📝 Guia GDScript no Godot na Pratica 2025

Este guia cobre a utilização de métodos de entrada, variáveis, estruturas de fluxo, listas, dicionários, configurações recomendadas para Pixel Art e mapeamento de Inputs no Godot Engine. Além disso, apresenta a sintaxe da linguagem GDScript e boas práticas para o seu uso.

🚀 Introdução ao GDScript

O GDScript é a linguagem de script do Godot, projetada para ser simples e intuitiva. Ele utiliza uma sintaxe similar ao Python, com indentação obrigatória para definir blocos de código.

🖌️ Convenções e Estilo

Manter um estilo consistente no código ajuda na legibilidade e manutenção. Aqui estão as principais convenções e boas práticas para GDScript:

🔤 Convenções de Nomeação

- **snake_case**: Use letras minúsculas com underscore (_) para separar palavras em variáveis e funções.
  - **Exemplo:**

```
var minha_variavel = 10
func calcular_dano(dano: int):
    return dano * 2
```

- **UPPERCASE**: Constantes são declaradas em maiúsculas para diferenciá-las de variáveis comuns.
  - **Exemplo:**

```
const VELOCIDADE_MAXIMA: int = 200
```

- **PascalCase**: Use para nomear classes ou scripts.
  - **Exemplo:**

```
class_name Personagem
```

📐 Formatação de Código

- **Indentação**: Utilize 4 espaços (nunca misture com tabs) para identar blocos de código.
  - **Errado:**

```
func exemplo():
  print("Indentação incorreta")
```

  - **Correto:**

```
func exemplo():
    print("Indentação correta")
```

- **Espaços ao Redor de Operadores**: Use espaços para separar operadores para melhor leitura.
  - **Exemplo:**

```
var soma = 1 + 2
```

- **Linhas em Branco**: Insira uma linha em branco entre funções ou blocos lógicos para separar visualmente o código.
  - **Exemplo:**

```
func inicializar():
    print("Início do jogo")

func atualizar(delta: float):
    print("Atualizando lógica do jogo")
```

📋 Comentários

- **Comentários de Linha**: Use `#` para explicar partes específicas do código.
  - **Exemplo:**

```
# Calcula o dano do personagem
func calcular_dano(dano: int):
```

```
    return dano * 2
```

- **Comentários de Bloco**: Use várias linhas para descrever trechos complexos.
  - **Exemplo:**

```
# Este método é responsável por atualizar a posição do personagem
# baseado no vetor de movimento e no delta time fornecido.
func atualizar_posicao(delta: float):
    posicao += movimento * delta
```

🧩 Exemplo de Sintaxe

Abaixo está um exemplo básico de um script GDScript que demonstra a criação de constantes, variáveis e uma função:

```
# Este é um exemplo de código GDScript
const GRAVIDADE: float = 9.8  # Constante que representa a gravidade
var velocidade: Vector2 = Vector2.ZERO  # Variável inicializada com vetor zero

# Função para calcular movimento com base no delta time
func calcular_movimento(delta: float):
    velocidade.y += GRAVIDADE * delta  # Incrementa a velocidade no eixo Y
    move_and_slide(velocidade)  # Move o objeto com base na velocidade
```

📜 Outras Práticas Relevantes

- **Nomeação de Funções**: Escolha nomes que descrevam claramente sua finalidade.
  - **Exemplo:**

```
func calcular_pontos(dano: int):
    return dano * 10
```

- **Limite de Linhas**: Mantenha funções curtas e diretas (máximo de 20 linhas sempre que possível).
- **Evite Ambiguidade**: Escreva código claro em vez de compacto.
  - **Errado:**

```
if x: print("Verdadeiro")
```

  - **Correto:**

```
if x:
    print("Verdadeiro")
```

🌎 Métodos de Entrada

🎯 Input.get_vector()

Esse método é usado para obter direções de movimento com base nas teclas configuradas no editor.

Exemplo:

```
var direcao: Vector2 = Input.get_vector("move_esquerda", "move_direita",
"move_cima", "move_baixo")
velocidade = direcao * VELOCIDADE_PERSONAGEM
```

🕹️ Input.get_axis()

Captura valores de eixo para uma direção específica (horizontal ou vertical).

Exemplo:

```
var horizontal: float = Input.get_axis("move_esquerda", "move_direita")
var vertical: float = Input.get_axis("move_cima", "move_baixo")
var direcao: Vector2 = Vector2(horizontal, vertical)
```

🔧 Dica Prática

- Use `Input.get_vector()` para movimentos gerais e `Input.get_axis()` para controles analógicos, como joysticks de gamepads.

🔢 Variáveis e Constantes

📂 Declaração de Variáveis

- Use `var` para criar variáveis.

Exemplo:

```
var vida: int = 100
var nome: String = "Herói"
```

🔒 Declaração de Constantes

- Use `const` para valores que não podem ser alterados após a definição.

Exemplo:

```
const VELOCIDADE_MAXIMA: int = 200
const TAMANHO_TELA: Vector2 = Vector2(320, 240)
```

🌟 @export var e @onready var

- **@export var**: Permite que a variável seja ajustada diretamente no editor do Godot.

Exemplo:

```
@export var vida_inicial: int = 100
@export var nome_personagem: String = "Aventureiro"
```

- **@onready var**: Inicializa a variável apenas quando o nó ao qual ela se refere está carregado na cena.

Exemplo:

```
@onready var sprite: Sprite2D = $Sprite
```

🔵 Estruturas de Dados: Listas e Dicionários

📜 Listas

Listas são como caixas onde você guarda várias coisas em uma ordem específica. Cada item dentro da lista tem um número chamado **índice**. Esse número é usado para encontrar o item na lista. Lembre-se: as listas começam a contar do **índice 0** (zero). Isso é um pouco como uma fila de objetos, onde o primeiro item é o índice 0, o segundo item é o índice 1, e assim por diante.

Exemplo:

```
var dialogos_npc: Array = ["Olá, aventureiro!", "Posso te ajudar?", "Boa sorte na sua jornada!"]
```

Aqui, criamos uma lista chamada `dialogos_npc`. Ela tem 3 itens:

1. "Olá, aventureiro!" (índice 0)
2. "Posso te ajudar?" (índice 1)
3. "Boa sorte na sua jornada!" (índice 2)

Se quisermos mostrar o primeiro diálogo, usamos o índice 0:

```
print(dialogos_npc[0])  # Saída: Olá, aventureiro!
```

Dica: Se você tentar acessar um índice que não existe, o jogo vai dar um erro. Então, sempre verifique se o índice está dentro do intervalo de itens da lista!

🗂️ Dicionários

Agora, dicionários são um pouco diferentes. Em vez de usar índices numéricos, você cria um **par chave-valor**. A **chave** é como o nome do item, e o **valor** é o que ele representa ou contém.

Exemplo:

```
var inventario: Dictionary = {
    "poções": 5,
    "ouro": 100
}
```

Aqui, criamos um dicionário chamado `inventario`, que tem duas chaves:

- A chave **"poções"** tem o valor **5**, ou seja, você tem 5 poções.
- A chave **"ouro"** tem o valor **100**, ou seja, você tem 100 moedas de ouro.

Para acessar o valor de um item, usamos a chave:

```
print(inventario["poções"])  # Saída: 5
```

Dica: Sempre que quiser saber o valor de algo específico no seu dicionário, use a chave correspondente! Se a chave não existir, o jogo vai dar um erro, então fique atento.

📊 Estruturas de Controle de Fluxo

As **estruturas de controle** são como placas de sinalização no código, dizendo para o jogo o que ele deve fazer em diferentes situações. Vamos ver algumas das mais importantes!

🔀 if e else

O `if` é uma estrutura de controle que verifica se algo é **verdadeiro**. Se for, ele executa um bloco de código. Se não for, ele pode fazer outra coisa (com o `else`).

Exemplo:

```
var vida: int = 0

if vida <= 0:
    print("Game Over")  # Se a vida for menor ou igual a 0, o jogo acaba.
else:
    print("Continue jogando")  # Se a vida for maior que 0, você continua jogando.
```

Aqui, o código vai verificar se a vida é **menor ou igual a 0**. Se for, ele imprime **"Game Over"**, senão, ele imprime **"Continue jogando"**.

Dica: Você pode usar o `if` para testar qualquer condição, como se o jogador tem uma arma, se o tempo acabou, ou qualquer outra coisa que precise de verificação.

🔄 for

O `for` é usado para repetir um bloco de código várias vezes. Ele percorre uma lista, dicionário ou intervalo (um conjunto de números).

Exemplo 1: Iterando sobre uma lista

```
var nomes: Array = ["Ana", "Carlos", "Luana"]
```

```
for nome in nomes:
    print(nome)  # Vai imprimir cada nome na lista
```

Aqui, o `for` vai passar por cada nome da lista `nomes` e imprimir. O primeiro nome será "Ana", depois "Carlos", e por último "Luana".

Exemplo 2: Iterando sobre um intervalo

```
for i in range(3):
    print("Número: " + str(i))  # Vai imprimir os números 0, 1 e 2
```

Aqui, o `range(3)` cria um intervalo de 0 a 2 (o número 3 não é incluído). O `for` vai passar por esses números e imprimir.

Dica: O `for` é super útil quando você tem uma lista de coisas e quer fazer algo com cada uma delas, como exibir itens de um inventário ou somar pontos.

🔁 while

O `while` é uma estrutura de controle que repete um bloco de código enquanto uma condição for verdadeira. Isso é legal para situações em que você não sabe exatamente quantas repetições vai precisar, mas sabe até que ponto deve continuar.

Exemplo:

```
var contador: int = 0

while contador < 5:
    print(contador)  # Vai imprimir os números de 0 a 4
    contador += 1  # A cada repetição, incrementa 1 ao contador
```

Aqui, o `while` vai continuar rodando enquanto o `contador` for menor que 5. Cada vez que o loop passa, o contador aumenta de 1.

Dica: Cuidado para não criar loops infinitos! Certifique-se de que a condição vai mudar algum dia, caso contrário o código vai ficar preso no loop para sempre.

🎯 match

O `match` funciona como um "se... caso" em outras linguagens de programação. Ele é útil quando você tem várias opções para verificar e agir conforme o caso.

Exemplo:

```
var direcao: Vector2 = Vector2.UP

match direcao:
    Vector2.UP:
        print("Movendo para cima")  # Se a direção for para cima, faz isso
    Vector2.DOWN:
        print("Movendo para baixo")  # Se for para baixo, faz aquilo
    _:
        print("Parado")  # Se não for nem para cima nem para baixo, faz outra coisa
```

Aqui, o `match` está comparando o valor de `direcao`. Se for **Vector2.UP** (cima), ele imprime "Movendo para cima". Se for **Vector2.DOWN** (baixo), ele imprime "Movendo para baixo". Se for qualquer outro valor (o `_` é um "corpo vazio", um padrão), ele imprime "Parado".

Dica: O `match` é ótimo quando você tem várias condições possíveis, e quer tratá-las de forma organizada e legível, como decidir a direção do personagem.

📂 Funções

Funções em GDScript: O Que São e Como Funcionam? 🚀

Imagine que uma **função** é como uma **máquina** que você monta para fazer uma tarefa específica. Você dá alguns **ingredientes** (valores) para a máquina, ela faz algo com eles e às vezes **devolve** algo como resultado. Vamos ver como isso tudo funciona com explicações simples!

---

1. Como Declarar uma Função: Montando a Máquina 🛠️

Quando você cria uma função, você está **construindo uma máquina**. Aqui está a estrutura básica de como declarar uma função em GDScript:

```
func nome_da_funcao():  # Aqui você está nomeando a função (máquina)
    # O que a função faz vai aqui dentro
    print("Isso é o que a função faz!")
```

- `func`: Essa palavra começa a definição da sua **máquina** (função).
- `nome_da_funcao`: É o **nome da sua máquina**. Você pode chamar sua função de qualquer nome que quiser, por exemplo, `somar`, `imprimir_mensagem` ou `fazer_pizza`.
- `()` (os parênteses): Dentro desses parênteses, você pode **colocar ingredientes** (valores) que a função vai usar. Se não precisar de ingredientes, deixa vazio.
- `:` (dois pontos): Diz que a parte importante da função começa ali, o que significa que tudo dentro dela é o **trabalho** que a função vai fazer.
- **Dentro da função**, colocamos o que ela vai **fazer**. No exemplo acima, a função simplesmente mostra uma mensagem.

---

2. Funções Que Não Retornam Nada: Só Fazem e Pronto! 🚫

Algumas funções não precisam devolver nada, elas só fazem alguma coisa para você. Por exemplo, uma função pode **mostrar uma mensagem**, mas não precisa devolver nada para o programa.

```
func mostrar_mensagem() -> void:
    print("Eu sou uma função que só faz e não devolve nada!")
```

- `-> void`: Isso aqui é uma maneira de dizer: "Eu não vou devolver nada". Quando a função não precisa devolver nada, colocamos o `-> void` para avisar ao código que ela não vai ter um **resultado** para entregar, ela só vai fazer alguma coisa.

Exemplo Simples: A função `mostrar_mensagem()` apenas mostra a mensagem "Eu sou uma função que só faz e não devolve nada!" na tela. Não precisamos de nenhum resultado de volta, só queremos que a função **faça**.

---

3. Funções Com Ingredientes (Parâmetros) 🍎🍌

Às vezes, sua função precisa de **ingredientes** para fazer algo. Esses **ingredientes** são os **parâmetros** da função. Por exemplo, se você quiser fazer um bolo, precisa de **ingredientes**

como farinha e açúcar. Assim, a função vai usar esses ingredientes para fazer a tarefa dela.

```
func somar(a: int, b: int) -> int:
    return a + b
```

- `a` e `b`: São os **ingredientes** que a função precisa para fazer seu trabalho. No caso da função `somar`, ela precisa de dois números para somar.
- `int`: Quer dizer que `a` e `b` devem ser **números inteiros** (sem casas decimais).
- `return`: Significa que a função vai **devolver o resultado** depois de fazer seu trabalho.

Exemplo Simples: A função `somar(a, b)` pega dois números e devolve a soma deles. Se você passar 2 e 3, ela vai devolver **5**!

```
print(somar(2, 3))  # Saída: 5
```

---

4. Funções Que Trabalham Com Listas (Arrays) 🪣

Uma lista é como uma caixa onde você guarda várias coisas, uma após a outra. Você pode dar uma lista para a função fazer alguma coisa com ela, como adicionar mais coisas na lista ou contar quantos itens tem nela.

Exemplo 1: Passando uma Lista para a Função (e não mudando a original) 🔄

Se você passar uma lista para a função e ela fizer mudanças, mas você não quiser que a lista original seja alterada, você a passa **por valor**.

```
func adicionar_fruta(lista: Array, fruta: String) -> Array:
    lista.append(fruta)  # A função coloca a fruta na lista
    return lista  # A função devolve a nova lista
```

- `lista: Array`: Aqui, `lista` é um **ingrediente** do tipo **Array** (uma lista de itens).
- `fruta: String`: A função também recebe a fruta como **ingrediente** do tipo **String** (um nome, como "Maçã" ou "Banana").
- `.append()`: Esse é o **truque** que a função usa para adicionar uma nova fruta à lista.

Exemplo Simples: A função `adicionar_fruta` pega uma lista e coloca uma nova fruta nela. Porém, a lista original **não muda**, pois a função cria uma nova lista e a devolve.

```
var frutas = ["Maçã", "Banana"]
var nova_lista = adicionar_fruta(frutas, "Laranja")

print(frutas)  # Saída: ["Maçã", "Banana"]
print(nova_lista)  # Saída: ["Maçã", "Banana", "Laranja"]
```

Exemplo 2: Mudando a Lista Original (Por Referência) 🖇️

Se você quiser que a função **mude a lista original**, você pode passá-la **por referência**.

```
func adicionar_fruta_por_referencia(lista: Array, fruta: String) -> void:
    lista.append(fruta)  # A função coloca a fruta na lista original
```

- **Lista Original**: Nesse caso, a função vai modificar a **lista original**, então a fruta vai ser adicionada diretamente à lista que você passou.

```
var frutas = ["Maçã", "Banana"]
adicionar_fruta_por_referencia(frutas, "Laranja")

print(frutas)  # Saída: ["Maçã", "Banana", "Laranja"] (A lista original foi
modificada)
```

---

5. Funções Que Devolvem Algo (Retorno) 🔄

Às vezes, você quer que a função **devolva** algo para você depois de fazer sua tarefa. Esse "algo" pode ser qualquer coisa: um número, uma lista, ou até mesmo uma mensagem.

Exemplo: Função que Retorna o Resultado de uma Tarefa

```
func multiplicar(a: int, b: int) -> int:
    return a * b  # A função devolve o resultado da multiplicação
```

- `return a * b`: Isso significa que a função vai fazer a multiplicação de `a` e `b`, e devolver esse valor para quem chamou a função.

Exemplo Simples: Se você usar `multiplicar(4, 5)`, a função vai devolver **20**!

```
var resultado = multiplicar(4, 5)
print(resultado)  # Saída: 20
```

Conclusão: Como Funciona Cada Parte? 🛠️

- `func nome_da_funcao():` : Declara a função, como se estivesse montando uma máquina.
- **Parâmetros (Ingredientes)**: São os valores que a função precisa para fazer seu trabalho.
- `-> void` : Diz que a função não vai devolver nada (só faz algo).
- `return` : Se a função precisar devolver algo, você usa `return` para entregar o resultado.
- **Listas e Dicionários**: São tipos especiais de "ingredientes" que você pode passar para a função para ela trabalhar com mais coisas, como adicionar itens em uma lista ou atualizar um dicionário.

🛠️ Parte 2: Funções - Conceitos Avançados (Explicado de Forma Simples)

Agora que você já sabe o básico sobre funções, vamos aprender um pouco mais sobre como elas podem ser ainda mais poderosas! Vou explicar tudo de uma maneira bem simples para que você consiga entender cada pedacinho do que está acontecendo dentro do seu código.

---

1. Funções com Vários Parâmetros e Retornos 🧩

Às vezes, uma função precisa de **muitos números ou informações** para fazer o seu trabalho. Vamos imaginar que queremos calcular tanto a **área** quanto o **perímetro** de um retângulo (que são duas coisas diferentes, mas precisam ser calculadas de uma vez só). Para isso, usamos **dois parâmetros**: a **largura** e a **altura**. E a função vai devolver essas duas respostas para a gente.

O que está acontecendo?

- **Parâmetros** são como caixas onde guardamos as informações que vamos usar dentro da função.
- **Retorno** é quando a função nos entrega a resposta depois de fazer a tarefa.

Exemplo: Função que calcula área e perímetro de um retângulo

```
func calcular_area_perimetro(largura: int, altura: int) -> Dictionary:
    var area = largura * altura  # Aqui, estamos multiplicando a largura pela altura para achar a área.
    var perimetro = 2 * (largura + altura)  # O perímetro é a soma de todos os lados.
    return {"area": area, "perimetro": perimetro}  # A função devolve um "dicionário" com as duas respostas.
```

- `largura` e `altura` são os parâmetros (informações que passamos para a função).
- `area` e `perimetro` são os resultados (valores que a função vai calcular e devolver).
- `Dictionary` é uma estrutura que guarda múltiplos valores, como se fosse uma caixa com várias divisórias.

Exemplo de uso:

```
var resultado = calcular_area_perimetro(4, 7)  # Chamamos a função e passamos a largura (4) e a altura (7).
print(resultado["area"])  # Saída: 28
print(resultado["perimetro"])  # Saída: 22
```

- **Resultado**: A função devolve um **dicionário** com dois valores. Você pode pegar esses valores usando `resultado["area"]` e `resultado["perimetro"]`.

---

2. Funções e Objetos 🧸

Às vezes, em vez de passar só números para uma função, queremos passar **objetos**. Um objeto é como uma "coisa" que tem vários **detalhes** ou **informações** sobre ela. No nosso caso, vamos usar um **personagem**, que tem um **nome** e uma **idade**.

O que está acontecendo?

- **Objeto**: É um conjunto de informações, como uma ficha de personagem, que tem vários detalhes dentro dela.
- **Propriedades**: Cada detalhe do objeto, como nome ou idade, é chamado de **propriedade**.

Exemplo: Modificando o nome de um personagem

```
class Personagem:
    var nome: String  # Nome do personagem
    var idade: int    # Idade do personagem

    func _init(n: String, i: int):  # Esta função cria o personagem com nome e
idade.
        nome = n
        idade = i

func mudar_nome(personagem: Personagem, novo_nome: String) -> void:
    personagem.nome = novo_nome  # Aqui estamos mudando o nome do personagem.
    print("Nome alterado para: " + personagem.nome)
```

- `Personagem` é uma classe que cria um **objeto** com um nome e idade.
- `mudar_nome` é uma função que recebe um **objeto** como parâmetro e muda o nome dele.

Exemplo de uso:

```
var personagem1 = Personagem.new("Gerson", 25)  # Criamos um personagem chamado
Gerson, de 25 anos.
mudar_nome(personagem1, "Guilherme")  # Mudamos o nome dele para Guilherme.
```

Agora o **nome do personagem** foi alterado, mas você não precisou criar um **novo personagem**. Apenas **alterou a propriedade** do personagem que já existia.

---

3. Funções Variádicas (Funções com um Número Variável de Argumentos) 🎲

Às vezes, você não sabe quantos valores vai precisar passar para a função. É como se fosse uma **mágica** onde você pode dar à função **qualquer número de números** e ela vai fazer o trabalho direitinho.

O que está acontecendo?

- **Função Variádica**: É uma função que pode receber **qualquer número de parâmetros**. Por exemplo, pode somar **quantos números você quiser**.

Exemplo: Função que soma qualquer quantidade de números

```
func somar_tudo(variavel: Array) -> int:
    var soma = 0
    for numero in variavel:  # A função vai somar todos os números que estão dentro
do array.
        soma += numero
    return soma  # Depois de somar tudo, ela devolve a resposta.
```

- **Array**: É uma lista onde guardamos **múltiplos valores**.
- **Laço de repetição (`for`)**: A função passa por cada número da lista e faz a soma.

Exemplo de uso:

```
var resultado = somar_tudo([1, 2, 3, 4, 5])  # Passamos uma lista com 5 números.
print(resultado)  # Saída: 15
```

- **Por que isso é legal?**: Você pode **passar qualquer número de valores** para a função sem se preocupar com a quantidade exata.

---

4. Funções em Funções (Funções de Ordem Superior) 🎭

Às vezes, você quer usar uma função **dentro de outra função**. Isso é muito útil quando você quer **personalizar** o comportamento da função. Em GDScript, podemos fazer isso passando uma **função como parâmetro**!

O que está acontecendo?

- **Função de Ordem Superior**: É uma função que **recebe outra função** como parâmetro. Assim, você pode **mudar o comportamento** da função dependendo do que você passar para ela.

Exemplo: Usando funções para aplicar filtros

```
func filtrar_numeros(lista: Array, filtro: Callable) -> Array:
    var resultado = []
    for numero in lista:
        if filtro(numero):  # A função 'filtro' vai dizer se o número deve ser
mantido ou não.
            resultado.append(numero)  # Se o número passar no filtro, ele é
```

```
adicionado ao resultado.
    return resultado
```

- `filtro`: Esse é o nome do parâmetro que representa **uma função**. A função `filtrar_numeros` vai usar a função `filtro` para decidir quais números da lista devem ser mantidos.

Exemplo de uso:

```
var lista_numeros = [5, 15, 7, 20, 30]
var numeros_maiores_que_10 = filtrar_numeros(lista_numeros, funcref(self,
"filtro_maior_que_10"))
print(numeros_maiores_que_10)  # Saída: [15, 20, 30]

func filtro_maior_que_10(numero: int) -> bool:
    return numero > 10
```

- `filtro_maior_que_10` é a função que vai **filtrar os números**. Ela vai **devolver** `true` se o número for maior que 10, e `false` se for menor ou igual.

---

5. Funções com Lambda (Funções Anônimas) 🖤

Funções **lambda** são aquelas **funções rápidas e sem nome**, que a gente usa quando não precisa de uma função grandona, mas só de uma coisinha simples. Elas são **ótimas** para tarefas pequenas e rápidas!

O que está acontecendo?

- **Função Lambda**: É uma função simples, que pode ser **usada no lugar de uma função normal** sem precisar de um nome.

Exemplo: Usando uma função lambda para multiplicar números

```
var multiplicar = funcref(self, "multiplicacao")
print(multiplicar.call(3, 4))  # Saída: 12

func multiplicacao(a, b):
    return a * b
```

Aqui, estamos chamando a função `multiplicacao` e pedindo para ela **multiplicar dois números**. Mas a função também poderia ser bem simples e **não precisar de um nome**.

Conclusão: Vamos Resumir! 📝

Agora que aprendemos tudo isso, aqui estão os principais pontos para lembrar:

- **Funções com múltiplos parâmetros** podem retornar vários resultados de uma vez, facilitando o cálculo de várias coisas.
- **Funções que recebem objetos** permitem que você altere os detalhes de uma "coisa" sem criar uma nova toda vez.
- **Funções variádicas** podem receber um número **qualquer de parâmetros**, o que é muito útil para situações onde não sabemos quantos valores vamos precisar.
- **Funções de ordem superior** permitem que você mude o comportamento de uma função passando **outras funções** para ela.
- **Funções Lambda** são funções simples e rápidas, ótimas para tarefas pequenas.

🖇️ Funções Aninhadas

Uma função pode ser definida dentro de outra, sendo acessível apenas no escopo da função externa.

Exemplo:

```
func externa():
    func interna():
        print("Função interna")
    interna()
```

⏭️ Comandos de Controle

- `break`: Interrompe a execução de loops.
- `continue`: Pula para a próxima iteração.

Exemplo:

```
for i in range(10):
    if i == 5:
        break  # Para o loop quando i é 5
    elif i % 2 == 0:
```

```
        continue  # Pula números pares
    print(i)
```

📐 Operações Matemáticas com `Math`

🔢 Funções Úteis

- `Math.abs(x)`: Retorna o valor absoluto de `x`.
- `Math.sqrt(x)`: Retorna a raiz quadrada de `x`.
- `Math.clamp(value, min, max)`: Limita o valor entre `min` e `max`.

Exemplos:

```
print(Math.abs(-10))  # Saída: 10
print(Math.sqrt(16))  # Saída: 4
print(Math.clamp(150, 0, 100))  # Saída: 100
```

🌟 Dica Prática com Movimentação

- Use `Math.lerp(a, b, t)` para suavizar transições.

Exemplo:

```
var posicao_atual: Vector2 = Vector2(0, 0)
var posicao_destino: Vector2 = Vector2(100, 100)
posicao_atual = posicao_atual.lerp(posicao_destino, 0.1)
```

📺 Configurações para Pixel Art

🎨 Texturas

1. Abra **Projeto > Configurações do Projeto > Renderização > Textura**.
2. Defina o filtro para **Nearest** para evitar borrão.

📏 Tela e Resolução

1. Configure a resolução para **320 x 192** no editor.

2. Escale para HD ajustando para **1280 x 768** (4x).
3. Marque a opção **canvas_item**.

🎮 Mapeamento de Inputs

1. Acesse **Projeto > Configurações do Projeto > Entradas Personalizadas**.
2. Acesse **Projeto > Configurações do Projeto > Entradas Personalizadas**.
3. Adicione uma nova entrada para cada ação (como "move_esquerda", "move_direita", etc.).
4. Associe teclas ou botões de controle às ações mapeadas.

Exemplo de Mapeamento de Movimento:

Ação	Tecla	
move_esquerda	A	
move_direita	D	
move_cima	W	
move_baixo	S	

📍 Movimentação do Personagem

📖 Entendendo o Eixo X e Y

- **Eixo X**: Determina o movimento horizontal (esquerda ou direita).
  - Valores negativos indicam movimento para a esquerda.
  - Valores positivos indicam movimento para a direita.
- **Eixo Y**: Determina o movimento vertical (cima ou baixo).
  - Valores negativos indicam movimento para cima.
  - Valores positivos indicam movimento para baixo.

Exemplo de Implementação:

```
var direcao: Vector2 = Input.get_vector("move_esquerda", "move_direita",
"move_cima", "move_baixo")
```

```
if direcao.x > 0:
    print("Movendo para a direita")
elif direcao.x < 0:
    print("Movendo para a esquerda")

if direcao.y > 0:
    print("Movendo para baixo")
elif direcao.y < 0:
    print("Movendo para cima")
```

💡 Boas Práticas

1. **Modularize o código**: Separe funções em blocos lógicos pequenos e reutilizáveis.
2. **Comente o código**: Explique as partes mais complexas.
3. **Teste frequentemente**: Valide a funcionalidade conforme o desenvolvimento.