基本テキスト

3章

GDScript

ver.1.0.0

この章ではGDScriptの基本的な要素と制御について解説します。GDScriptはPythonに非常によく似たコードの記述をするため、あらかじめPythonを少しでも学習しておくとスムーズに進めることができるでしょう。

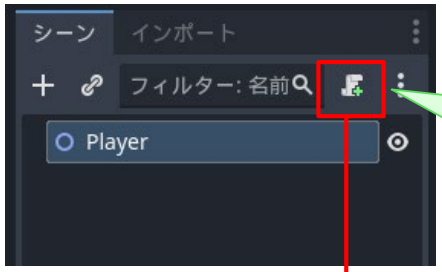
バージョン4.2.1をベースに作成し、4.3.0にも対応しています。

3-1 Scriptの基本要素

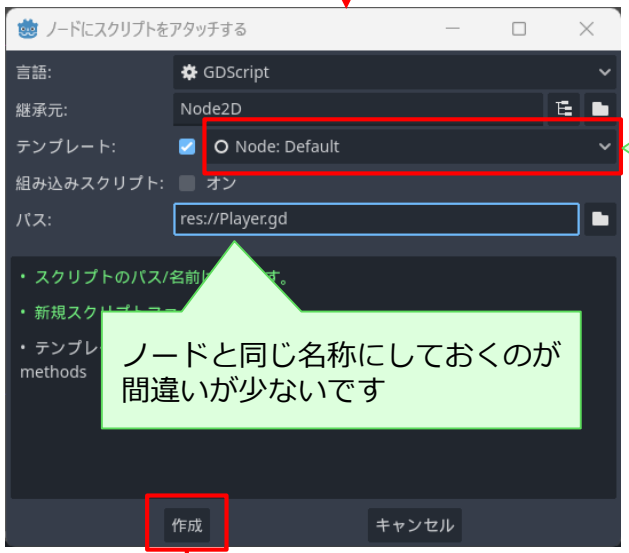
- Scriptのアタッチ
- GDScriptの基本
- コード検索
- 変数（１）
- 変数（２）
- Vector2
- String（文字列型）
- 変数の補足
- @export
- @onready
- 配列（１）
- 配列（２）
- 配列（３）
- 辞書（１）
- 辞書（２）
- enum

Scriptのアタッチ

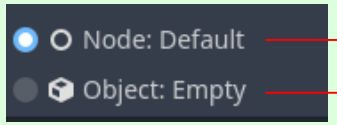
Scriptはノードに対して作成（アタッチ）します。



ノードを選択して、Scriptアタッチボタンをクリックします。



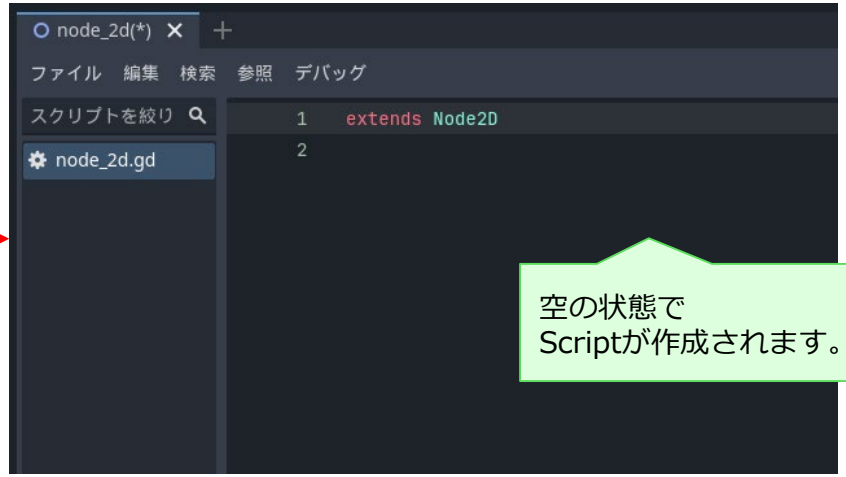
どちらかを選びます



ノードと同じ名称にしておくのが間違いが少ないです

Scriptファイルができあがります。
ファイルの拡張子は.gdとなります。

標準の命令が入った状態でScriptが作成されます。



空の状態でScriptが作成されます。

GDScript（GDスクリプト）はPython（パイソン）によく似た言語のため、Pythonの経験があれば特に違和感なく書くことができます。
逆にPython以外の言語しか経験していないと、はじめのうちは違和感を感じることもあるでしょう。

■インデントベース

多くのプログラミング言語では、条件分岐や関数のブロックを区別するために{}中括弧を使います。
例えばこのように書きます。

```
if（条件式）{
  処理１
} else {
  処理２
}
```

GDScriptではPythonと同じく、{}を用いずにインデントでブロックの区切りを表現します。インデントとは字下げのことです。

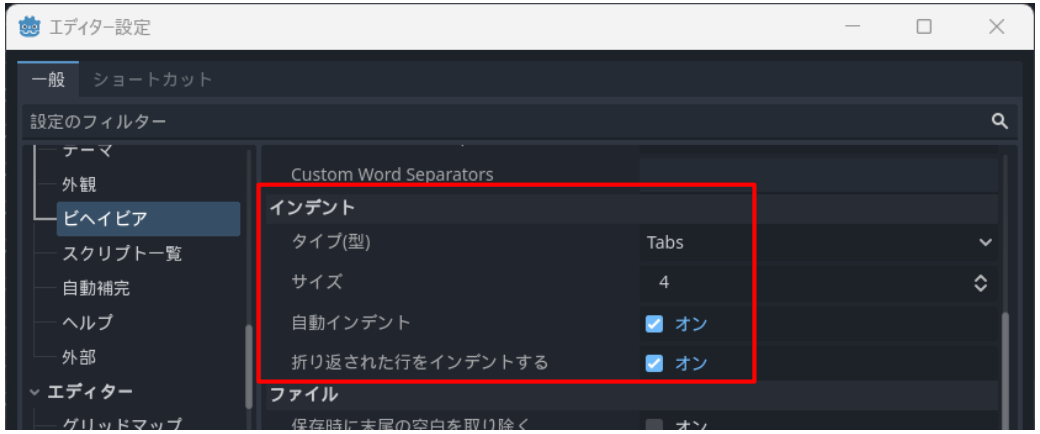
```
if 条件式：
    処理１
else:
    処理２
```

インデント → 処理１
インデント → 処理２

{}がない代わりに、:コロンが必要です。

インデントはスペースキー4つ分か[Tab]キーで入力します。
デフォルトではTabキーでの入力になっています。
※エディター設定で変更できます。

[エディター] > [エディター設定...]



他には文末の;セミコロンも不要です。ただし書いていてもエラーにはなりません。

```
var speed = 5
var hp = 20
```

```
var speed = 5;
var hp = 20;
```

■コメント

Pythonと同じく#から始まる行はコメントアウトされ、プログラムは実行されません。

```
#var speed = 5
#var hp = 20
```

複数行をまとめてコメントアウトする場合は、'''もしくは"""で挟みます。

```
'''
var speed = 5
var hp = 20
'''
```

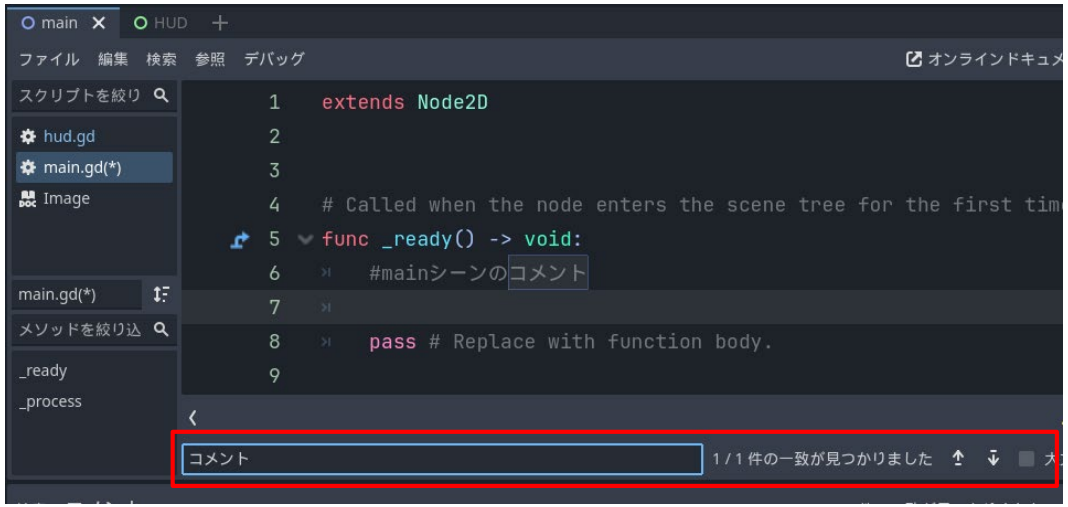
```
"""
var speed = 5
var hp = 20
"""
```

コード検索

ソースコード内の語句を検索・置換することができます。コードの量が多くなると、目的のコードを見つけにくくなっていきます。そのためコードの検索は頻繁に使うことになります。

■現在開いているコード

コードウインドウににフォーカスしている状態で、
ショートカット（Ctrl + F）で検索欄が開くので、検索したい語句を入力します。



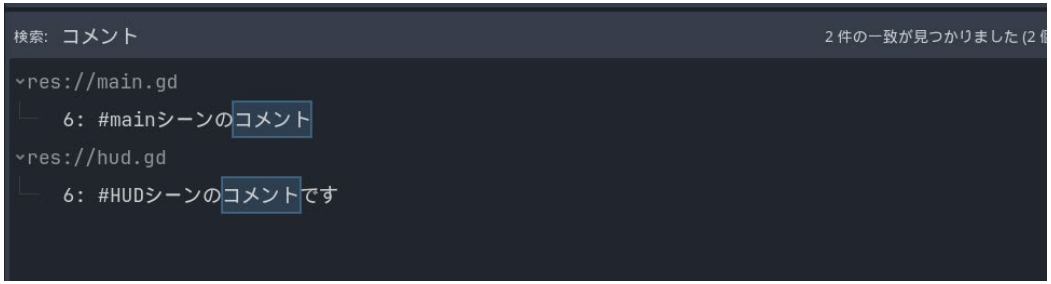
検索にヒットした語句はコードウインドウ内でハイライト表示されます。

コメント文も検索されますので、頻繁にチェックする箇所は分かりやすいコメントを入れておくようにすると良いでしょう。

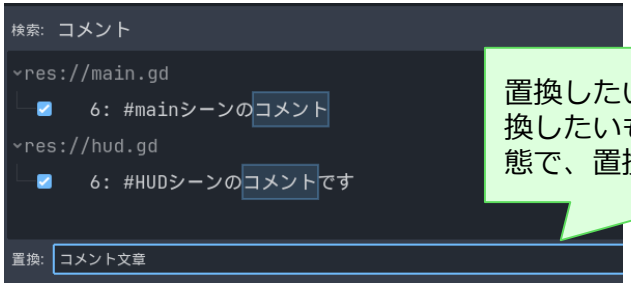
■全体検索

プロジェクト内の全てのコードを検索します。ただしスクリプトファイルが保存されている必要があります。（保存前の語句は検索されません）

ショートカット（Ctrl + Shift + F）で検索ウインドウが開くので、検索したい語句を入力して、検索ボタンをクリックします。



検索結果は下部に表示され、ソースコード内もハイライトされます。



置換したい場合は、検索結果から置換したいものにチェックを入れた状態で、置換後の語句を入力します。

変数（1）

変数はあらゆるプログラミング言語で重要な要素で、ゲーム制作においても、あらゆるところで変数を活用します。

変数は箱など入れ物をのように一つずつ入る場所を決めて使います。

■変数の型（かた）

扱うデータの種類によって型が決まります。代表的なものを記載します。

null （ヌル）	「値がない」または「無効な参照」を表す。
bool （ブール）	真偽（しんぎ）値（論理）型で、**true**（真・トゥルー）または**false**（偽・フォールス）のいずれかの値を取る。boolean（ブーリアン）の略。
int （イント）	整数型で、負の数、ゼロ、正の数を含む整数値を表す。英語のinteger（整数）の略。
float （フロート）	浮動小数点数型で、実数値を表す。小数点を含む数値や大きな数値に使用される。浮動小数点は英語で「floating point number」から。
String （ストリング）	文字列型で、テキストデータを格納する。ダブルクォーテーション(")またはシングルクォーテーション(')で囲む。Sは大文字なのに注意。 文字列（character string）から。

■変数の宣言

GDScriptは変数を宣言する際に「型」を指定してもしなくても使えますが、指定しない場合は型を気にせず使えます。

```
var A: int = 100
A = "500"

var B = 100
B = "500"
B = [1,2,3]
```

int（整数）型を指定して変数を宣言しているため、型に合わないデータを入れようとするとエラーになる。

型宣言をしていない変数は、後から異なる型のデータを入れてもエラーになりません。

あらかじめ型を宣言しておいた方が思わぬミスやバグの発生を防ぐことができます。
変数の宣言には **var** （ヴァー、バル）キーワードを使います。

var power	型、値を指定しない宣言
var power = 100	値のみを指定する宣言
var power:int	型のみ宣言（この場合はint型）
var power:int = 100	型と値を指定して宣言

var
varは、変数を意味するvariable（ヴァリアブル）からきています。
変数の名前に使用できる文字は、大文字・小文字の英数字、アンダー スコア（_）で、先頭の文字は英字かアンダースコアのみ。

特に指定がなければ変数名は、全て小文字で単語を_でつなぐ、snake_case（スネークケース）で命名するようにします。

変数名には使用できない語句もあるため注意が必要です。

変数名に使用できない予約語

and	as	assert	await
break	breakpoint	class	class_name
const	continue	elif	else
enum	extends	false	for
func	if	in	is
match	not	or	pass
preload	return	self	signal
static	true	var	void
while	yield		
INF	NAN	PI	TAU

■定数

変数は値を変更できますが、プログラム中で変更されたくない値を扱う場合は定数を使います。
定数は、**const**（コンスト）キーワードを使って宣言し、慣例的に全て大文字で記述します。

```
3   const SPEED = 100
4
5   func _ready():
6       SPEED = 50
```

変更しようとしても、エラーになり変更できません。

■変数のスコープ

変数を宣言する場所によって、その変数にアクセスできる範囲が変わります。この変数や関数がアクセス（参照）できる範囲のことをスコープとよびます。

```
1   extends Node
2
3   var test_1 = 100
4
5   func _ready():
6       var test_2 = 200
7       test()
8
9   func test():
10      print(test_1)
11      print(test_2)
```

関数（func）ブロックの外側で宣言された変数は、メンバ変数となります。

関数（func）ブロックの内側で宣言された変数は、ローカル変数となります。

ローカル変数は、その関数内でしかアクセスできないため、このように異なる関数からアクセスしようとするとエラーとなります。

メンバ変数は他のノードからもアクセスが可能です。

ポイント

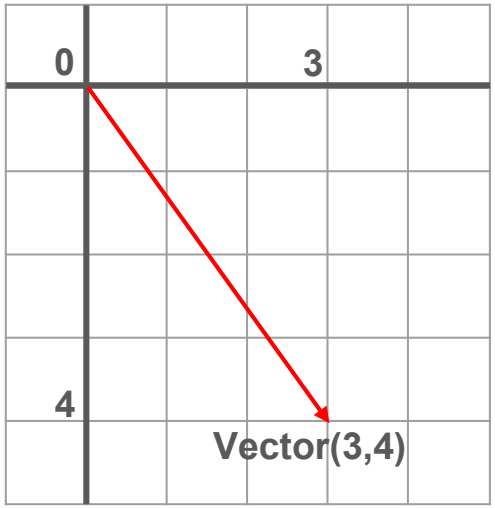
変数名やノード名を分かりやすく命名するのは「未来の自分のため」です。後から見たときには忘れていますので、何をする変数なのか、ノードなのかを分かるようにしておきます。

Vector2

Vector2（ベクター2）はGodot以外のゲームエンジンでも広く使われている概念で、2次元空間での**ベクトル**を表現できるデータ構造です。

ベクトル自体は数学、物理学、工学など様々な学問で用いられる概念で、その時々の扱われ方で違いはありますが、GodotのVector2では、xとyの２つの値を持ち、2次元空間内の点や位置、速度、方向などを表すために使用されるデータ型となります。

少し難しく感じますが、画面上のx軸とy軸の両方の位置を同時に扱える便利なデータ型と考えてください。



例えばこの図の場合、Aは座標(x=3,y=4)を表すデータとして考えることも、矢印の方向と長さをもつデータとも考えられます。

※Godotのy軸は下方向がプラスになります

vectorの読み方は使われるシーンによって「ベクトル」「ベクター」と表記は異なることがあります。このテキストでは英語の発音にも近い「ベクター」で表記しています。

Vector2型の変数名として速度の意味をもつ「velocity（ヴェロシティー）」がよく使われ、CharacterBody2DやRigidBody2Dには Vecto2型のプロパティが用意されています。

```
var test = Vector2(10,20)
position = test
```

位置（10,20) を表しています。

```
var test = Vector2(10,20)
position += test
```

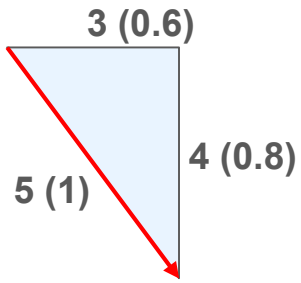
右下方向への移動量を表しています。（右に10、下に20）

```
var A = Vector2(50,50)
var B = Vector2(10,10)
print((A-B).length())
```

length()は、2点間の距離を計測します。（この例では56.568…）

```
var A = Vector2(3,4)
print(A.normalized())
```

normalized()（ノーマライズド）は、長さ1の単位ベクトルに正規化します。表示結果は(0.6, 0.8)になります。



正規化とは、(3,4)のベクトルは図の矢印で表され、これは5の長さ（大きさ）になりますが、これを長さ1にすることで、方向のみを抽出し大きさは1として、データとして扱いやすくするためのものです。

5を1として変換するため、3は0.6に、4は0.8に変換されます。

String（文字列型）

文字列データを格納するためのString（ストリング）は、単純に変数の型としてだけでなく、
メソッドを持つオブジェクトとしての一面も持っています。

```
func _ready():
    var test1:String = "My name is HANAKO"
    var test2:String = "私の名前は花子です"
    var test3:String = "100.33"

① print(test1 + "/" + test2)
② print(test1.length())
③ print(test2.length())
④ print(test1.to_upper())
⑤ print(test1.to_lower())
⑥ print(test2.replace("花子","太郎"))
⑦ print(test3.to_int() + 100)
⑧ print(test3.to_float() + 100)
```

実行結果

```
My name is HANAKO/私の名前は花子です
17
9
MY NAME IS HANAKO
my name is hanako
私の名前は太郎です
200
200.33
```

①文字列を接続するには「+」を使います。
②③length() は、文字数をカウントします。英数字やスペース、
　日本語もそのまま1文字としてカウントします。
④to_upper() は、アルファベットを全て大文字に変換します。
⑤to_lower() は、アルファベットを全て小文字に変換します。
⑥replace() は、文字を指定して置き換えます。
⑦to_int() は、文字列型から整数型に変換します。
　※小数部は無視されます
⑧to_float() は、文字列型から浮動小数点数型に変換します。

逆に、数値を文字列に変換するには、str()関数を使います。
例：str(12345)

文字列の中で特殊な記号を入力する際は、バックスラッシュを使います。環境によっては「＼」が「￥」に表示される場合があります。

\n	改行
\t	タブ
\”	ダブルクォート
\’	シングルクォート
\\	バックスラッシュ

```
print("\n")
print("\"\"")
```

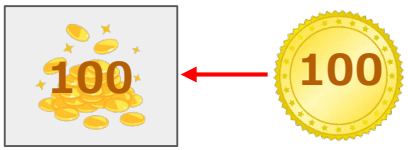
変数の補足

変数のコードは、見て目的には数学の式と同じように見えるため、はじめのうちは混乱してしまいがちです。ここで少し整理しておきます。

■代入（だいにゅう）

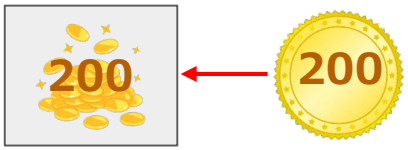
変数で使う「＝（イコール）」は右辺と左辺が同じという意味ではなく、値を入れる、更新する意味合いになるため、「代入」という表現になります。

```
var gold = 100
```



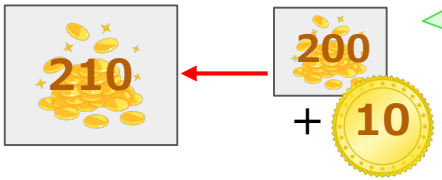
gold変数という入れ物を用意して、値として100をセットします。

```
gold = 200
```



goldの値を200にします。
どのような値であっても新しく更新するようなイメージです。

```
gold = gold + 10
```



さっきまでのgoldの値に対して計算した結果を、新しい値として更新します

図のように「＝」は「←」のイメージが近いです。

■値型と参照型

プログラムの中で扱うデータには大きく分けて「値型（あたいがた）」と「参照型（さんしょうがた）」があります。
データを変数に格納したときの挙動に違いがでます。

値型のグループ
数値、文字列、Vactor2 など

```
var A = 200
var B = A
A = 500
print(B)
```

実行結果：200

参照型のグループ
ノード、リソース、配列 など

```
var label = $Label
label.text = "XYZ"
$Label.text = "ABCD1234"
print(label.text)
```

実行結果：ABCD1234

値型の場合は、変数の中にはデータそのものが入っています。そのため、B=Aとしていても、BとAは異なるものになります。

参照型の場合は、変数の中にはデータそのものではなく、データの場所の情報が入っています。そのため元のデータに変更があれば、変数の中も変わります。

ネット上の動画に例えたイメージです。

値型：ダウンロードしたコピー
元データが変わってもコピーのため変わらない

参照型：元データをその都度見る
元データが変わると、それに置き換わる

@export

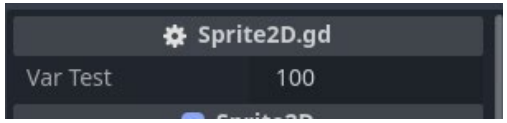
変数宣言時に **@export** をつけると、値をプログラムの変更をせずにエディタ画面上で操作できるようになります。

```
1    extends Sprite2D
2
3    @export var var_test = 100
```

エディタ画面で変数の値を変更できます

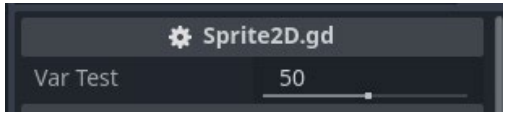
小数で設定すると、エディタでも小数入力可能になります。

```
@export var var_test = 100.0
```



@export_range は範囲を指定できます。3番目のパラメータによっては、スライダー表示にできます。

```
@export_range(0, 100, 5) var var_test = 50
```



Colorクラスを指定すればカラーピッカーにできます。

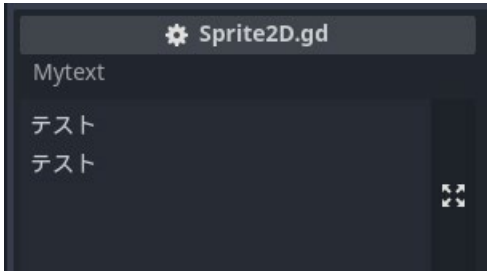
```
@export var col: Color
```

@export_file はファイル選択メニューになります。

```
@export_file var myfile
```

@export_multiline は複数行テキストになります。

```
@export_multiline var mytext = "テスト\nテスト"
```



@onready

あるノードのコード内から他のノードにアクセスする際に、毎回ノードを直接指定するコードを書くのではなく、変数を使用するとコードがシンプルになります。

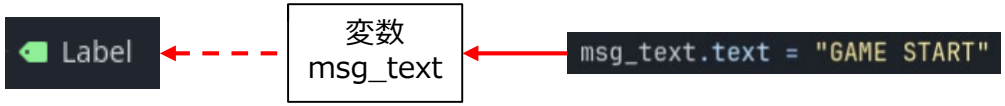
例として、Labelを表示/非表示させたり、テキストの内容を変更する場合を考えます。

このように、msg_text変数を用意してLabelノードを代入しておきます。こうすると、もしLabelノードの名前が変わっても、変数に代入している1ヶ所だけの変更で済みます。

```
3   var msg_text
4
5   func _ready():
6       msg_text = $Label
7
8   func game_start():
9       msg_text.text = "GAME START"
10
11  func game_over():
12      msg_text.text = "GAME OVER"
```

get_node("Label")の短縮表記で$Labelとしています。

変数を介して、ノードにアクセスしています。



しかし、msg_textの宣言とLabelノードの代入をコードの冒頭でまとめて行うと、エラーになってしまいます。

```
1   extends Node2D
2
3   var msg_text = $Label
```

ノードの参照を行うためには、そのノードがゲームが実行されてから実際に生成されていないと参照できません。そのために、準備が整った後に実行される_ready()関数内で初期化する必要があります。

この問題を防ぐために@onreadyを使います。

```
1   extends Node2D
2
3   @onready var msg_text = $Label
```

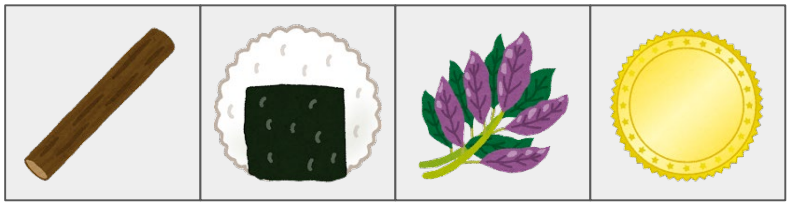
このように記述することで、_ready()関数と同じタイミングで実行されるようになります。また、変数の宣言をコードの冒頭にまとめて記述することができるようになるため、すっきりとまとまって分かりやすいコードになります。

Ctrlキーを押しながらノードをドラッグすると、自動的に@onreadyがついたコードになります。

配列（１）

配列も変数と同じくプログラミング言語で重要な要素で、複数のデータをまとめて処理できるため、ゲーム制作においても重要な要素です。

items配列



```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
```

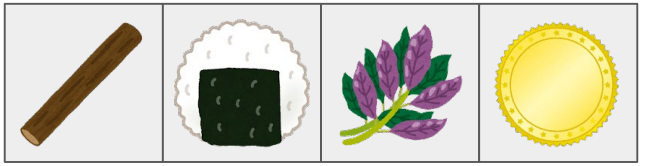
例えばゲームの中で所有しているアイテムのリスト（インベントリ）を管理する場合は、変数のように一つひとつ扱うより、配列でまとめて処理できるほうが便利です。

配列も変数と同じく宣言には **var** キーワードを使います。

var items = [] var items = Array()	型、値を指定しない
var items = [1,2,3,4,5] var items = ["one", 2, 3, "aaa"]	値を指定
var items:Array[int] var items:Array[String]	型を指定
var items:Array[int] = [1, 5, 34] var items:Array[String] = ["one", "aaa"]	型と値を指定

Array
アレイは英語でそのまま「配列」という意味です。

配列のデータにアクセスするには、インデックス番号を指定します。インデックスは先頭が[0]から始まるので注意が必要です。



インデックス　0　1　2　3

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
print(items[1])
```

実行結果

```
おにぎり
```

配列の内容を確認する際は、print()関数で出力できます。

```
print(items)
```

実行結果

```
["木の棒", "おにぎり", "薬草", "コイン"]
```

配列の要素数を取得するには、size()関数を使います。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
print(items.size())
```

実行結果

```
4
```

■要素の追加

配列の末尾に要素を追加する際は、append()を使います。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
items.append("指輪")
```

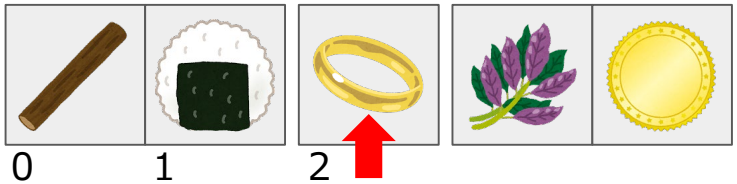
append()
アペンドは英語で「追加する」という意味

先頭に要素を追加する際は、push_front()を使います。

```
items.push_front("指輪")
```

途中に要素を追加する際は、insert()を使い、インデックスで追加したい場所を指定します。

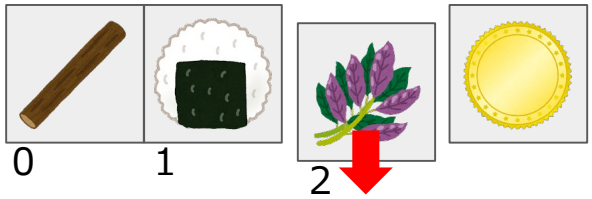
```
items.insert(2, "指輪")
```



■要素の削除

インデックスを指定して要素を削除する際は、remove_at()を使います。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
items.remove_at(2)
```



先頭の要素を削除する際は、pop_front()を使います。

```
items.pop_front()
```

最後の要素を削除する際は、pop_back()を使います。

```
items.pop_back()
```

要素の値を指定する場合は、erase()を使います。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
items.erase("薬草")
```

■要素のソート

配列に含まれる値を並べ替えることができます。
昇順にするには、sort()を使います。降順にする場合は、reverse()を使いますが、いきなり実行すると、現在の並び順が逆になるだけですので、sort()の後reverse()をします。

```
var items = [5,3,8,20,1,15]
items.sort()
print(items)
```

実行結果

```
[1, 3, 5, 8, 15, 20]
```

```
var items = [5,3,8,20,1,15]
items.sort()
items.reverse()
print(items)
```

実行結果

```
[20, 15, 8, 5, 3, 1]
```

文字列データでもソートは可能です。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
items.sort()
```

```
["おにぎり", "コイン", "木の棒", "薬草"]
```

■要素の存在チェック

in演算子を使って、配列に含まれる要素をチェックします。

```
var items = ["木の棒","おにぎり","薬草","コイン"]
if "おにぎり" in items:
    print("おにぎりを持っています")
```

実行結果

```
おにぎりを持っています
```

■反転、シャッフルする

reverse()は要素を反転し、shuffle()はランダムにシャッフルします。

```
func _ready():
    var bingo = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    bingo.reverse()
    print(bingo)
    bingo.shuffle()
    print(bingo)
```

実行結果

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[6, 8, 7, 3, 1, 5, 9, 2, 10, 4]
```

また、pick_random()を使うと、ランダムに要素を1つ取り出します。

```
var random_get = bingo.pick_random()
```

辞書（1）

配列の要素が値のみに対して、辞書は「キー」と「値」がペアになった要素を格納できます。

```
var player = {"name":"たろう", "LV": 1, "HP": 30}
```

キー	値
name（名前）	たろう
LV（レベル）	1
HP（ヒットポイント）	30

キーと値がペアになっていることで、キーを指定して値にアクセスが可能になり、配列のようにインデックスのみでアクセスするよりデータの扱いが分かりやすくなります。

```
print(player["name"])
print(player.name)
```

実際に値にアクセスするには、例のように **["キー"]** もしくは、**.キー** の2種類の書式があります。

キーには日本語も使用可能です。

変数、配列と同様に、**var**キーワードを用いて、辞書の作成を行います。

var items = {} var items = Dictionary()	要素を指定しない
var items = {“name”:”薬草”, “HP”:10}	要素を指定

■要素の追加・編集

辞書に要素を追加するには、直接キーと値を指定します。

```
var player = {"name":"たろう", "LV": 1, "HP": 30}
player["EXP"] = 50
print("経験値 " , player["EXP"])
```

実行結果

```
経験値 50
```

値を変更する場合も、直接キーと値を指定します。

```
var player = {"name":"たろう", "LV": 1, "HP": 30}
player["name"] = "はなこ"
```

■要素の削除

辞書の要素を削除するには、erase()メソッドを使います。

```
var player = {"name":"たろう", "LV": 1, "HP": 30}
player.erase("name")
print(player)
```

実行結果

```
{ "LV": 1, "HP": 30 }
```

■要素の存在チェック

辞書の要素を編集したり、削除する際に、存在しないキーを指定するとエラーになってしまいます。あらかじめキーが存在するか確認するために、has()メソッドを使います。

```
var player = {"name":"たろう", "LV": 1, "HP": 30}
print(player.has("name"))
print(player.has("MP"))
```

実行結果

```
true
false
```

存在する場合は、true
存在しない場合は、false
が値として返ってきます。

keys()メソッドでキーのみ、values()メソッドで値のみ取り出せます。

```
print(player.keys())
print(player.values())
```

実行結果

```
["name", "LV", "HP"]
["たろう", 1, 30]
```

in演算子でも存在チェックができます。

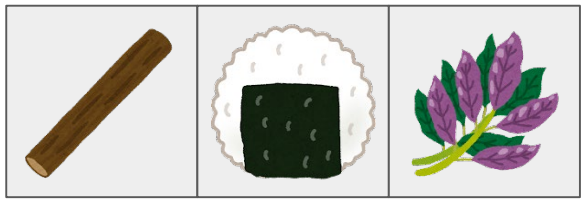
```
var player = {"name":"たろう", "LV":1, "HP":30}
print("name" in player)
```

実行結果

```
true
```

■入れ子構造

辞書の値にさらに辞書データを含めるようなデータの構成も可能です。
例えば各アイテムが持つステータスをまとめて管理するような場合には、このようなデータの作り方が考えられます。



木の棒	おにぎり	薬草
attack:15 price:25	recovery:5 price:10	recovery:20 price:30

それぞれの値自体が、さらに辞書データになっています。

```
var items = {
    "木の棒":{"attack":15, "price":25},
    "おにぎり":{"recovery":5, "price":10},
    "薬草":{"recovery":20, "price":30}
}
print(items["木の棒"]["attack"])
```

実行結果

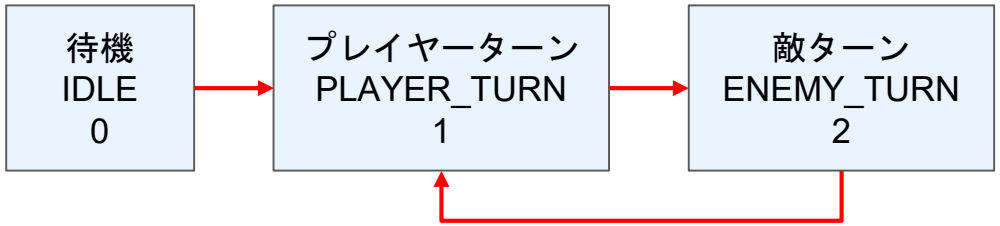
```
15
```

配列と辞書を組み合わせたデータの作り方も可能です。

```
var items = [
    {"name":"木の棒", "attack":15, "price":25},
    {"name":"おにぎり", "recovery":5, "price":10},
    {"name":"薬草", "recovery":20, "price":30}
]
print(items[1]["price"])
```

enumは列挙型と呼ばれる配列に似たものです。読み方はイーナム、イナム、イニュームなどで、由来は英語のenumeration（イニューマレイト：列挙する、数え上げる）からきています。

例えば、ターン制RPGのバトルのように、状態を管理することを考えます。



状態を配列で管理することもできます。

```
var turn_states = ["IDLE", "PLAYER_TURN", "ENEMY_TURN"]
```

配列を使った場合は、実際に状態を管理するのは配列インデックスの「0,1,2」のように数値になるため、処理的には以下のようになります。

```
#プレイヤーターンに切り替える
func start_battle():
    current_turn_state = turn_states[1]
```

状態の種類が少なければこれでも良いですが、ソースコードを見たときに「0,1,2」の数値が何を表しているのか分かりにくいです。

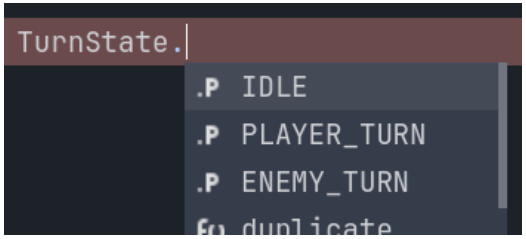
enumを使う場合はこのような書き方になります。

```
enum TurnState {
    IDLE,
    PLAYER_TURN,
    ENEMY_TURN
}
```

定数を列挙する形で書きます

```
#プレイヤーターンに切り替える
func start_battle():
    current_turn_state = TurnState.PLAYER_TURN
```

数値ではなく文字情報としてコードを見ることができるので、何が行われているのか確認しやすくなります。
また、コード補完機能も働くため、文字のタイプミスも軽減できます。



enumのそれぞれの値は内部的には、配列のインデックスのように数値データとして扱われています。

```
print("現在の状態：", current_turn_state)
```

実行結果

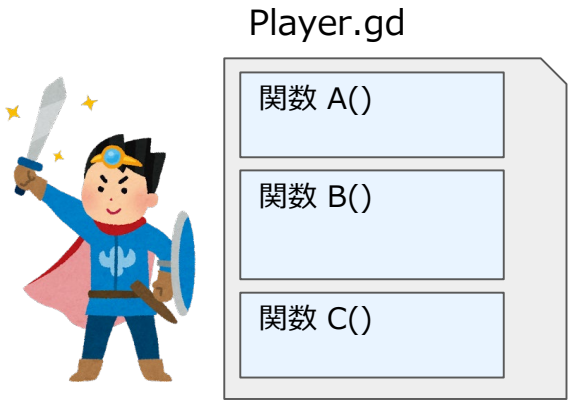
```
現在の状態：1
```

3-2 関数とメソッド

- 関数の基礎とメソッド
- 関数（１）
- 関数（２）
- 標準関数（１）
- 標準関数（２）
- 標準関数（３）
- コールバックメソッド

関数の基礎とメソッド

関数は、複数の処理をまとめたり、任意のタイミングで他のプログラミングブロックから呼び出したり、同じ処理を何度も使いまわしたりと、様々な場面で利用できるコードブロックです。
Godotではほとんどのプログラムを関数の中に書くことになります。



関数は、**func** 関数名(): の書式で定義します。

```
func say_hello():
    print("Hello, world!")
```

func 関数名(引数):
　　# 関数が実行するコード

func（ファンク）は関数を定義するためのキーワードです。関数は英語で「function（ファンクション）」のため。
Godotでは、関数名は変数と同じく全て小文字で単語を_でつなぐ、snake_case（スネークケース）で命名するようにします。

引数（ひきすう）には、関数が呼び出された際に引き渡される、関数内で利用できるデータが格納されています。

■関数とメソッド

用語として関数とメソッドは、ほぼ同じような意味で使われることが多くありますが厳密には異なっています。またプログラミング言語によっても用語の扱いが異なることもあります。

Godotではノードやシーンが「クラス」にあたるため、スクリプト内に定義する関数は「メソッド」と呼ぶほうが適切です。

基本テキスト 1-5 オブジェクト指向プログラミング

例として左のコード例ですと、say_hello()はメソッドですが、print()はGodotに組み込まれている標準の命令で、どこからでも呼び出せますので、関数となります。

ただ、あまり厳密に使い分けても説明が複雑になりがちですので、プログラミングの説明のなかでは単純に「関数」と表記していることも多くあります。

こういった用語については、特に一人でゲーム制作をする上で問題となることはないため、必ずしもこう呼ばなければならない訳ではありません。
ただ、将来チームで共同制作する場合などは、用語の統一をしておかないと誤解が生じることもあります。

■引数

引数（ひきすう）は、関数を呼び出すときに値を受け渡すことで、同じ関数でも異なる結果で実行することができます。

```
func _ready():
    say_hello("花子")

func say_hello(name):
    print("こんにちは", name,"さん")
```

実行結果

```
こんにちは花子さん
```

引数には変数と同じく「型指定」をしておくこともでき、間違った型の値が引数として送られて、関数内でエラーが起こるのを防ぐことができます。

```
func _ready():
    say_hello(123)

func say_hello(name:String):
    print("こんにちは",name,"さん")
```

String（文字列）型で宣言された引数に、数値を指定しているためエラーが出ています。

引数はカンマ（,）で区切って複数指定することもできます。

```
func _ready():
    say_hello("花子",16)

func say_hello(name:String, age:int):
    print(name,"さん は",age,"才です")
```

実行結果

```
花子さん は16才です
```

■デフォルト引数

引数にデフォルト値を設定しておけば、呼び出し時に指定がない場合は、デフォルト値がセットされます。

```
func _ready():
    say_hello()

func say_hello(name = "名無し"):
    print("こんにちは", name,"さん")
```

実行結果

```
こんにちは名無しさん
```

引数の指定がない場合は、”名無し”がnameの値として適用されます。

型指定する場合は、このような書式になります。

```
func say_hello(name:String = "名無し"):
    print("こんにちは",name,"さん")
```

引数が設定された関数を、引数なしで呼び出すとエラーになってしまいます。プロジェクトの規模が大きくなったり、複数人で開発を行うようになると、このようなエラーが起こりがちになるため、デフォルト値を設定しておくと良いでしょう。

```
func _ready():
    say_hello()

func say_hello(name:String):
    print("こんにちは",name,"さん")
```

引数なしで呼び出すとエラーに。

■戻り値

関数でなんらかの処理をした後に、処理結果（出力）を返す場合には、return文を使用して、関数の呼び出し元に値を返します。「返り値（もどりち）」と呼ぶ場合もあります。

数学で用いる「関数」には、関数への「入力」と結果の「出力」がセットになっているため、戻り値はその「出力」にあたります。

```
func _ready():
    var result = say_hello("たろう")
    print(result)

func say_hello(name):
    return "こんにちは、"+name+"さん"
```

実行結果

```
こんにちは、たろうさん
```

result（リザルト）は「結果」という意味で、戻り値を受け取る変数名としてよく使われます。

返り値についても、どんな「型」の値が返されるのかをあらかじめ指定しておくことができます。

```
func say_hello(name) -> String:
    return "こんにちは、"+name+"さん"
```

```
func say_hello(name) -> int:
    return "こんにちは、"+name+"さん"
```

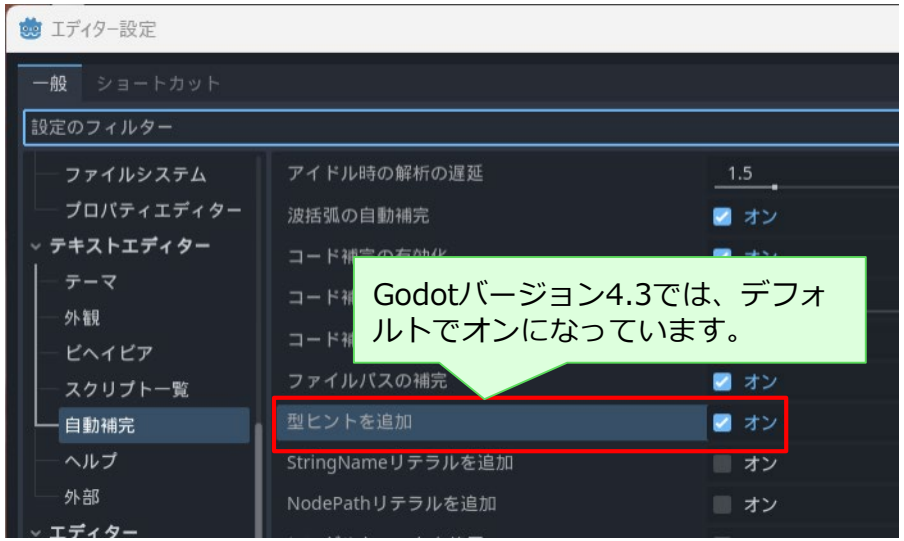
文字列を返す関数で、int型を指定しているため、実行時にエラーになります。

戻り値がない関数の場合は、特に何も書かなくてもエラーにはなりませんが、明確に戻り値があるのか無いのかを区別しておくために、返り値がない場合には「void（ボイド）」を指定しておけます。

```
func _ready() -> void:
```

スクリプトをテンプレートから作成する際に、デフォルトで型指定が入るように設定できます。

[エディター] > [エディター設定...]




```
func _ready() -> void:
    pass

func _process(delta: float) -> void:
    pass
```

標準関数（１）

関数（メソッド）は自分で作成することもできますが、Godotにはじめから用意されている標準関数もあります。

■print

print()は、コンソールにメッセージを出力するために使用します。デバッグ目的で使われ、開発中に変数の値やプログラムの状態を確認するのに役立ちます。

```
print("これからGodotでゲームを作ります")
```

実行結果

```
これからGodotでゲームを作ります
```

複数の値を出力する際は、「,（カンマ）」で区切ります。

```
print("体力：", 100, "  魔力：", 20)
```

実行結果

```
体力：100  魔力：20
```

■printerr

printerr()は、print()と似た処理をしますが、コンソールにはこの例のようにエラー表記で出力されます。

```
printerr("変数numの値が不正です")
```

実行結果

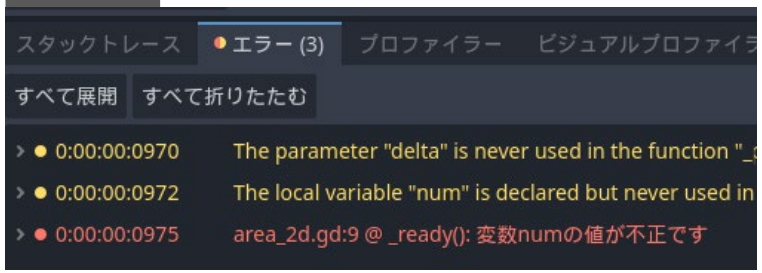
```
● 変数numの値が不正です
```

■push_error

push_error()は、printerr()と同じようにエラーを出力しますが、エラーログに記録されるため、明確にエラーを検出しやすくなります。

```
push_error("変数numの値が不正です")
```

実行結果



エラー処理についてはこちらも参照。

基本テキスト 1-2 エラー処理

■str

str()は、数値を文字列に変換します。

```
var num = 12345
var str = "文字列：" + str(num)
print(str)
```

実行結果

```
文字列：12345
```

標準関数には数学的要素を扱う関数もたくさんあります。

■floor/ceil/round

floor()	小数以下を切り捨て。（読み：フロア）英語のfloorが床、地面という意味を持つところから
ceil()	小数以下を切り上げ。（読み：シール）英語のceilingが天井という意味を持つところから
round()	小数以下を四捨五入。（読み：ラウンド）英語のroundが丸めるという意味を持つところから

```
var num1 = 123.45
var num2 = 67.89

print("切り捨て：", floor(num1))
print("切り上げ：", ceil(num1))
print("四捨五入：", round(num1))
print("四捨五入：", round(num2))
```

実行結果

```
切り捨て：123
切り上げ：124
四捨五入：123
四捨五入：68
```

■abs/pow/sqrt

abs()	絶対値を返します。絶対値は「-」マイナスの符号を取り除いた数値そのものです。（読み：エービーエス、アブス）英語で絶対値は「absolute value」のため。
pow()	べき乗計算をします。（読み：パウ）英語でべき乗は「power」のため。
sqrt()	平方根を計算します。（読み：スクエアルート、スクルト、スクアートなど）英語で平方根は「square root」のため。

```
print("12の絶対値：", abs(-12))
print("2の3乗：", pow(2, 3))
print("16の平方根：", sqrt(16))
```

実行結果

```
12の絶対値：12
2の3乗：8
16の平方根：4
```

■max/min

与えられた数値の中から、最大値、最小値を出力します。

```
print(max(3.5, 7, -2))
print(min(3.5, 7, -2))
```

実行結果

```
7
-2
```

■clamp

与えられた値を、指定した範囲内に収める。キャラクターの座標が画面の範囲外へ出ないようにする場合などに活用できます。

clamp(値, 最小値, 最大値)

```
print(clamp(25, 5, 15))
```

実行結果

```
15
```

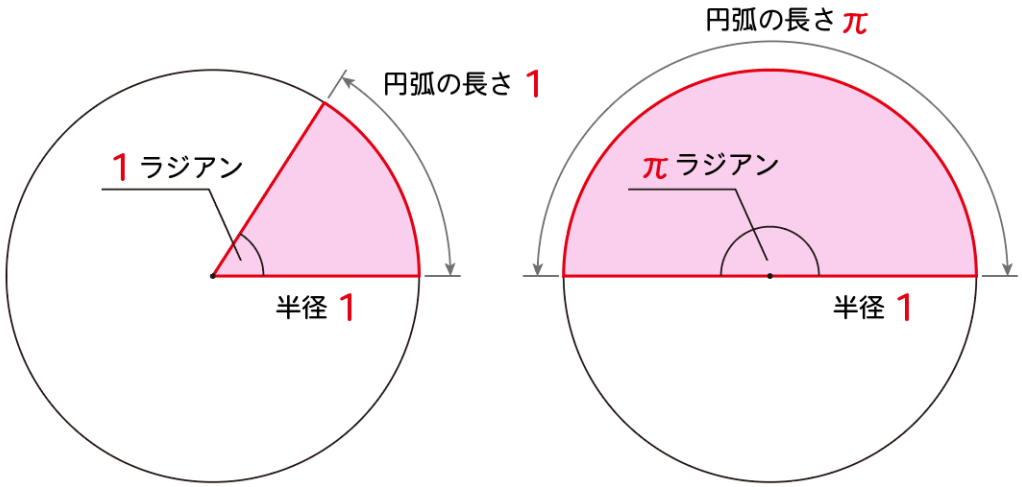
値25が、5～15の範囲に収まるようになるため、15が出力されます。

■deg2rad/rad2deg

Godotでは角度を表す単位は、ラジアン（radian）を使用しています。一般的には「度数法」を用いて角度の単位を「°」で表していますが、ラジアンとは弧度法（こどほう）と呼ばれる角度を表す方法で、単位は「rad」を用います。
弧度法は高校の数学Ⅱで習います。

deg2rad()は、度数法の角度をラジアンに、
rad2deg()は、ラジアンから度数方の角度に変換します。

少しだけラジアンについて説明しておきます。
図のように半径が１，円弧の長さが１のときの角度が１ラジアンとなります。



また円周＝直径x円周率（π）であることから、半径１のとき半円の弧の長さがπとなり、180°＝πラジアンとなります。
このように、角度を円周率で表現できるのが弧度法の特徴で、これにより画面上を移動する物体の計算がやりやすくなります。

■sin/cos/tan/asin/acos/atan

これらは三角関数を扱う関数です。ゲーム内では様々なところで使用されます。前述のラジアンとも密接な関りがあります。三角関数も高校数学で学習します。
ゲームのキャラクターの動きなどで使用する際に具体的に解説することになります。

■randf/randf_range/randi/randi_range

ランダムな数字を生成します。

```
print("0.0～1.0の乱数：", randf())
print("0.0～10.0の乱数：", randf_range(0, 10))
print("0～4294967295の乱数：", randi())
print("0～10の乱数：", randi_range(0, 10))
```

実行結果

```
0.0～1.0の乱数：0.34645766019821
0.0～10.0の乱数：6.85627385704998
0～4294967295の乱数：2345163661
0～10の乱数：1
```

■seed

seed()で引数に何らか決まった数字を設定すると、ゲーム実行時から毎回同じランダム値が生成されます。ゲームのデバッグを行う際に使うと便利です。

```
seed(1234)
print("0.0～1.0の乱数：", randf())
```

実行結果

```
0.0～1.0の乱数：0.10128547996283
```

何度実行しても同じ値が生成されます。

スクリプトをテンプレートから作成したときに、はじめから用意されている_ready()や_process()など、_（アンダースコア）で始まるものは特別なメソッドで、ゲーム実行時に任意のタイミングで自動的に呼び出されます。

■ready

_ready()は、ゲーム実行開始時に各シーンそれぞれの中で全てのノードが読み込まれた（準備できた）タイミングで呼び出されます。
一度だけ実行されるので、ノードを初期化したり、何かゲーム内での準備を行うような処理を入れます。

■process

_process(delta)は、ゲーム開始後に毎フレームごとに実行されます。例えば60FPS（frames per second）の場合は、1秒間に60回画面が書き変わりますので、1秒間に60回呼び出されることになります。

引数のdeltaは、前のフレームからの経過時間を秒単位で計測したものです。

```
func _process(delta):
    print(delta)
```

```
0.01666666666667
0.01666666666667
0.01666666666667
```

このコードを実行すると、毎フレームdeltaの値を出力します。
結果は0.0166...となっており、これは1/60の値です。
ですので、60FPSで動作していることになります。

特に設定しなければPCの性能によって、1秒間の呼び出し回数も変わってしまいます。

■physics_process

_physics_process(delta)は、_process()と似ていますが違いは、こちらは物理エンジン処理の更新後tに呼び出されることです。そのため_process()とは異なるタイミングで呼び出され、基本的には60FPS固定です。

_process()は、アニメーションやグラフィック、タイマーの処理などを行うのに対し、_physics_process()は、キャラクターや物体の移動など物理処理を行う際に使用します。

■input

_input(event)は、キーボードやマウス、ゲームパッドなどユーザーからの入力があった際に呼び出されます。引数のeventには、どのような種類の入力があったのかが設定されます。

pass
何もしない場合に使用します。とりあえず関数を定義だけしておいたり、条件分岐で何もしない場合に使います。

```
5  func _ready():
6      pass # Replace with functi
```

3-3 制御構文

- 演算子
- ループ（1）
- ループ（2）
- 条件分岐
- タイマー（1）
- タイマー（2）
- await

■算術演算子

+	足し算、プラスの符号
-	引き算、マイナスの符号
*	掛け算
/	割り算
%	割り算のあまり（10%3の場合1となる）
**	べき乗（5**2の場合25となる）

算術演算子は短縮した記述も可能です。例えば
a = a+1 は a+=1 のように書くこともできます。

■比較演算子

左辺と右辺の数値の大小関係を比較します。

==	A == B（AとBは等しい）
!=	A != B（AとBは等しくない）
>	A > B（AはBより大きい）
>=	A >= B（AはBと同じか大きい）
<	A < B（AはBより小さい）
<=	A <= B（AはBと同じか小さい）

■論理演算子

if文、while文などの条件式の中で使用します。

and , &&	A and B , A && B　※andを推奨
or , ||	A or B , A || B　※orを推奨
not , !	not A , !A　※notを推奨

※orの||は、キーボード右上の方にある  で入力できます。

■in、is演算子

in演算子は、配列や辞書、文字列に特定の値が含まれているかどうかを調べる際に使います。
→配列、辞書のページを参照

```
if "ABC" in "ABCDEFG":
    print("ABCが含まれます")
```

is演算子はオブジェクトの型チェックに使用します。

```
if get_node("Player") is Node2D:
    print("PlayerはNode2D型です")
```

```
func _on_input_event(viewport, event, shape_idx):
    if event is InputEventMouseButton:
```

■forループ

forループは、もっとも基本的な繰り返し処理です。

```
for i in range(5):
    print(i)
```

実行結果

```
0
1
2
3
4
```

```
for i in range(5,10):
    print(i)
```

実行結果

```
5
6
7
8
9
```

for 変数 in 繰り返し回数：

変数は例では i を使うことが多くありますが、それに限りません。
※indexの頭文字として i がよく使われています。
繰り返し回数の指定は、range()関数を使います。

range()関数は、指定された範囲の数値を生成します。

range(5)	0～4を出力
range(3, 8)	3～7を出力
range(0, 10, 2)	0,2,4,6,8を出力 3つ目の引数はステップを指定します

forループで配列を指定すれば、配列の要素を順番に取り出すことができます。

```
var items = ["木の棒", "おにぎり", "薬草", "コイン"]
for name in items:
    print(name)
```

実行結果

```
木の棒
おにぎり
薬草
コイン
```

■breakとcontinue

breakはループの途中から強制的に抜けます。
continueはそれ以降の処理を中断して、次のループに進みます。

```
for i in range(10):
    if i == 6:
        break
    elif i < 3:
        continue
    print(i)
```

実行結果

```
3
4
5
```

3より小さい場合はprint()をせず、
6になった際にループを終了します。

■ネストしたループ

ネストとは入れ子のことです、ループの中にループがあるような構造です。

```
for x in range(3):
    for y in range(5):
        print("(", x, ",", y, ")")
```

実行結果

```
(0,0)
(0,1)
(0,2)
(0,3)
(0,4)
(1,0)
(1,1)
(1,2)
(1,3)
(1,4)
(2,0)
(2,1)
(2,2)
(2,3)
(2,4)
```

この例だとまず外側のループで
　x = 0
の状態になり、次に内側のループで
　y = 0 ～ 4
までくりかえされます。

その後に外側のループに戻り
　x = 1
の状態で再び内側のループがくりかえされます。

結果的に3x5 = 15回ループがくりかえされます。

■whileループ

whileループは特定の条件が、true（トゥルー）の間は繰り返しを続け、false（フォールス）になるとループを抜けます。

```
var count = 0
while count < 5:
    print(count)
    count += 1
```

実行結果

```
0
1
2
3
4
```

```
while true:
    print(0)
```

実行結果

```
0
0
0
0
0
```

条件がずっとtrueなため、無限ループします

while 条件：

条件がtrueの間はループを続けます。上記の例だと、countが1ずつ足されて、5になった場合に条件 count<5 を満たさなくなる=falseになるため、ループから抜けます。

■if elif else

GDScriptの条件分岐処理も他の言語と特別異なるものはありません。

```
var test = 10
if test == 5:
    print("5です")
elif test < 5:
    print("5より小さい")
else:
    print("5以上です")
```

論理式と組み合わせることで複雑な条件で分岐処理を行うことができます。

```
var test = 7
if test < 3 or test > 8:
    print("3より小さい、もしくは、8より大きい")
elif test >= 3 and test <=5:
    print("3以上、かつ、5以下")
elif test != 6:
    print("6ではない")
```

■match

match文は他の言語の、switchやcaseと同じような使い方をします。

```
var fruit = "APPLE"
match fruit:
    "APPLE":
        print("りんご")
    "BANANA":
        print("バナナ")
    _:
        print("他のくだもの")
```

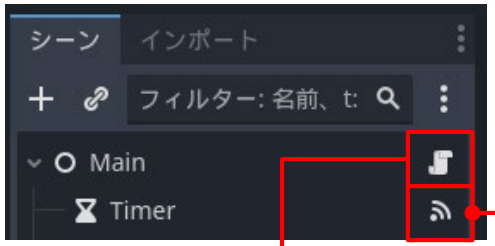
条件のどれにも当てはまらない場合には、_で示したものが実行されます。

特定の値ごとに実行するアクションが決まっている場合は、if文よりもmatch文を使うほうが、コードの見た目も読みやすさにおいても優れています。

タイマーはゲーム制作においても、様々な場面で活用します。
一定時間ごとに動作させたり、ゲーム内で「間」をとることでゲーム体験をより良いものにすることができます。

■Timerノード

エディタであらかじめTimerノードをつくっておく方法です。



シグナル設定をしておきます

```gdscript
extends Node

func _ready():
    get_node("Timer").start()

func _on_timer_timeout():
    print("タイマー経過")
```

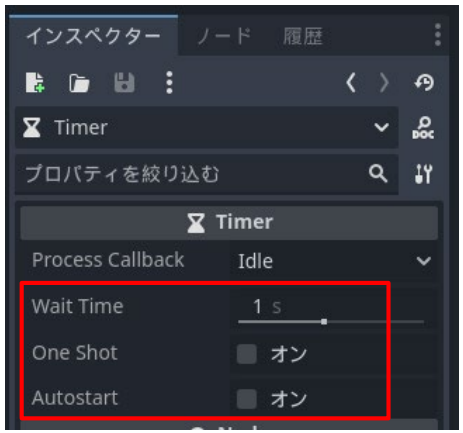
ゲームが実行されたらタイマーを開始します。

タイマーがカウントされるたびにシグナルにより呼び出されます。

実行結果

```
タイマー経過
タイマー経過
タイマー経過
タイマー経過
```

設定された時間が来るたびにシグナルが発信され、メソッドを呼び出します。
これにより一定間隔で決まった動作をさせることができます。



インスペクターで設定できるプロパティです。

Wait Time	呼び出し時間（秒）
One Shot	一度だけ呼び出す
Autostart	自動的にスタートする

このサンプルは、5回カウントしたらタイマーを停止します。

```gdscript
extends Node

var count = 0

func _ready():
    get_node("Timer").start()

func _on_timer_timeout():
    count += 1
    print("タイマー：", count, "回目")
    if count > 4:
        get_node("Timer").stop()
        print("タイマー停止")
```

実行結果

```
タイマー：1回目
タイマー：2回目
タイマー：3回目
タイマー：4回目
タイマー：5回目
タイマー停止
```

タイマーはあらかじめエディタでノードを作成しておくことも、プログラムの実行中に生成することもできます。一時的に少し処理待ちをしたい場合などに使い捨てのタイマーをつくるイメージです。

簡易的なタイマーをつくるだけなら、create_timer()関数を使います。

```
print("タイマースタート")
await get_tree().create_timer(1).timeout
print("タイマー終了")
```

これだけで１秒間待ってから処理するタイマーになります。
注意が必要なのが下の例のように、_process() 関数のように毎フレーム呼び出される処理の中で使ってしまうと、毎フレームタイマーが生成されてしまいます。

```
func _process(delta):
    print("タイマースタート")
    await get_tree().create_timer(1).timeout
    print("タイマー終了")
```

実行結果

```
タイマースタート
タイマー終了
タイマースタート
タイマー終了
タイマースタート
```

タイマーノードを直接コード内で生成する方法もあります。

```
var timer = Timer.new()
add_child(timer)
timer.wait_time = 2.0
timer.one_shot = true
timer.timeout.connect(_on_timer_timeout)
timer.start()
print("タイマースタート")
```

コードが長くなってしまいますが、エディタで管理するよりコード内で全て完結させた方がよい場合などでは、こちらの方法を使うと良いでしょう。

awaitは、非同期処理を扱うためのキーワードです。非同期処理（例えばシグナルの待機、アニメーションの終了など）が完了するまで、スクリプトの実行を一時停止することができます。

awaitで良く使う例は、この一時的に時間を待つ処理です。
この例はシグナルを待つ利用例となっています。

```
func _ready():
    print("スタート")
    await get_tree().create_timer(1).timeout
    print("1秒経過")
```

この部分は複数の処理が一行にまとめられています。

```
await get_tree().create_timer(1).timeout
```

それぞれの処理で行っていることは・・・
・get_tree()：シーンツリーを参照します
・create_timer(1)：1秒間のタイマーを生成します
・timeout：timeoutシグナルを参照します
結果的に、この一行では、awaitによって、1秒間のタイマーを待つ処理になります。

実行結果

```
スタート
1秒経過
```

スタートが表示されてから1秒後に表示されます

他にも「コルーチン」を待機する処理が可能です。コルーチンは簡単に言うとawaitで待機できる関数のことです。

```
func _ready():
    print("スタート")
    await await_test()
    print("await関数終了")

func await_test():
    print("await関数スタート")
    await get_tree().create_timer(3).timeout
    print("3秒経過")
```

実行結果

```
スタート
await関数スタート
3秒経過
await関数終了
```

awaitを付けずに呼び出すと、関数の処理を待たずに処理が進んでいることが実行結果から分かります。

```
func _ready():
    print("スタート")
    await_test()
    print("await関数終了")
```

実行結果

```
スタート
await関数スタート
await関数終了
3秒経過
```