000
var large_bin_number = 0b1101_0010_1010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12345
```

Bad:

```
var large_number = 1234567890
var large_hex_number = 0xffffff8f80000
var large_bin_number = 0b110100101010
# Numbers lower than 1000000 generally don't need separators.
var small_number = 12_345
```

Naming conventions

These naming conventions follow the Godot Engine style. Breaking these will make your code clash with the built-in naming conventions, leading to inconsistent code.

File names

Use snake_case for file names. For named classes, convert the PascalCase class name to snake_case:

```
# This file should be saved as `weapon.gd`.
class_name Weapon
extends Node
```

```
# This file should be saved as `yaml_parser.gd`.
class_name YAMLParser
extends Object
```

This is consistent with how C++ files are named in Godot's source code. This also avoids case sensitivity issues that can crop up when exporting a project from Windows to other platforms.

Classes and nodes

Use PascalCase for class and node names:

```
extends CharacterBody3D
```

Also use PascalCase when loading a class into a constant or a variable:

```
const Weapon = preload("res://weapon.gd")
```

Functions and variables

Use snake_case to name functions and variables:

```
var particle_effect
func load_level():
```

Prepend a single underscore (_) to virtual methods functions the user must override, private functions, and private variables:

```
var _counter = 0
func _recalculate_path():
```

Signals

Use the past tense to name signals:

```
signal door_opened
signal score_changed
```

Constants and enums

Write constants with CONSTANT_CASE, that is to say in all caps with an underscore (_) to separate words:

```
const MAX_SPEED = 200
```

Use PascalCase for enum *names* and CONSTANT_CASE for their members, as they are constants:

```
enum Element {
    EARTH,
    WATER,
    AIR,
    FIRE,
}
```

Code order

This first section focuses on code order. For formatting, see Formatting. For naming conventions, see Naming conventions.

We suggest to organize GDScript code this way:

```
01. @tool
02. class_name
03. extends
04. # docstring

05. signals
```

```
06. enums
07. constants
08. @export variables
09. public variables
10. private variables
11. @onready variables

12. optional built-in virtual _init method
13. optional built-in virtual _enter_tree() method
14. built-in virtual _ready method
15. remaining built-in virtual methods
16. public methods
17. private methods
18. subclasses
```

We optimized the order to make it easy to read the code from top to bottom, to help developers reading the code for the first time understand how it works, and to avoid errors linked to the order of variable declarations.

This code order follows four rules of thumb:

1. Properties and signals come first, followed by methods.
2. Public comes before private.
3. Virtual callbacks come before the class's interface.
4. The object's construction and initialization functions, `_init` and `_ready`, come before functions that modify the object at runtime.

Class declaration

If the code is meant to run in the editor, place the `@tool` annotation on the first line of the script.

Follow with the `class_name` if necessary. You can turn a GDScript file into a global type in your project using this feature. For more information, see GDScript reference.

Then, add the `extends` keyword if the class extends a built-in type.

Following that, you should have the class's optional documentation comments. You can use that to explain the role of your class to your teammates, how it works, and how other developers should use it, for example.

```
class_name MyNode
extends Node
## A brief description of the class's role and functionality.
##
## The description of the script, what it can do,
## and any further detail.
```

Signals and properties

Write signal declarations, followed by properties, that is to say, member variables, after the docstring.

Enums should come after signals, as you can use them as export hints for other properties.

Then, write constants, exported variables, public, private, and onready variables, in that order.

```
signal player_spawned(position)

enum Jobs {KNIGHT, WIZARD, ROGUE, HEALER, SHAMAN}

const MAX_LIVES = 3

@export var job: Jobs = Jobs.KNIGHT
@export var max_health = 50
@export var attack = 5

var health = max_health:
    set(new_health):
        health = new_health

var _speed = 300.0

@onready var sword = get_node("Sword")
@onready var gun = get_node("Gun")
```

Note

The GDScript compiler evaluates onready variables right before the `_ready` callback. You can use that to cache node dependencies, that is to say, to get child nodes in the scene that your class relies on. This is what the example above shows.

Member variables

Don't declare member variables if they are only used locally in a method, as it makes the code more difficult to follow. Instead, declare them as local variables in the method's body.

Local variables

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Methods and static functions

After the class's properties come the methods.

Start with the `_init()` callback method, that the engine will call upon creating the object in memory. Follow with the `_ready()` callback, that Godot calls when it adds a node to the scene tree.

These functions should come first because they show how the object is initialized.

Other built-in virtual callbacks, like `_unhandled_input()` and `_physics_process`, should come next. These control the object's main loop and interactions with the game engine.

The rest of the class's interface, public and private methods, come after that, in that order.

```
func _init():
    add_to_group("state_machine")


func _ready():
    state_changed.connect(_on_state_changed)
    _state.enter()


func _unhandled_input(event):
    _state.unhandled_input(event)


func transition_to(target_state_path, msg={}):
    if not has_node(target_state_path):
        return

    var target_state = get_node(target_state_path)
    assert(target_state.is_composite == false)

    _state.exit()
    self._state = target_state
    _state.enter(msg)
    Events.player_state_changed.emit(_state.name)


func _on_state_changed(previous, new):
    print("state changed")
    state_changed.emit()
```

Static typing

Since Godot 3.1, GDScript supports optional static typing.

Declared types

To declare a variable's type, use `<variable>: <type>`:

```
var health: int = 0
```

To declare the return type of a function, use `-> <type>`:

```
func heal(amount: int) -> void:
```

Inferred types

In most cases you can let the compiler infer the type, using `:=`. Prefer `:=` when the type is written on the same line as the assignment, otherwise prefer writing the type explicitly.

Good:

```
var health: int = 0 # The type can be int or float, and thus
should be stated explicitly.
var direction := Vector3(1, 2, 3) # The type is clearly
inferred as Vector3.
```

Include the type hint when the type is ambiguous, and omit the type hint when it's redundant.

Bad:

```
var health := 0 # Typed as int, but it could be that float
was intended.
var direction: Vector3 = Vector3(1, 2, 3) # The type hint
has redundant information.

# What type is this? It's not immediately clear to the
reader, so it's bad.
var value := complex_function()
```

In some cases, the type must be stated explicitly, otherwise the behavior will not be as expected because the compiler will only be able to use the function's return type. For example, `get_node()` cannot infer a type unless the scene or file of the node is loaded in memory. In this case, you should set the type explicitly.

Good:

```
@onready var health_bar: ProgressBar = get_node("UI/LifeBar")
```

Alternatively, you can use the `as` keyword to cast the return type, and that type will be used to infer the type of the var.

```
@onready var health_bar := get_node("UI/LifeBar") as
ProgressBar
# health_bar will be typed as ProgressBar
```

This option is also considered more type-safe than the first.

Bad:

```
# The compiler can't infer the exact type and will use Node
# instead of ProgressBar.
@onready var health_bar := get_node("UI/LifeBar")
```

Static typing in GDScript

In this guide, you will learn:

- how to use static typing in GDScript;
- that static types can help you avoid bugs;
- that static typing improves your experience with the editor.

Where and how you use this language feature is entirely up to you: you can use it only in some sensitive GDScript files, use it everywhere, or don't use it at all.

Static types can be used on variables, constants, functions, parameters, and return types.

A brief look at static typing

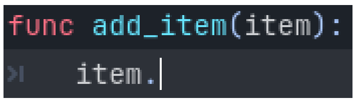
With static typing, GDScript can detect more errors without even running the code. Also type hints give you and your teammates more information as you're working, as the arguments' types show up when you call a method. Static typing improves editor autocompletion and documentation of your scripts.

Imagine you're programming an inventory system. You code an `Item` class, then an `Inventory`. To add items to the inventory, the people who work with your code should always pass an `Item` to the `Inventory.add()` method. With types, you can enforce this:

```
class_name Inventory


func add(reference: Item, amount: int = 1):
    var item := find_item(reference)
    if not item:
        item = _instance_item_from_db(reference)
    item.amount += amount
```

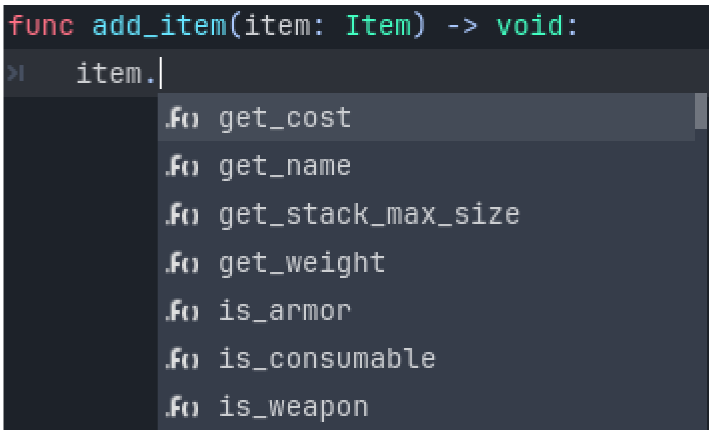
Static types also give you better code completion options. Below, you can see the difference between a dynamic and a static typed completion options.

You've probably encountered a lack of autocomplete suggestions after a dot:



```
func add_item(item):
    item.
```

This is due to dynamic code. Godot cannot know what value type you're passing to the function. If you write the type explicitly however, you will get all methods, properties, constants, etc. from the value:



```
func add_item(item: Item) -> void:
    item.
        get_cost
        get_name
        get_stack_max_size
        get_weight
        is_armor
        is_consumable
        is_weapon
```

Tip

If you prefer static typing, we recommend enabling the **Text Editor > Completion > Add Type Hints** editor setting. Also consider enabling some warnings that are disabled by default.

Also, typed GDScript improves performance by using optimized opcodes when operand/argument types are known at compile time. More GDScript optimizations are planned in the future, such as JIT/AOT compilation.

Overall, typed programming gives you a more structured experience. It helps prevent errors and improves the self-documenting aspect of your scripts. This is especially helpful when you're working in a team or on a long-term project: studies have shown that developers spend most of their time reading other people's code, or scripts they wrote in the past and forgot about. The clearer and the more structured the code, the faster it is to understand, the faster you can move forward.

How to use static typing

To define the type of a variable, parameter, or constant, write a colon after the name, followed by its type. E.g. `var health: int`. This forces the variable's type to always stay the same:

```
var damage: float = 10.5
const MOVE_SPEED: float = 50.0
func sum(a: float = 0.0, b: float = 0.0) -> float:
    return a + b
```

Godot will try to infer types if you write a colon, but you omit the type:

```
var damage := 10.5
const MOVE_SPEED := 50.0
func sum(a := 0.0, b := 0.0) -> float:
    return a + b
```

Note

1. There is no difference between `=` and `:=` for constants.
2. You don't need to write type hints for constants, as Godot sets it automatically from the assigned value. But you can still do so to make the intent of your code clearer. Also, this is useful for typed arrays (like `const A: Array[int] = [1, 2, 3]`), since untyped arrays are used by default.

What can be a type hint

Here is a complete list of what can be used as a type hint:

1. `Variant`. Any type. In most cases this is not much different from an untyped declaration, but increases readability. As a return type, forces the function to explicitly return some value.
2. *(Only return type)* `void`. Indicates that the function does not return any value.
3. Built-in types.
4. Native classes (`Object`, `Node`, `Area2D`, `Camera2D`, etc.).
5. Global classes.
6. Inner classes.
7. Global, native and custom named enums. Note that an enum type is just an `int`, there is no guarantee that the value belongs to the set of enum values.
8. Constants (including local ones) if they contain a preloaded class or enum.

You can use any class, including your custom classes, as types. There are two ways to use them in scripts. The first method is to preload the script you want to use as a type in a constant:

```
const Rifle = preload("res://player/weapons/rifle.gd")
var my_rifle: Rifle
```

The second method is to use the `class_name` keyword when you create. For the example above, your `rifle.gd` would look like this:

```
class_name Rifle
extends Node2D
```

If you use `class_name`, Godot registers the `Rifle` type globally in the editor, and you can use it anywhere, without having to preload it into a constant:

```
var my_rifle: Rifle
```

Specify the return type of a function with the arrow ->

To define the return type of a function, write a dash and a right angle bracket -> after its declaration, followed by the return type:

```gdscript
func _process(delta: float) -> void:
    pass
```

The type `void` means the function does not return anything. You can use any type, as with variables:

```gdscript
func hit(damage: float) -> bool:
    health_points -= damage
    return health_points <= 0
```

You can also use your own classes as return types:

```gdscript
# Adds an item to the inventory and returns it.
func add(reference: Item, amount: int) -> Item:
    var item: Item = find_item(reference)
    if not item:
        item = ItemDatabase.get_instance(reference)

    item.amount += amount
    return item
```

Covariance and contravariance

When inheriting base class methods, you should follow the Liskov substitution principle [https://en.wikipedia.org/wiki/Liskov_substitution_principle].

Covariance: When you inherit a method, you can specify a return type that is more specific (**subtype**) than the parent method.

Contravariance: When you inherit a method, you can specify a parameter type that is less specific (**supertype**) than the parent method.

Example:

```gdscript
class_name Parent


func get_property(param: Label) -> Node:
    # ...
```

```
class_name Child extends Parent


# `Control` is a supertype of `Label`.
# `Node2D` is a subtype of `Node`.
func get_property(param: Control) -> Node2D:
    # ...
```

Specify the element type of an `Array`

To define the type of an `Array`, enclose the type name in `[]`.

An array's type applies to `for` loop variables, as well as some operators like `[]`, `[]=`, and `+`. Array methods (such as `push_back`) and other operators (such as `==`) are still untyped. Built-in types, native and custom classes, and enums may be used as element types. Nested array types are not supported.

```
var scores: Array[int] = [10, 20, 30]
var vehicles: Array[Node] = [$Car, $Plane]
var items: Array[Item] = [Item.new()]
# var arrays: Array[Array] -- disallowed

for score in scores:
    # score has type `int`

# The following would be errors:
scores += vehicles
var s: String = scores[0]
scores[0] = "lots"
```

Since Godot 4.2, you can also specify a type for the loop variable in a `for` loop. For instance, you can write:

```
var names = ["John", "Marta", "Samantha", "Jimmy"]
for name: String in names:
    pass
```

The array will remain untyped, but the `name` variable within the `for` loop will always be of `String` type.

Type casting

Type casting is an important concept in typed languages. Casting is the conversion of a value from one type to another.

Imagine an `Enemy` in your game, that `extends Area2D`. You want it to collide with the `Player`, a `CharacterBody2D` with a script called `PlayerController` attached to it. You use the `body_entered` signal to detect the collision. With typed code, the body you detect is going to be a generic `PhysicsBody2D`, and not your `PlayerController` on the `_on_body_entered` callback.

You can check if this `PhysicsBody2D` is your `Player` with the `as` keyword, and using the colon `:` again to force the variable to use this type. This forces the variable to stick to the `PlayerController` type:

```
func _on_body_entered(body: PhysicsBody2D) -> void:
    var player := body as PlayerController
    if not player:
        return

    player.damage()
```

As we're dealing with a custom type, if the `body` doesn't extend `PlayerController`, the `player` variable will be set to `null`. We can use this to check if the body is the player or not. We will also get full autocompletion on the player variable thanks to that cast.

Note

The `as` keyword silently casts the variable to `null` in case of a type mismatch at runtime, without an error/warning. While this may be convenient in some cases, it can also lead to bugs. Use the `as` keyword only if this behavior is intended. A safer alternative is to use the `is` keyword:

```
if not (body is PlayerController):
    push_error("Bug: body is not PlayerController.")

var player: PlayerController = body
if not player:
    return
```

```
player.damage()
```

or `assert()` statement:

```
assert(body is PlayerController, "Bug: body is not
PlayerController.")

var player: PlayerController = body
if not player:
    return

player.damage()
```

Note

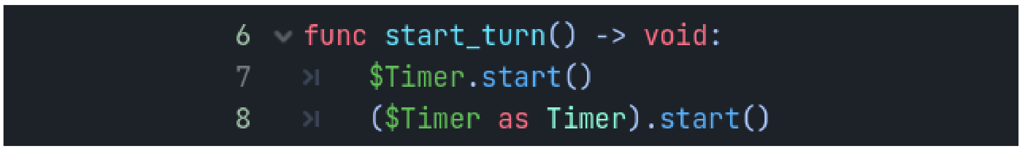
If you try to cast with a built-in type and it fails, Godot will throw an error.

Safe lines

You can also use casting to ensure safe lines. Safe lines are a tool to tell you when ambiguous lines of code are type-safe. As you can mix and match typed and dynamic code, at times, Godot doesn't have enough information to know if an instruction will trigger an error or not at runtime.

This happens when you get a child node. Let's take a timer for example: with dynamic code, you can get the node with `$Timer`. GDScript supports duck-typing [https://stackoverflow.com/a/4205163/8125343], so even if your timer is of type `Timer`, it is also a `Node` and an `Object`, two classes it extends. With dynamic GDScript, you also don't care about the node's type as long as it has the methods you need to call.

You can use casting to tell Godot the type you expect when you get a node: `($Timer as Timer)`, `($Player as CharacterBody2D)`, etc. Godot will ensure the type works and if so, the line number will turn green at the left of the script editor.



```
6  func start_turn() -> void:
7      $Timer.start()
8      ($Timer as Timer).start()
```

Unsafe line (line 7) vs Safe Lines (line 6 and 8)

Note

Safe lines do not always mean better or more reliable code. See the note above about the `as` keyword. For example:

```
@onready var node_1 := $Node1 as Type1 # Safe line.
@onready var node_2: Type2 = $Node2 # Unsafe line.
```

Even though `node_2` declaration is marked as an unsafe line, it is more reliable than `node_1` declaration. Because if you change the node type in the scene and accidentally forget to change it in the script, the error will be detected immediately when the scene is loaded. Unlike `node_1`, which will be silently cast to `null` and the error will be detected later.

Note

You can turn off safe lines or change their color in the editor settings.

Typed or dynamic: stick to one style

Typed GDScript and dynamic GDScript can coexist in the same project. But it's recommended to stick to either style for consistency in your codebase, and for your peers. It's easier for everyone to work together if you follow the same guidelines, and faster to read and understand other people's code.

Typed code takes a little more writing, but you get the benefits we discussed above. Here's an example of the same, empty script, in a dynamic style:

```
extends Node


func _ready():
    pass


func _process(delta):
    pass
```

And with static typing:

```
extends Node


func _ready() -> void:
    pass


func _process(delta: float) -> void:
    pass
```

As you can see, you can also use types with the engine's virtual methods. Signal callbacks, like any methods, can also use types. Here's a `body_entered` signal in a dynamic style:

```
func _on_area_2d_body_entered(body):
    pass
```

And the same callback, with type hints:

```
func _on_area_entered(area: CollisionObject2D) -> void:
    pass
```

Warning system

Note

Detailed documentation about the GDScript warning system has been moved to GDScript warning system.

From version 3.1, Godot gives you warnings about your code as you write it: the engine identifies sections of your code that may lead to issues at runtime, but lets you decide whether or not you want to leave the code as it is.

We have a number of warnings aimed specifically at users of typed GDScript. By default, these warnings are disabled, you can enable them in Project Settings (**Debug > GDScript**, make sure **Advanced Settings** is enabled).

You can enable the `UNTYPED_DECLARATION` warning if you want to always use static types. Additionally, you can enable the `INFERRED_DECLARATION` warning if you prefer a more readable and reliable, but more verbose syntax.

`UNSAFE_*` warnings make unsafe operations more noticeable, than unsafe lines. Currently, `UNSAFE_*` warnings do not cover all cases that unsafe lines cover.

Cases where you can't specify types

To wrap up this introduction, let's mention cases where you can't use type hints. This will trigger a **syntax error**.

1. You can't specify the type of individual elements in an array or a dictionary:

```
var enemies: Array = [$Goblin: Enemy, $Zombie: Enemy]
var character: Dictionary = {
    name: String = "Richard",
    money: int = 1000,
    inventory: Inventory = $Inventory,
}
```

2. Nested types are not currently supported:

```
var teams: Array[Array[Character]] = []
```

3. Typed dictionaries are not currently supported:

```
var map: Dictionary[Vector2i, Item] = {}
```

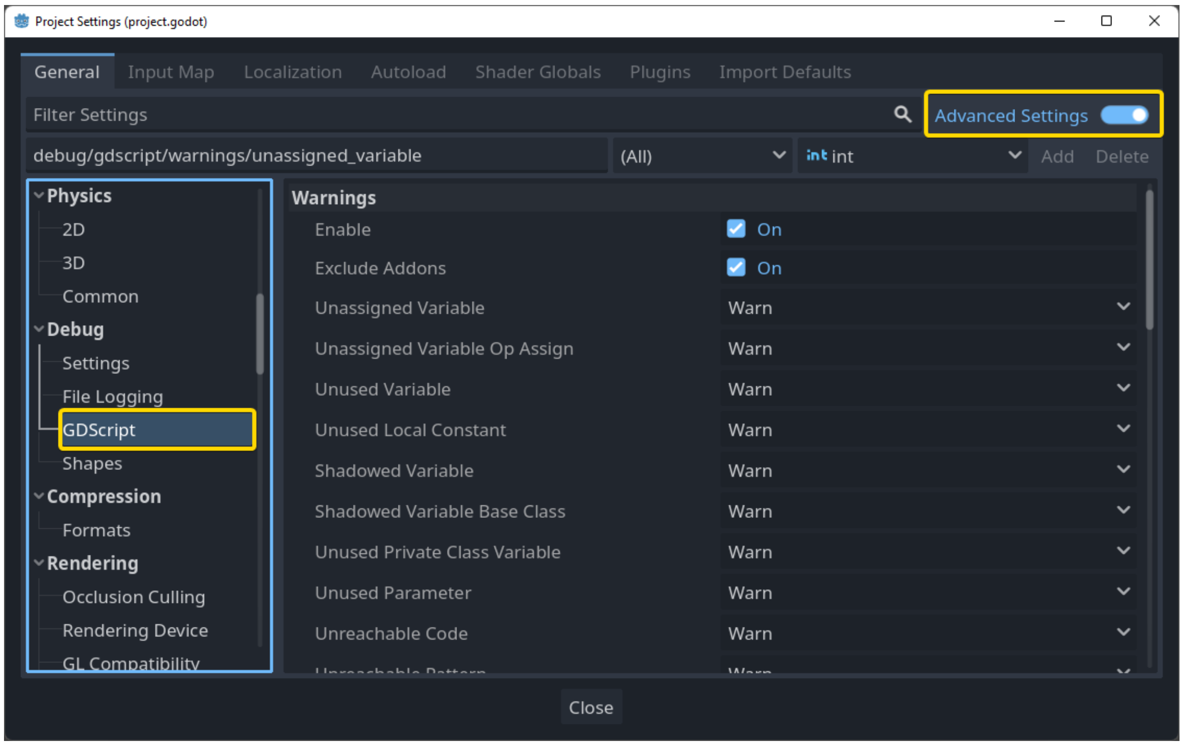
Summary

Typed GDScript is a powerful tool. It helps you write more structured code, avoid common errors, and create scalable and reliable systems. Static types improve GDScript performance and more optimizations are planned for the future.

GDScript warning system

The GDScript warning system complements static typing (but it can work without static typing too). It's here to help you avoid mistakes that are hard to spot during development, and that may lead to runtime errors.

You can configure warnings in the Project Settings under the section called **GDScript**:



debug/gdscript/warnings/unassigned_variable

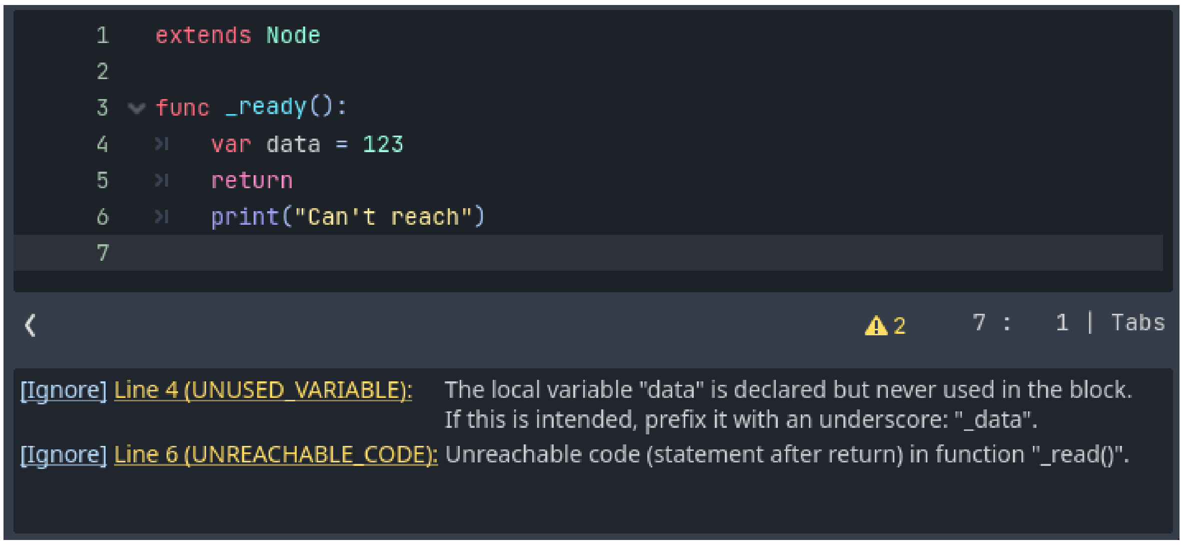
Warnings

Enable	On
Exclude Addons	On
Unassigned Variable	Warn
Unassigned Variable Op Assign	Warn
Unused Variable	Warn
Unused Local Constant	Warn
Shadowed Variable	Warn
Shadowed Variable Base Class	Warn
Unused Private Class Variable	Warn
Unused Parameter	Warn
Unreachable Code	Warn
Unreachable Pattern	Warn

Note

You must enable **Advanced Settings** in order to see the GDScript section in the sidebar. You can also search for "GDScript" when Advanced Settings is off.

You can find a list of warnings for the active GDScript file in the script editor's status bar. The example below has 2 warnings:

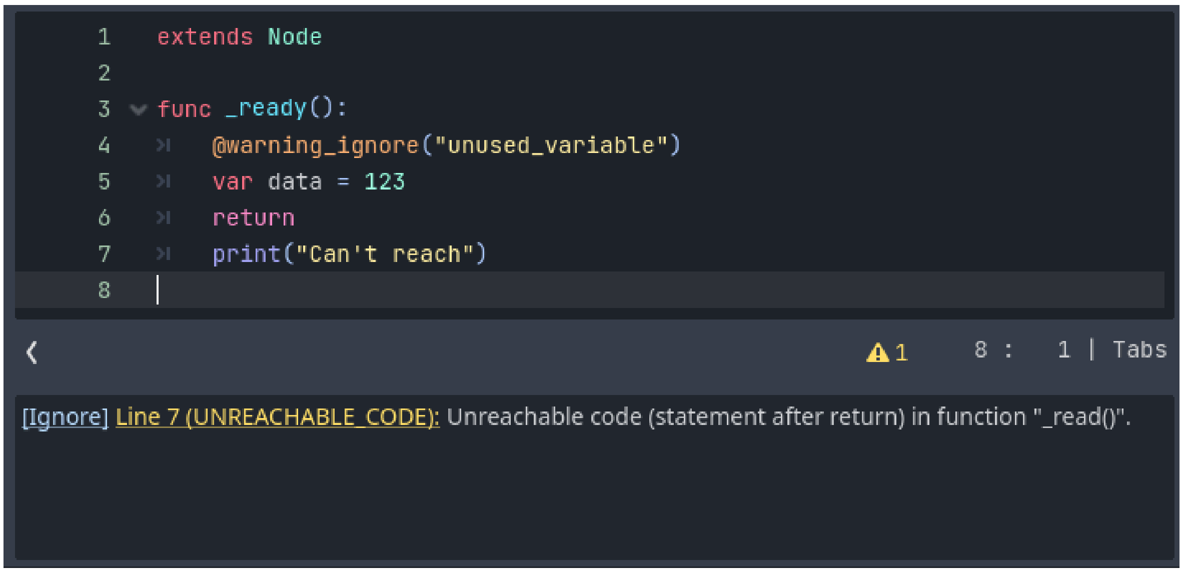


```
extends Node

func _ready():
	var data = 123
	return
	print("Can't reach")

```

⚠2   7 :   1 | Tabs

[Ignore] Line 4 (UNUSED_VARIABLE): The local variable "data" is declared but never used in the block. If this is intended, prefix it with an underscore: "_data".
[Ignore] Line 6 (UNREACHABLE_CODE): Unreachable code (statement after return) in function "_read()".

To ignore specific warnings in one file, insert an annotation of the form `@warning_ignore("warning-id")`, or click on the ignore link to the left of the warning's description. Godot will add an annotation above the corresponding line and the code won't trigger the corresponding warning anymore:

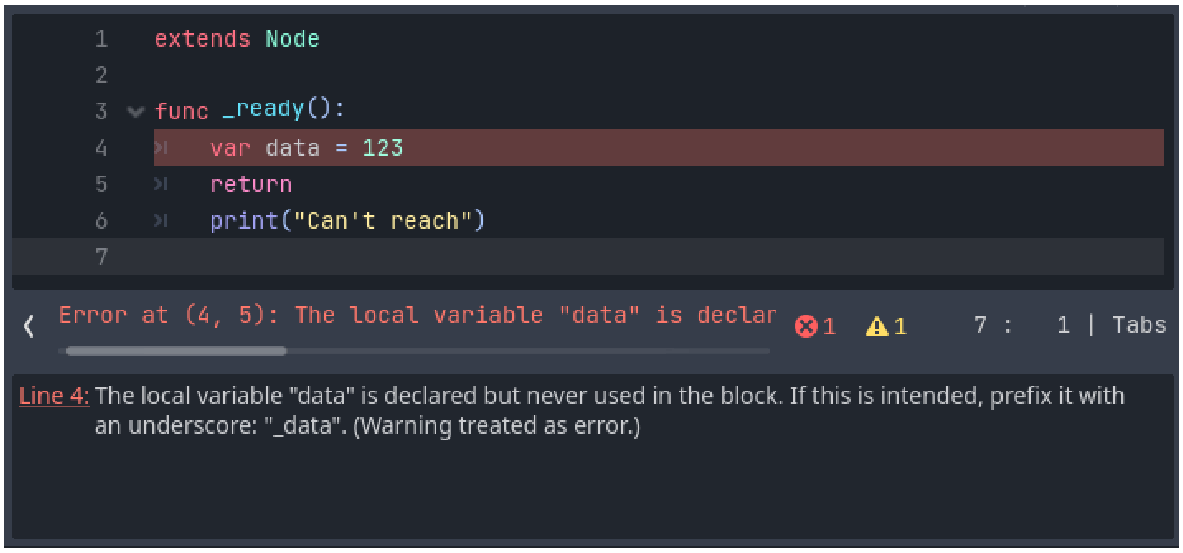


```
extends Node

func _ready():
	@warning_ignore("unused_variable")
	var data = 123
	return
	print("Can't reach")

```

⚠1   8 :   1 | Tabs

[Ignore] Line 7 (UNREACHABLE_CODE): Unreachable code (statement after return) in function "_read()".

Warnings won't prevent the game from running, but you can turn them into errors if you'd like. This way your game won't compile unless you fix all warnings. Head to the `GDScript` section of the Project Settings to turn on this option to the warning that you want. Here's the same file as the previous example with the warning `unused_variable` as an error turned on:



```
extends Node

func _ready():
	var data = 123
	return
	print("Can't reach")

```

```
Error at (4, 5): The local variable "data" is declar   1   1    7 :   1 | Tabs
```

Line 4: The local variable "data" is declared but never used in the block. If this is intended, prefix it with an underscore: "_data". (Warning treated as error.)

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.
`v`	A **vector**. Expects any float or int-based vector object (`Vector2`, `Vector3`, `Vector4`, `Vector2i`, `Vector3i` or `Vector4i`). Will display

the vector coordinates in parentheses, formatting each coordinate as if it was an `%f`, and using the same modifiers.

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 or real number placeholder. The leading `0` is ignored if `-` is present. When used after `.`, see `.`.
`.`	Before `f` or `v`, 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