



CODE

*like a* **girl**

PROJETO DE ENGENHARIA Mulheres em



Instituto Tecnológico de Aeronáutica



Divisão de Ciência da Computação - IEC  
Instituto Tecnológico de Aeronáutica  
Mulheres em STEM<sup>2</sup>D

Projeto de Engenharia Mulheres em STEM<sup>2</sup>D

# Code Like a girl

Fernanda Rodrigues, Isabela Gomes, Letícia Barcelar, Lívia Fragoso,  
Maihara Santos, Stéfanie Fraga e Thayná Baldão

1. Revisora Lara Kühl Teles  
Departamento de Física - IEF  
Instituto Tecnológico de Aeronáutica

*2. Revisora* Juliana de Melo Bezerra  
Divisão de Ciência da Computação - IEC  
Instituto Tecnológico de Aeronáutica

*Supervisoras* Lara Kühl Teles e Thayná Pires Baldão

**Fernanda Rodrigues, Isabela Gomes, Letícia Barcelar, Lívia Fragoso, Maihara Santos, Stéfanie Fraga e Thayná Baldão**

*Code Like a girl*

Projeto de Engenharia Mulheres em STEM<sup>2</sup>D, 24 de Agosto de 2019

Revisoras: Lara Kühl Teles e Juliana de Melo Bezerra

Supervisoras: Lara Kühl Teles e Thayná Pires Baldão

**Instituto Tecnológico de Aeronáutica**

*Mulheres em STEM<sup>2</sup>D*

Instituto Tecnológico de Aeronáutica

Divisão de Ciência da Computação - IEC

Praça Marechal Eduardo Gomes, 50 - Vila das Acacias  
12228-900 e São José dos Campos

# Sumário

|           |   |           |
|-----------|---|-----------|
| <b>1</b>  | <b>Introdução à programação</b>   | <b>5</b>  |
| 1.1       | O que é programação? . . . . .  | 5         |
| 1.2       | O que é lógica de programação? . . . . .                                | 5         |
| 1.3       | Linguagens de Programação . . . . .                                     | 6         |
| 1.3.1     | <i>Python</i> . . . . .   | 7         |
| 1.4       | <i>IDE</i> . . . . .  | 7         |
| 1.5       | Erros de desenvolvimento . . . . .                                      | 9         |
| <b>2</b>  | <b>Utilização de variáveis</b>  | <b>9</b>  |
| 2.1       | Atribuição de variáveis . . . . .                                       | 9         |
| 2.2       | Função <i>type</i> . . . . .  | 9         |
| 2.3       | Nomenclatura de variáveis . . . . .                                     | 10        |
| <b>3</b>  | <b>Impressão na tela</b>  | <b>10</b> |
| <b>4</b>  | <b>Leitura na tela</b>  | <b>11</b> |
| <b>5</b>  | <b>Boas práticas de programação</b>                                     | <b>12</b> |
| 5.1       | Indentar o código . . . . .   | 12        |
| 5.2       | Dar nomes significativos para as variáveis . . . . .                    | 12        |
| 5.3       | Comentar o código . . . . .   | 13        |
| <b>6</b>  | <b>Operadores</b>   | <b>14</b> |
| 6.1       | Operadores aritméticos . . . . .  | 14        |
| 6.2       | Operadores lógicos . . . . .  | 15        |
| 6.3       | Conectivos lógicos . . . . .  | 16        |
| 6.3.1     | <i>AND</i> . . . . .  | 17        |
| 6.3.2     | <i>OR</i> . . . . .   | 17        |
| 6.3.3     | <i>NOT</i> . . . . .  | 18        |
| <b>7</b>  | <b>Comandos de seleção</b>  | <b>19</b> |
| 7.1       | Formatação de expressões condicionais . . . . .                         | 19        |
| 7.2       | O que é uma condição? . . . . .   | 19        |
| 7.3       | O uso de <i>if-else</i> . . . . .                                       | 20        |
| 7.4       | O uso de <i>if-elif-else</i> . . . . .                                  | 20        |
| <b>8</b>  | <b>Comando de repetição</b>   | <b>22</b> |
| 8.1       | <i>for</i> . . . . .  | 22        |
| 8.2       | <i>while</i> . . . . .  | 23        |
| 8.3       | <i>break</i> . . . . .  | 24        |
| <b>9</b>  | <b>Funções</b>  | <b>25</b> |
| 9.1       | Como criar funções em <i>Python</i> ? . . . . .                         | 26        |
| 9.2       | Funções prontas em <i>Python</i> . . . . .                              | 27        |
| 9.3       | Funções de bibliotecas externas . . . . .                               | 28        |
| 9.3.1     | O que significa <b>importar</b> um módulo, ou um pacote? . . . . .      | 28        |
| 9.4       | Vantagens de se utilizar funções no desenvolvimento de código . . . . . | 29        |
| <b>10</b> | <b>Vetores</b>  | <b>30</b> |
| 10.1      | Construção de um vetor . . . . .  | 30        |
| 10.2      | Manipulação de um vetor . . . . .                                       | 31        |
| 10.3      | Operações com vetores . . . . .   | 32        |
| 10.3.1    | Soma . . . . .  | 32        |
| 10.3.2    | Multiplicação . . . . .   | 33        |

|   |           |
|---|-----------|
| <b>11 Strings</b>   | <b>33</b> |
| 11.1 Manipulação de <i>strings</i> . . . . .  | 34        |
| 11.2 Concatenação e multiplicação de <i>strings</i> . . . . .                                     | 35        |
| 11.3 Funções próprias para manipulação de <i>strings</i> . . . . .                                | 35        |
| 11.3.1 Função <code>len()</code> . . . . .  | 35        |
| 11.3.2 Função <code>replace()</code> . . . . .  | 35        |
| 11.3.3 Função <code>count</code> . . . . .  | 36        |
| 11.3.4 Função <code>find</code> . . . . .   | 36        |
| 11.3.5 Função <code>split</code> . . . . .  | 36        |
| 11.3.6 Funções <code>upper</code> e <code>lower</code> . . . . .                                  | 37        |
| <b>12 Introdução a Programação Orientada a Objetos</b>  | <b>37</b> |
| 12.1 Objetos . . . . .  | 38        |
| 12.2 Classes . . . . .  | 38        |
| 12.3 Herança . . . . .  | 39        |
| <b>13 Desenvolvimento do Jogo</b>   | <b>41</b> |
| 13.1 <i>Pygame</i> . . . . .  | 41        |
| 13.2 Telas . . . . .  | 41        |
| 13.2.1 Tela de Início . . . . .   | 44        |
| 13.2.1.1 Inicializando a classe <code>TelaDeInicio</code> . . . . .                               | 45        |
| 13.2.1.2 Desenhando a tela de início . . . . .  | 46        |
| 13.2.1.3 Comportamento do botão de instruções . . . . .   | 47        |
| 13.2.1.4 Comportamento do botão <i>play</i> . . . . .   | 48        |
| 13.2.1.5 Interpretando os eventos disparados pelo usuário . . . . .                               | 48        |
| 13.2.1.6 Executando a tela de início . . . . .  | 49        |
| 13.2.2 Tela de Fim . . . . .  | 49        |
| 13.2.2.1 Inicializando a classe <code>TelaDeFim</code> . . . . .                                  | 50        |
| 13.2.2.2 Desenhando a tela de fim . . . . .   | 51        |
| 13.2.2.3 Comportamento do botão <i>replay</i> . . . . .   | 51        |
| 13.2.3 Tela de Instruções . . . . .   | 51        |
| 13.2.3.1 Inicializando a classe <code>TelaDeInstrucoes</code> . . . . .                           | 52        |
| 13.2.3.2 Método auxiliar <code>imprimirInstrucoes</code> . . . . .                                | 53        |
| 13.2.3.3 Desenhando a tela de instruções . . . . .  | 53        |
| 13.2.3.4 Comportamento do botão <i>play</i> . . . . .   | 53        |
| 13.2.3.5 Comportamento do botão voltar . . . . .  | 54        |
| 13.2.4 Tela de Perguntas . . . . .  | 54        |
| 13.2.4.1 Inserção de perguntas na tela . . . . .  | 55        |
| 13.2.4.2 Inicializando a classe <code>TelaDePerguntas</code> . . . . .                            | 55        |
| 13.2.4.3 Escolha aleatória da pergunta que aparecerá na tela . . . . .                            | 56        |
| 13.2.4.4 Desenhando a tela de perguntas . . . . .   | 57        |
| 13.2.4.5 Comportamento dos botões das alternativas . . . . .                                      | 57        |
| 13.2.4.6 Comportamento do botão pular . . . . .   | 58        |
| 13.2.5 Tela de Resposta de Perguntas . . . . .  | 58        |
| 13.2.5.1 Inicializando a classe <code>TelaResultadoDaPergunta</code> . . . . .                    | 59        |
| 13.2.5.2 Desenhando a tela de resposta de perguntas . . . . .                                     | 60        |
| 13.2.5.3 Comportamento do botão <i>continuar</i> . . . . .  | 60        |
| 13.2.5.4 Incrementando o número de vidas extras do usuário quando ele acerta a pergunta . . . . . | 60        |
| 13.2.6 Tela de Jogo . . . . .   | 61        |
| 13.2.6.1 Elementos do cenário . . . . .   | 61        |
| 13.2.6.2 Aparecimento de elementos do cenário . . . . .   | 62        |
| 13.2.6.3 Efeito de rolagem de paralaxe . . . . .  | 65        |
| 13.2.6.4 Pontuação . . . . .  | 66        |
| 13.3 Jogador . . . . .  | 67        |
| 13.3.1 Movimentos do jogador . . . . .  | 68        |
| 13.3.1.1 Identificação das teclas pressionadas . . . . .  | 68        |
| 13.3.1.2 Andar . . . . .  | 69        |
| 13.4 Pular . . . . .  | 71        |

|          |   |           |
|----------|---|-----------|
| 13.4.1   | Tiro . . . . .  | 73        |
| 13.4.1.1 | Disparo . . . . .   | 74        |
| 13.4.1.2 | Movimento do tiro . . . . .                               | 74        |
| 13.5     | Inimigos . . . . .  | 74        |
| 13.5.1   | Movimentos do inimigo . . . . .                           | 74        |
| 13.5.2   | Tiros do inimigo . . . . .                                | 75        |
| 13.6     | Colisões . . . . .  | 76        |
| 13.6.1   | Colisões com obstáculos . . . . .                         | 76        |
| 13.6.2   | Colisões com vidas . . . . .                              | 76        |
| 13.6.3   | Colisões com impulsionadores . . . . .                    | 76        |
| 13.6.4   | Colisões com inimigos . . . . .                           | 77        |
| 13.6.5   | Colisões com tiros . . . . .                              | 77        |
| 13.7     | Sonorização . . . . .                                     | 77        |
| 13.7.1   | Deixando o jogo mudo . . . . .                            | 77        |
| 13.7.2   | Tocando efeitos sonoros e a música de fundo . . . . .     | 78        |
| 13.7.3   | Adicionando efeitos sonoros e a música de fundo . . . . . | 78        |
|          | <b>Glossário</b>  | <b>80</b> |
|          | <b>Siglas</b>   | <b>81</b> |

# 1 Introdução à programação

## 1.1 O que é programação?

Esse questionamento parece óbvio demais, mas não estamos falando sobre o planejamento que você faz para as férias ou o rumo que dá ao dinheiro que sobra no final do mês. Estamos nos referindo a programação de computadores.

Nesse contexto, a programação é o ato de escrever códigos que instruem um computador a realizar uma tarefa. Essa tarefa pode ser tão simples quanto somar dois números ou tão complexa quanto enviar uma nave espacial para a órbita!



## 1.2 O que é lógica de programação?

Lógica de Programação é a técnica de encadear ideias para atingir determinado objetivo. Essas ideias são representadas como instruções. No contexto de criação de programas, uma **instrução** pode ser definida como a informação que indica a um computador uma ação elementar a executar, por exemplo, ler as notas das provas, calcular a média das notas, verificar situação de aprovação, imprimir o resultado na tela.

Essa sequência lógica de instruções deve ser seguida para se cumprir uma determinada tarefa. Nesse sentido, podemos entender uma sequência lógica de instruções como passos, logicamente encadeados, que são executados para atingir um objetivo ou solução de um problema. Assim, para obtermos o resultado, precisamos colocar em prática o conjunto de todas as instruções, na ordem correta.

Para representar a sequência lógica de instruções, nós fazemos uso de **algoritmos**. Um algoritmo pode ser definido de várias formas. De forma informal, um algoritmo pode ser definido como uma “receita” para se executar uma tarefa ou resolver algum problema. Embora às vezes não percebamos, utilizamos algoritmos no nosso dia-a-dia e não sabemos.

Imagine o trabalho de um recepcionista de cinema, ele deve conferir os bilhetes e direcionar o cliente para a sala correta. Além disso, se o cliente estiver 30 minutos adiantado o recepcionista deve informar que a sala do filme ainda não está aberta. E quando o cliente estiver 30 minutos atrasado o recepcionista deve informar que a entrada não é mais permitida. Observe o Algoritmo 1, o qual descreve a atividade do recepcionista.

---

**Algoritmo 1** Repcionista

---

**início**

Solicitar ao cliente o bilhete do filme.

Conferir a data e o horário do filme no bilhete.

**se** *data/hora atual > data/hora do filme + 30 minutos* **então**

Informar ao cliente que o tempo limite para entrada foi excedido.

Não permitir a entrada.

**senão se** *data/hora atual < data/hora do filme - 30 minutos* **então**

Informar ao cliente que a sala do filme ainda não foi liberada para entrada.

Não permitir a entrada.

**senão**

Permitir a entrada.

Indicar ao cliente onde fica a sala do filme.

**fim****fim**

---

Você concorda que qualquer pessoa que seguisse esses passos seria capaz de executar a função do recepcionista do cinema? É importante notar, ainda, que o algoritmo tem um fluxo que pode seguir diferentes caminhos dependendo da situação em que se encontra.

Agora que você já possui uma intuição do que é um algoritmo, vamos descrevê-lo mais formalmente: algoritmo é uma sequência finita e não ambígua de passos para a resolução de um problema. Os algoritmos são finitos porque, para que sejam úteis, uma hora eles tem que acabar! Além disso, eles devem ser não-ambíguos pois o computador não pode ter dúvidas do que fazer para executar cada passo. Do contrário, erros podem acontecer.



É importante ressaltar que diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferenciado de instruções em mais ou menos tempo, espaço ou esforço do que outros. Além disso, todas as tarefas executadas pelo computador, são baseadas em algoritmos.

### 1.3 Linguagens de Programação

Uma **linguagem de programação** é um método padronizado para expressar instruções para um computador. É um conjunto de regras sintáticas e semânticas usadas para definir um programa. O conjunto de palavras, compostos de acordo com essas regras, constituem o **código-fonte** de um *software*.

Basicamente, para se desenvolver um código-fonte, o programador deve criar um algoritmo e, então, traduzi-lo para alguma linguagem de programação.

Existem diversas linguagens de programação, por exemplo: *C*, *C++*, *Java*, *PHP*, *Pascal*, *Python*, etc. Neste projeto, utilizaremos a linguagem de programação *Python*.

Depois de escrever o programa, um compilador converte o código-fonte em **código de máquina**, isto é, a linguagem de nível mais baixo para um computador. Então, o código de máquina instrui a unidade central de processamento, ou *Central Processing Unit (Unidade Central de Processamento) (CPU)*, sobre as etapas a serem seguidas, como carregar um valor ou executar alguma aritmética. Se você já ouviu alguém dizer “Eu compilei meu programa”, isso significa que eles converteram códigos em código de máquina.

Mas se o computador interpreta apenas código de máquina, por que não escrevemos código de máquina diretamente? A razão é que o código de máquina é ilegível para humanos. Compare, por exemplo, a versão *Python* de um programa (Código 1) com seu conjunto correspondente de linguagem de máquina (Código 2).

```
1 print('Hello, World!')
```

Código 1: Programa para impressão na tela em *Python*.

```
1 c7 3c 2a 3c 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c 3c
2 28 5c 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c 3c 28 5c
3 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c 3c 28 5c 2a 2b
4 2a 5c 3c 28 5c 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c
5 3c 28 5c 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c 3c 28
6 5c 2a 2b 2a 5c 3c 28 5c 2a 2b 2a 5c 3c 28 5c 2a
7 2b 2a 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13 00 00 00 00 00 00 00 64 48 65 6c 6c 6f 2c 20 57
14 6f 72 6c 64 21 00 00 00 00 00 00 00 00 00 00 00 00
15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
17 Hello, World!
```

Código 2: Programa para impressão na tela em linguagem de máquina.

Após olhar esses exemplos, fica bem evidente por que você não quer programar diretamente em linguagem de máquina.

### 1.3.1 Python

*Python* é uma linguagem de programação criada por Guido van Rossum, em 1991. Os objetivos do projeto da linguagem eram: produtividade e legibilidade. Em outras palavras, *Python* é uma linguagem que foi criada para produzir código bom e fácil de manter de maneira rápida.

*Python* tem uma biblioteca padrão imensa, que contém classes, métodos e funções para realizar essencialmente qualquer tarefa, desde acesso a bancos de dados a interfaces gráficas com o usuário. Por causa de todas essas vantagens, esta será a linguagem utilizada neste curso.

Há um detalhe menor que deixamos de fora: *Python* é o que chamamos de **linguagem interpretada**, isto é, você não a compila diretamente em código de máquina, como mencionamos acima. Ao invés disso, o *Python* usa algo chamado **interpretador**.

Um interpretador é outro programa que compila o código em algo chamado *Bytecode*, que é um formato de código intermediário entre o código-fonte, o texto que o programador consegue manipular, e o código de máquina. Então, o *Bytecode* é traduzido para linguagem de máquina enquanto o programa é executado. É importante ressaltar, contudo, que você não precisa realmente entender o funcionamento interno dos compiladores para começar a programar em *Python*, tudo que foi falado nesta Subseção é apenas curiosidade.

## 1.4 IDE

Considerando que você tenha instalado na sua máquina o *Python*, uma opção para desenvolver programas seria escrever o código em *Python* em um editor de texto qualquer e salvar o arquivo com a extensão

.py. Posteriormente, bastaria abrir um terminal (Prompt de Comando no Windows) e manualmente executar seu código, desde que você possuisse o *Python* instalado em sua máquina.

Contudo, um editor de texto qualquer (não específico para programar) não possui ferramentas de correção do seu código. Isso quer dizer que se você sem querer digitar uma letra errada ou esquecer de algo, o editor de texto não irá te alertar do erro. Você só irá ser notificado que algo está errado quando ele parar de funcionar durante a execução.

Por causa disso, com o intuito de facilitar a vida do desenvolvedor, criou-se o **IDE**, do inglês *Integrated Development Environment* ou **Ambiente de Desenvolvimento Integrado**, isto é, um software que contém todas as funções necessárias para o desenvolvimento de programas de computador, assim como alguns recursos que diminuem a ocorrência de erros nas linhas de código. As características e ferramentas mais comuns encontradas nos IDEs são:

- Editor: edita o código-fonte do programa escrito na(s) linguagem(ns) suportada(s) pela IDE;
- Compilador (*compiler*): compila o código-fonte do programa, editado em uma linguagem específica e a transforma em linguagem de máquina;
- Linker: liga (“*linka*”) os vários “pedaços” de código-fonte, compilados em linguagem de máquina, em um programa executável que pode ser executado em um computador ou outro dispositivo computacional;
- Depurador (*debugger*): auxilia no processo de encontrar e corrigir defeitos no código-fonte do programa, na tentativa de aprimorar a qualidade do software;
- Testes Automatizados (*automated tests*): realiza testes no software de forma automatizada, com base em scripts ou programas de testes previamente especificados, gerando um relatório, e assim auxiliando na análise do impacto das alterações no código-fonte;

As principais vantagens de se utilizar um Integrated Development Environment (Ambiente de Desenvolvimento Integrado) (IDE) durante o desenvolvimento de código são:

- Notifica em tempo real erros de sintaxe;
- Faz sugestões para correção de erros;
- Auto-completa códigos;
- Permite “debugar” o programa;
- Altera cores de palavras-chaves da linguagem no seu código, tornando-o mais legível;
- O processo de execução do programa pode ser feito com o *click* de um botão e o *output* visualizado diretamente na IDE;

Dadas essas vantagens, neste curso utilizaremos uma *IDE* para o desenvolvimento do nosso jogo, denominada *PyCharm*. O *PyCharm* possui uma interface muito limpa e personalizável, ideal para aqueles que estão dando os primeiros passos com a linguagem *Python*. Além disso, se você se esquecer como cria uma função, o *PyCharm* lhe ajudará com um passo a passo ou com o manual da biblioteca que possui aquela função, dentro da própria aplicação. Isso sem falar das suas funcionalidades clássicas de *console* e *debugger*.



Figura 1: Logotipo do *PyCharm*.

Não obstante, é importante ressaltar que o *IDE* facilita e muito na programação, mas ele não é mágico e não realiza as atividades para o desenvolvedor. O *IDE* te indica o erro ou falha na linguagem, porém ele não vai corrigir o seu código automaticamente, cabe a você resolver o problema encontrado. Por causa disso, é necessário possuir conhecimento das linguagens de programação para realizar as atividades corretamente.

## 1.5 Erros de desenvolvimento

É natural que, durante o desenvolvimento de programas para o computador, nos deparamos com **algunserros** que precisam ser corrigidos para o correto funcionamento do programa. Em geral, os erros são causados por dois principais motivos:

1. Escrita incorreta da sintaxe da linguagem de programação;
2. Algoritmo com falhas lógicas, que fazem com que o comportamento do programa divirja do esperado.

Em geral, os compiladores e os interpretadores avisam os programadores, por meio de mensagens, problemas de sintaxe do código, o que torna a identificação do primeiro tipo de erro muito mais fácil que o do segundo, o qual, normalmente, requer que o programador revise os passos de seu algoritmo.



## 2 Utilização de variáveis

**Variáveis** são um dos recursos mais básicos das linguagens de programação. Utilizadas para armazenar valores em memória, elas nos permitem gravar e ler esses dados com facilidade a partir de um nome definido por nós.

### 2.1 Atribuição de variáveis

Assim como em outras linguagens, o *Python* pode manipular variáveis básicas como palavras ou cadeias de caracteres (**string**), números inteiros (**int**) e números reais (**float**). Para criá-las, basta utilizar um comando de atribuição, que define seu tipo e seu valor, conforme vemos no Código 3.

```
1 mensagem = 'Exemplo de mensagem!'
2 n = 25
3 pi = 3.141592653589931
```

Código 3: Atribuições de variáveis do tipo **string**, **int** e **float** respectivamente.

### 2.2 Função type

Observe que não foi necessário fazer uma declaração explícita de cada variável, indicando o tipo ao qual ela pertence, pois isso é definido pelo valor que ela armazena. Para saber o tipo de determinado objeto, usamos a função **type(x)**, que retorna o tipo do objeto x, conforme o Código 4

```
1 type (mensagem)
2 type (n)
3 type (pi)
```

Código 4: Utilizando a função `type(x)` para obtenção do respectivos tipos das variáveis declaradas.

O resultado da execução deste código está explicitado no Código 5.

```
1 <class 'str'>
2 <class 'int'>
3 <class 'float'>
```

Código 5: Resultado dos tipos das variáveis declaradas.

## 2.3 Nomenclatura de variáveis

As variáveis podem ser nomeadas conforme a vontade do programador<sup>1</sup>, com nomes contendo **letras e números**. No entanto, elas devem necessariamente começar com letras minúsculas. Lembre-se de não utilizar espaços ou símbolos. Também é importante estar atento às palavras reservadas da linguagem listadas na Figura 2, que não podem ser utilizadas para nomear variáveis.

|        |          |         |          |        |
|--------|----------|---------|----------|--------|
| False  | class    | finally | is       | return |
| None   | continue | for     | lambda   | try    |
| True   | def      | from    | nonlocal | while  |
| and    | del      | global  | not      | with   |
| as     | elif     | if      | or       | yield  |
| assert | else     | import  | pass     |        |
| break  | except   | in      | raise    |        |

Figura 2: Palavras reservadas na linguagem *Python*.

## 3 Impressão na tela

Exibir coisas na tela é, sem dúvida, a comunicação mais básica e uma das mais importantes. Afinal, você não sabe o que um programa está fazendo por trás dos panos, você não vê bits se movendo, nem a memória ou o processador trabalhando. Portanto, precisamos de comunicação humana com nossos programas em *Python*, e isso é feito através da impressão de informações na tela, por meio da função `print`.

A grosso modo a função `print` serve para imprimir os argumentos passados a ela no terminal. Basicamente, para utilizá-la, basta escrever `print(< o que se deseja imprimir >)`. Para testarmos a função `print()` abra o *PyCharm* e implemente o Código 6.

```
1 print('Olá, mundo!')
```

Código 6: Primeiro exemplo de utilização da função de impressão na tela `print`.

Para escrever um texto, basta colocar entre aspas simples ou aspas duplas o que se deseja imprimir.



### 1. Faça um programa que imprima seu nome.

É importante ressaltar que as linguagens de programação, em geral, não funcionam com acentos ortográficos, de modo que você **não** deve utilizá-los em seus códigos em *Python*.

Por outro lado, se você quiser imprimir o valor armazenado em uma variável, deve escrever o nome desta dentro do comando `print`, como no Código 7.

<sup>1</sup> Apesar da liberdade para se escolher nomes de variáveis, recomenda-se fortemente seguir os padrões de nomenclaturas discutidos na Subseção 5.2.

```
1 seunome = 'Larissa'
2 print(seunome)
```

Código 7: Código para a impressão na tela do valor de uma variável.



### 1. Faça um programa que guarde seu nome em uma variável e a imprima.

Se for necessário imprimir o valor de uma variável no meio de um texto, é preciso avisar ao código o que se deseja imprimir. Para isso, colocamos o texto entre aspas simples e, no lugar onde vai ser impresso o valor da variável, coloca-se um abrir e fechar de chaves ("{}"). Depois de fechar as aspas simples do texto, coloque o método `.format()` e, como argumento, a variável que deseja imprimir. Um exemplo de impressão de texto com variável está descrito no Código 8.

```
1 print('Meu nome eh {}'.format(seunome))
```

Código 8: Código para a impressão na tela do valor de uma variável em texto.

Para imprimir mais de uma variável no texto, basta colocá-las como argumento, na ordem em que elas devem aparecer, conforme exemplificado no Código 9.

```
1 print('Meu nome eh {} e tenho {} anos'.format(seunome, idade))
```

Código 9: Código para a impressão na tela do valor de mais de uma variável em texto.

Se tivermos uma variável que armazena um número real (tipo `float`, por exemplo), você pode colocar dentro das chaves que indicam a variável o número de casas decimais que você deseja imprimir: `.(<número de casas decimais>)f`. Se você quiser imprimir sua altura com duas casas decimais, seu código deve estar formatado conforme o Código 10.

```
1 altura = 1.6789
2 print('Minha altura eh {:.2f}'.format(altura))
```

Código 10: Código para a impressão na tela de uma variável com duas casas decimais em texto.

Por fim, se você quiser pular uma linha, basta colocar um `print` vazio, ou ainda, escrever a sequência de caracteres `\n` dentro de um texto, conforme descrito no Código 11.

```
1 print()
2 print('Vamos pular uma linha.\nE mais uma linha.')
```

Código 11: Código para pular uma linha.

## 4 Leitura na tela

Além de imprimir, muitas vezes precisamos que o programa leia e armazene coisas. Para isso, utilizamos a função `input`. Se você quiser que o usuário escreva na tela o valor a ser armazenado em uma variável, utilize algo semelhante ao Código 12.

```
1 variavel = input()
```

Código 12: Código para ler na tela.

Para avisar ao usuário o que ele deve escrever, é possível colocar texto explicando, o qual será mostrado logo antes da requisição, como exemplificado no Código 13.

```
1 variavel = input('O valor da variavel eh: ')
```

Código 13: Código para ler na tela com explicação.

É importante saber que o `input` sempre guarda uma variável do tipo `string`. Se for desejado outro tipo de variável, é preciso fazer conversões<sup>2</sup>, como descrito no Código 14.

```
1 variavel = int(input('O valor da variavel int eh: '))
```

Código 14: Código para ler na tela uma variável int.

<sup>2</sup>Será falado melhor sobre conversão de variáveis na Seção 11.



1. Faça um programa que requisite ao usuário seu nome e responda “Bom dia, <nome da pessoa>”.

## 5 Boas práticas de programação

Um bom código não é somente aquele que é funcional, mas também aquele que é fácil de compreender. Por causa disso, existem uma série de premissas essenciais para se desenvolver um programa. Portanto, é importante se atentar para as recomendações que serão descritas nas próximas Subseções.

### 5.1 Indentar o código

A indentação consiste na adição de tabulações no início de cada linha, na quantidade equivalente ao número de blocos em que cada linha está contida. No *Python*, a tabulação é feita quando você pressiona a tecla *tab* ou quando você insere 4 espaços seguidos. O código que está na mesma posição (recuado o mesmo número de espaços da margem esquerda) é agrupado em um bloco e, sempre que você inicia uma nova linha com mais espaços que o anterior, você está iniciando um novo bloco que faz parte do bloco anterior. Sem uma boa indentação o código perde a fácil visualização da hierarquia de seus comandos. Verifique a diferença de clareza entre o Código 15, que possui indentação, e o Código 16, que não possui indentação.

```
1 def atualizacaoBasica(self):
2     if self.x > -200:
3         self.x -= self.largura
4     self.rect.x = self.x
5     self.rect.y = self.y
```

Código 15: Código de um método em *Python* indentado.

```
1 def atualizacaoBasica(self):
2 if self.x > -200:
3 self.x -= self.largura
4 self.rect.x = self.x
5 self.rect.y = self.y
```

Código 16: Código de um método em *Python* não indentado.

Na maioria das linguagens, a indentação tem um papel meramente estético (tornando a leitura do código fonte muito mais fácil), porém em algumas linguagens, como *Python*, a indentação é obrigatória, já que substitui os identificadores de blocos. No exemplo do 15, a linha 3 pertence ao *if*<sup>3</sup>, ou seja, será executada somente se a *idade* for maior ou igual a 18. Dessa maneira, o Código 16, que não possui indentação, sequer consegue ser executado em *Python*. Sendo assim, é **extremamente importante utilizar indentação** ao se desenvolver códigos nesta linguagem.

### 5.2 Dar nomes significativos para as variáveis

Colocando nomes adequados e padronizados, você passará informações que ajudarão na compreensão do código. Por causa disso, recomenda-se fortemente utilizar as seguintes convenções para nomear variáveis:

1. Criar nomes curtos (por exemplo: *nota*, *primeiroNome*, *mediaAluno*) e significativos, em que ao bater o olho no nome, o programador já saiba que informação a variável vai armazenar.
2. Nomear as variáveis, com exceção das constantes, conforme o padrão **camelCase**. Esse padrão consiste em declarar variáveis com nomes compostos (mais de um nome), fazendo com que as palavras sejam unidas sem espaço, e iniciem, com exceção da primeira palavra, por letras maiúsculas, por exemplo: *imagemInvencivel*, *telaAtual*.

<sup>3</sup>O comando *if* será explicado mais adiante na Seção 7.

3. Em variáveis que possuem um valor fixo, isto é, constantes, (e apenas nelas) todas as letras devem estar em maiúsculo e com as palavras separadas por sublinhado (“\_”), por exemplo: TAXA\_IMPOSTO;
4. Utilizar nomes passíveis de busca: Evitar declarar nomes de variáveis de apenas um caractere (i, j, x, y), que tornam muito mais difícil a busca de tais variáveis, a não ser para aquelas usadas para índices em laços de repetição<sup>4</sup> onde o algoritmo percorre vetores.
5. Preferir declarar nomes pronunciáveis: declarar nomes de variáveis como hndlimghght torna muito mais difícil de memorizar o seu propósito e, até mesmo, de citar esta variável quando está se pedindo auxílio de alguém para debuggar uma parte do código.



### 5.3 Comentar o código

Muitos programadores ignoram a importância dos comentários no seu arquivo fonte, geralmente por falta de conscientização dessa boa prática para que o código não gera dor de cabeça no futuro. Você pode usar comentários com a finalidade de explicar o algoritmo ou a lógica usada, mostrando o objetivo de uma variável, método, classe, etc.

Sendo assim, para viabilizar a manutenção de um programa, é fundamental, além de dar nomes significativos às variáveis e utilizar indentação, fazer comentários. Para se realizar comentários em *Python* basta utilizar o carácter `#` e escrever seu comentário depois disso. Observe no Código 17, um exemplo de programa em *Python* que possui comentários.

```

1 def interpretarEventos(self, game):
2     game.clock.tick(game.fps)
3
4     for evento in pygame.event.get():
5         pos = pygame.mouse.get_pos()
6
7         # checa se o usuario quer sair do jogo
8         self.comportamentoBotaoDeSair(game, evento)
9
10        # checa se o usuario quer tirar o som
11        self.comportamentoBotaoDeAudio(game, evento, pos)
12
13        # checa se o usuario clicou no botao para abrir a tela de instrucoes
14        self.comportamentoBotaoDeInstrucoes(game, evento, pos)
15
16        # checa se o usuario quer jogar
17        self.comportamentoBotaoDeJogar(game, evento, pos)

```

<sup>4</sup>Laços de repetição serão abordados com mais profundidade na Seção 8.

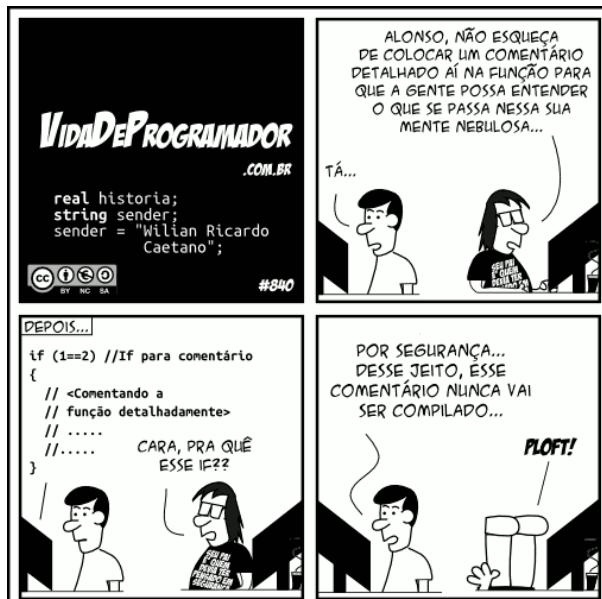
18  
19

```
# print("pos0: ", pos[0], " pos1: ", pos[1])
```

Código 17: Exemplo de programa com comentários.

Os comentários servem para explicar seu código e não são interpretados na execução do programa, servindo, exclusivamente, para melhorar a clareza do programa. Sendo assim, no Código 17, o método `print` não será executado, pois está comentado.

Cansamos de ouvir “tudo o que é demais pode ser ruim”. Essa premissa vale para os comentários, que podem deixar o código poluído, difícil de ler e atualizar. Por isso, tenha bom senso quando comentar as linhas do sistema, colocando textos curtos e somente aqueles que você achar relevante para a leitura e principalmente para a compreensão do código. E quando achar necessário colocar um comentário para que outras pessoas consigam entender o algoritmo, procure rever o seu código. Talvez melhorando ele, nem seja necessário colocar essas informações adicionais.



## 6 Operadores

### 6.1 Operadores aritméticos

Operadores são símbolos especiais que representam cálculos como adições e multiplicações. Para fazer cálculos com números utilizamos os operadores `+`, `-`, `*`, `/` e `**` que representam, respectivamente, adição, subtração, multiplicação, divisão e potenciação.

Uma expressão é uma combinação de valores, variáveis e operadores como `x + 17`, `1 + 1`, etc. Quando digitamos uma expressão no modo interativo, o interpretador vai calcular e imprimir o seu resultado na tela. Por exemplo, o Código 18 tem como resultado na tela o Código 19.

1 1 + 1  
2 2 \* 3

Código 18: Exemplos de operações aritméticas com valores numéricos.

1 2  
2 6

Código 19: Resultado na tela do Código 18.

Além de usar números, também podemos usar variáveis. Para isso consideremos o exemplo apresentado no Código 20, nele, primeiramente, atribuímos o valor 1 para a variável `x` e 3 para a variável `y`, em seguida realizamos as operações que desejamos, como exemplificado no Código 20, cujo resultado na tela é o Código 21.

1 x = 1  
2 y = 3

```
3 x + y
4 x - y
5 x * y
6 x / y
7 x ** y
```

Código 20: Exemplos de operações aritméticas com variáveis.

```
1 4
2 -2
3 3
4 0.3333333333333333
5 1
```

Código 21: Resultado na tela do Código 20.

Além dos operadores já citados, temos também o operador `//` que representa a divisão inteira, e o operador módulo `%`, que resulta no resto da divisão entre dois números inteiros. Um exemplo de utilização destes operadores está apresentado no Código 22, cujo resultado na tela é o Código 23.

```
1 7 // 2
2 7 % 3
```

Código 22: Exemplo das operações divisão inteira e módulo respectivamente.

```
1 3
2 1
```

Código 23: Resultado do Código 23.

## 6.2 Operadores lógicos

O *Python* possui poucas constantes embutidas. As mais utilizadas são `True`, `False` e `None`. Essas também são palavras-chaves do *Python*, portanto palavras reservadas que não podemos utilizar como nomes de variáveis.

`True` e `False` são valores booleanos que representam, respectivamente, verdadeiro e falso. O *Python* também possui a função `bool()` que retorna `True` quando o argumento passado é verdadeiro e retorna `False`, caso contrário.

Podemos representar `True` e `False` através de expressões. Por exemplo “O número 1 é igual a string ‘1’?”, vamos perguntar ao *Python* se isso é verdade, como descrito no Código 24.

```
1 1 == '1'
```

Código 24: Exemplo de operação lógica.

A resposta que obtemos para o Código 24 está descrita no Código 25.

```
1 False
```

Código 25: Resposta na tela do Código 24.

O operador `==` é usado para verificar se algo é igual a outro. **Não confunda-o com o `=`, o qual atribui um valor a uma variável**. Também podemos verificar se um número é maior, utilizando o operador `>`, ou menor (`<`) do que outro, como exemplificado no Código 26.

```
1 2 > 1
2 2 < 1
```

Código 26: Exemplo numérico com operadores lógicos.

A resposta que obtemos para o Código 26 está descrita no Código 27.

```
1 True
2 False
```

Código 27: Resposta na tela do Código 26.

Podemos também utilizar a função `bool()` para fazer a verificação, como descrito no Código 28.

```
1 bool(1 == 1)
```

Código 28: Exemplo da utilização da função `bool()`.

A resposta que obtemos para o Código 28 está descrita no Código 29.

```
1 True
```

Código 29: Resposta na tela do Código 28.

Na Tabela 1, temos uma lista com os chamados operadores de comparação, os quais retornam valores *booleanos* como `True` (verdadeiro) e `False` (falso).

Tabela 1: Operadores de comparação.

| Operação               | Descrição                                      |
|------------------------|--|
| <code>a == b</code>    | <code>a</code> igual a <code>b</code>          |
| <code>a != b</code>    | <code>a</code> diferente de <code>b</code>     |
| <code>a &lt; b</code>  | <code>a</code> menor do que <code>b</code>     |
| <code>a &gt; b</code>  | <code>a</code> maior do que <code>b</code>     |
| <code>a &lt;= b</code> | <code>a</code> menor ou igual a <code>b</code> |
| <code>a &gt;= b</code> | <code>a</code> maior ou igual a <code>b</code> |

Outros operadores que retornam valores *booleanos* estão representados na Tabela 2.

Tabela 2: Operadores de comparação que retornam valores booleanos.

| Operação                | Descrição  |
|-------------------------|--|
| <code>a is b</code>     | <code>True</code> se <code>a</code> e <code>b</code> são idênticos     |
| <code>a is not b</code> | <code>True</code> se <code>a</code> e <code>b</code> não são idênticos |
| <code>a in b</code>     | <code>True</code> se <code>a</code> é membro de <code>b</code>         |
| <code>a not in b</code> | <code>True</code> se <code>a</code> não é membro de <code>b</code>     |

### 6.3 Conectivos lógicos

Até o momento, realizamos apenas um teste por vez com os operadores lógicos, cujo resultado da expressão era uma *booleana*. Os conectivos lógicos, por sua vez, nos permitem relacionar várias expressões lógicas em uma única expressão.

As linguagens de programação utilizam os conectivos lógicos na construção de expressões lógicas. Existem dois conectivos lógicos e, mesmo que não os conheçamos por esse nome, utilizamo-os constantemente ao conversarmos ou então, para explicarmos coisas para outras pessoas.

Os dois conectivos lógicos são:

- Conectivo de conjunção: `AND`;
- Conectivo de disjunção: `OR`.

Por exemplo, a simples frase “A e B são caracteres iguais” implica numa expressão lógica e acabamos de representá-la textualmente. Porém, a expressão pode ser facilmente escrita matematicamente, ou então, com o uso de uma linguagem de programação. Por essa razão, devemos olhar para as linguagens de programação como sendo, antes de tudo, formas ou estilos de notação lógica.

Os conectivos lógicos devem ser entendidos como ferramentas de notação utilizadas para unir duas ou mais expressões, e como resultado da união, forma-se uma nova expressão. Eles são comumente utilizados para definir se certa função ou operação será realizada ou não dependendo do valor lógico retornado pela expressão. O resultado desta nova expressão depende do resultado de cada sub-expressão que a compõe e do operador utilizado.

A tabela verdade das operações condicionais é um mecanismo que nos auxilia a definir o valor lógico de uma proposição, isto é, saber quando uma sentença é verdadeira ou falsa.

### 6.3.1 AND

Para que a expressão conectada por `and` seja verdadeira, todas as suas sub-expressões devem ser verdadeiras. Uma única sub-expressão falsa torna toda expressão falsa. Vejamos na Tabela 3 como o conectivo funciona para duas sub-expressões.

Tabela 3: Tabela verdade para o operador `and`.

| a     | b     | a and b |
|-------|-------|---------|
| True  | True  | True    |
| True  | False | False   |
| False | True  | False   |
| False | False | False   |

Um exemplo de aplicação do operador condicional `and` está descrito no Código 30, cujo resultado na tela está descrito no Código 31.

```

1 x = 1
2 y = 4
3 nome = 'STEM'
4 (x > 0) and (x < 100)
5 (x != 0) and (y > 4)
6 (x != 0) and (y >= 4)
7 (x + y < 10) and (nome == 'livro') and (y%3 == 1)
8 (x + y < 10) and (nome == 'STEM') and (y%3 == 1)

```

Código 30: Exemplos de expressões lógicas com o operador `and`.

```

1 True
2 False
3 True
4 False
5 True

```

Código 31: Resultado na tela do Código 30.

### 6.3.2 OR

Para que a expressão conectada por `or` seja verdadeira, pelo menos uma de suas sub-expressões deve ser verdadeira. Uma única sub-expressão verdadeira torna toda expressão verdadeira e apenas se todas as sub-expressões forem falsas, a expressão resultante será falsa. Vejamos na Tabela 4 como o conectivo funciona para duas sub-expressões.

Tabela 4: Tabela verdade para o operador `or`.

| a     | b     | a or b |
|-------|-------|--------|
| True  | True  | True   |
| True  | False | True   |
| False | True  | True   |
| False | False | False  |

Um exemplo de aplicação do operador condicional `and` está descrito no Código 32, cujo resultado na tela está descrito no Código 33.

```

1 x = 1
2 y = 4
3 nome = 'STEM'
4 (x > 0) or (x < 100)
5 (x == 0) or (y > 4)
6 (x == 0) or (y >= 4)
7 (x + y < 10) or (nome == 'livro') or (y%3 == 1)
8 (x*y%2 == 1) or (nome == 'STEM') or (y/2 == 1) or (3 > 5) or (x == 6)

```

Código 32: Exemplos de expressões lógicas com o operador `or`.

```

1 True
2 False
3 True
4 True
5 True

```

Código 33: Resultado na tela do Código 32.

### 6.3.3 NOT

O operador `not`, cuja tabela verdade está explicitada na Tabela 5, não conecta expressões lógicas, no entanto, é utilizado para negar uma expressão invertendo seu resultado lógico. Ou seja, uma expressão antes verdadeira torna-se falsa e vice-versa.

Tabela 5: Tabela verdade para o operador `not`.

| a     | not a |
|-------|-------|
| True  | False |
| False | True  |

Um exemplo de aplicação do operador `not` está descrito no Código 34, cujo resultado na tela está descrito no Código 35.

```

1 x = 1
2 y = 4
3 nome = 'STEM'
4 not (x*y%2 == 1)
5 True
6 (x + y < 10) and (nome == 'STEM') and (y%3 == 1)
7 True
8 (x + y < 10) and not (nome == 'STEM') and (y%3 == 1)
9 False

```

Código 34: Exemplos de expressões lógicas com o operador `not`.

```

1 True
2 True
3 False

```

Código 35: Resultado na tela do Código 34.

Vimos nos exemplos que podemos utilizar os mais diversos operadores e expressões lógicas na construção de uma nova expressão. Tais operadores podem relacionar duas ou mais expressões, mas à medida que adicionamos condições, sua interpretação torna-se mais complexa. Além disso, podemos intercalar operadores. Vejamos alguns exemplos no Código 36, cujo resultado na tela está descrito no Código cod:resteste13.

```

1 x = 1
2 y = 4
3 nome = 'STEM'
4 (x > 0) and (x < 100) or (x + y < 100)
5 ((nome == 'livro') or (x + y < 100)) and (y >= 4)
6 ((nome == 'livro') and (x + y < 100)) or (y > 4)
7 ((nome == 'STEM') or (y/x < 0)) and ((nome == 'livro') or ((y%3 == 1)))

```

Código 36: Exemplos de expressões lógicas utilizando combinações dos conectivos `and` e `or`.

```

1 True
2 True
3 False
4 False

```

Código 37: Resultado na tela do Código 36.

Quando fazemos essa composição de operadores, uma boa prática é separar as operações por (), dessa forma, será construída uma ordem de prioridade. Por exemplo, na linha 5 do exemplo acima, primeiro é analisado a sub-expressão `((nome == 'livro') or (x + y < 100))`, a qual retorna `True`, visto que suas sub-expressões retornam `False` e `True`, respectivamente. Desse modo, a expressão torna-se: `(True) and (y >= 4)`. Essa expressão retorna `True` já que ambas as sub-expressões retornam `True`.

## 7 Comandos de seleção

Na programação, muitas vezes fazemos perguntas sim ou não e decidimos fazer algo com base na resposta. Por exemplo, podemos perguntar: “Você tem mais de 20 anos?” E, se a resposta for sim, responder “Você é muito velho!”

Esses tipos de perguntas são chamadas de condições e combinamos essas condições e as respostas em declarações do tipo `if`. As condições podem ser mais complicadas do que uma única pergunta, e expressões `if` podem ser combinadas com várias perguntas e respostas diferentes com base na resposta a cada pergunta. Nesta seção, você aprenderá como usar o comando `if` para criar programas com expressões condicionais.

### 7.1 Formatação de expressões condicionais

Expressões condicionais podem ser escrita em *Python*, conforme o Código 38.

```
1 idade = 15
2 if idade > 20:
3     print('Voce eh mais velho que o meu irmao!')
```

Código 38: Exemplo de expressão condicional em *Python*.

Uma expressão condicional é composta da palavra-chave `if`, seguida por uma condição e dois pontos (:), como em `if idade > 20:`. As linhas que seguem o sinal de dois pontos devem estar em um mesmo bloco, e se a resposta à pergunta for sim (ou `True`, como dizemos na programação em *Python*), os comandos no bloco serão executados. Agora vamos explorar como escrever blocos e condições.

Um bloco de código é um conjunto agrupado de instruções de programação. Por exemplo, quando `if idade > 20:` é verdadeiro, você pode querer fazer mais do que apenas imprimir “Você eh mais velho que o meu irmão!”. Talvez você queira imprimir algumas outras frases, como apresentado no Código 7.1.

```
1 idade = 25
2 if idade > 20:
3     print('Voce eh mais velho que o meu irmao!')
4     print('Faz tempo que voce mora aqui?')
5     print('Voce estudou em qual escola?')
```

Código 39: Exemplo de bloco de código usando `if`.

Esse bloco de código é composto de três instruções de impressão (`print`) que são executadas somente se a condição de `if idade > 20:` for considerada verdadeira. É necessário lembrar de indentar o código (vide 5.1), note que cada linha no bloco tem quatro espaços no começo, quando você compara com a instrução `if` acima dela.

Nós agrupamos as declarações em blocos porque elas estão relacionadas. As declarações precisam ser executadas juntas. Quando você altera o recuo, geralmente cria novos blocos. Então, cuidado com a declaração de blocos: um bloco com quatro espaços em uma linha e seis espaços na próxima produzirá um erro de recuo ao executá-lo, porque o *Python* espera que você use o mesmo número de espaços para todas as linhas em um bloco. Então, se você iniciar um bloco com quatro espaços, você deve usar consistentemente quatro espaços para todas as instruções presentes nesse bloco.

### 7.2 O que é uma condição?

Uma condição é uma instrução de programação que compara as coisas e nos informa se os critérios definidos pela comparação são verdadeiros ou falsos (`True` ou `False`). Por exemplo, `if idade > 20:` é uma condição, e é outra maneira de dizer: “O valor da variável de `idade` é maior que 20?” Essa também é uma condição: `corCabelo == 'ruivo'`, que é outra maneira de dizer: “O valor da variável `corCabelo` é ruivo?”.

Para as comparações são usados os operadores ensinados na Seção 6. Por exemplo, se você tem 10 anos, a condição `idade == 10` retornaria verdadeiro, já se você tem qualquer outra idade, retornaria falso. Já se você tem 12 anos, a condição `idade == 10` retorna verdadeiro. Analise, agora, o Código 40.

```
1 idade = 10
2 if idade > 10:
3     print('Voce esta velho demais para as minhas piadas!')
```

Código 40: Exemplo de expressão condicional em *Python*.

O que acontece quando esse código rodar? **Nada!** Como a condição é falsa, o código não executa o bloco abaixo. No entanto se fizéssemos `idade = 20`, seria impresso na tela “Voce esta velho demais para as minhas piadas!”.



1. Faça um programa que leia da tela três números e diga qual é o maior deles.
2. Faça um programa que pergunte o preço de três produtos e informe qual produto você deve comprar, sabendo que a decisão é sempre pelo mais barato.

### 7.3 O uso de `if-else`

Assim como usamos o `if` para fazer algo quando a condição dada é verdadeira, também podemos usar expressões condicionais para fazer algo quando a condição é falsa. Por exemplo, podemos imprimir uma mensagem na tela se sua idade fosse 10, e outra se sua idade fosse diferente de 10. O truque aqui é usar expressões com `if` seguidas de expressões com `else`. Basicamente é um modo de dizer “`if` (se) alguma coisa é verdadeira, faça isso, `else` (do contrário), faça aquilo”. Observe a utilização de `if-else` no Código 41.

```
1 print('Voce quer ouvir uma piada?')
2 idade = 8
3 if idade >= 10:
4     print ('O porco caiu na lama.')
5 else:
6     print ('Shh, eh um segredo!')
```

Código 41: Exemplo de expressão condicional utilizando `if` e `else` em *Python*.

Desse modo, o Código 41 geraria a seguinte mensagem na tela: ‘Shh, é um segredo!’, uma vez que a condição `if idade >= 10:` é falsa para `idade = 8`.



1. Faça um programa que peça um valor e mostre na tela se o valor é positivo ou negativo.

### 7.4 O uso de `if-elif-else`

Podemos estender expressões condicionais ainda mais utilizando o comando `elif`, o qual é uma palavra-chave que indica a abreviatura de `else` e `if`. Por exemplo, podemos checar se a idade de alguém é 10, 11, 12 ou 13 anos e, assim, podemos fazer nosso programa executar algo diferente baseado na resposta, como exemplificado no Código 42. Essas expressões se diferenciam da utilização do comando `if` seguido do comando `else`, explicado na Subseção 7.3, por que podemos ter mais de um `elif` na mesma expressão.

```
1 idade = 12
2 if idade == 10:
3     print("Porque o pinheiro nao se perde na floresta?")
4     print("Porque ele tem uma pinha!")
5 elif idade == 11:
6     print("Sabe como eh a piada do pintinho caipira?")
7     print("Pir")
```

```

8 elif idade == 12:
9     print("Por que o lapis esta triste?")
10    print("Porque ele foi desapontado")
11 elif idade == 13:
12     print("Qual a formula da agua benta?")
13     print("H Deus O")
14 else:
15     print("Huh?")

```

Código 42: Exemplo de expressão condicional utilizando `elif` em *Python*.

Dessa maneira, no Código 42 é checados se a variável `idade` vale 10, 11, 12 ou 13 e é impresso algo dependendo da resposta, e caso a idade não seja nenhuma dessas é impresso “Huh?”.

Ademais, é importante lembrar que você pode combinar condições usando as palavras chave `and` (e) e `or` (ou), produzindo códigos menores e mais simples. Um exemplo é apresentado no Código 43.

```

1 if idade == 10 or idade == 11 or idade == 12 or idade == 13:
2     print('Olá')
3 else:
4     print('Tchau')

```

Código 43: Exemplo de expressão condicional combinando condições em *Python*.

No Código 43, caso qualquer uma das condições na primeira linha seja verdadeira (se `idade` for 10 ou 11 ou 12 ou 13) o primeiro bloco será executado. Se essas condições forem todas falsas, o bloco de baixo será executado.



1. Faça um Programa que leia um número e exiba o dia correspondente da semana. (1-Domingo, 2- Segunda, etc.), se digitar outro valor deve aparecer valor inválido.
2. Escreva um programa que faça 5 perguntas para uma pessoa sobre um crime. As perguntas são: “Telefonou para a vítima?”, “Esteve no local do crime?”, “Mora perto da vítima?”, “Devia para a vítima?”, “Já trabalhou com a vítima?”. O programa deve no final emitir uma classificação sobre a participação da pessoa no crime. Se a pessoa responder positivamente a 2 questões ela deve ser classificada como “Suspeita”, entre 3 e 4 como “Cúmplice” e 5 como “Assassino”. Caso contrário, ele será classificado como “Inocente”.
3. Faça um Programa que peça os 3 lados de um triângulo. O programa deverá informar se os valores podem ser um triângulo. Indique, caso os lados formem um triângulo, se o mesmo é: equilátero, isósceles ou escaleno. Dicas: três lados formam um triângulo quando a soma de quaisquer dois lados for maior que o terceiro; triângulo equilátero: três lados iguais; triângulo isósceles: quaisquer dois lados iguais; triângulo escaleno: três lados diferentes;
4. Escreva um Programa que peça um número correspondente a um determinado ano e em seguida informe se este ano é ou não bissexto. Dica: chama-se ano bissexto o ano ao qual é acrescentado um dia extra, ficando com 366 dias, um dia a mais do que os anos normais de 365 dias, ocorrendo a cada quatro anos, exceto anos múltiplos de 100 que não são múltiplos de 400.



## 8 Comando de repetição

Existe uma ferramenta usada para que não seja necessário fazer a mesma coisa repetidamente na maioria das linguagens de computação chamada **laço**, que repete um bloco automaticamente por quantas vezes você desejar.

### 8.1 for

Para escrever “Olá” 5 vezes podemos fazer conforme descrito no Código 44.

```

1 print('hello')
2 print('hello')
3 print('hello')
4 print('hello')
5 print('hello')
```

Código 44: Exemplo de código imprimindo “hello” 5 vezes.

No entanto, fazer isso é tedioso e dá muito trabalho. Ao invés disso podemos usar a ferramenta de laço **for** como descrito no Código 45.

```

1 for x in range(0, 5):
2     print('Olá')
```

Código 45: Exemplo de código imprimindo “hello” 5 vezes utilizando **for**.

O código está basicamente dizendo para o *Python*: comece a contar a partir do 0 e só pare quando chegar em 5, armazene o valor da contagem em **x** a cada mudança e imprima “Olá”. Isso resulta na mesma impressão que o primeiro exemplo, mas sem precisar repetir a instrução `print('Olá')`.

Você deve estar pensando: mas qual é a grande vantagem disso? Imagine que, por alguma razão, você precisasse imprimir ““Olá” mil vezes na tela. Você ainda ia querer repetir `print('Olá')` mil vezes? Esse é o grande poder dos comandos de repetição como o **for**: repetir instruções por quantas vezes for necessário.

**CODElike a girl**

1. Faça um programa que imprima na tela os números de 1 a 20, um abaixo do outro.
2. Faça um programa que imprima na tela os números múltiplos de 5 presentes entre 1 a 20, um abaixo do outro.

É importante ressaltar que não é necessário o uso da função `range` em um `for`, você pode usar algo que você já criou, como no Código 46.

```
1 lista = ['maca', 'banana', 'arroz', 'feijao']
2 for i in lista:
3     print(i)
```

Código 46: Exemplo de código imprimindo os elementos de uma lista utilizando `for`.

O código 46 é um jeito de dizer “para cada item na lista, armazene-o na variável `i` e imprima o seu valor”. O resultado do Código 46 está representado no Código 47.

```
1 maca
2 banana
3 arroz
4 feijao
```

Código 47: Resultado na tela do Código 46.

Lembre-se de utilizar indentação para denotar o bloco de instruções dentro do `for` a ser repetido, ou então você acabará com uma mensagem de erro. Por exemplo, o Código 48 gera erro de execução.

```
1 lista = ['batata', 'carne', 'macarrao']
2 for i in lista:
3     print(i)
4     print(i)
```

Código 48: Exemplo de código utilizando `for` com erro de indentação.

Observe que no Código 48, o bloco de instruções do `for`, os dois `print` possuem um número de espaços diferentes, o que resulta em um erro.

Um exemplo mais complicado dessa situação de indentação está representado no Código 49.

```
1 lista = ['caneta', 'lapis', 'borracha']
2 for i in lista:
3     print(i)
4     for j in lista:
5         print(j)
```

Código 49: Exemplo de código com `fors` encaixados.

A saída do Código 49 está representada no Código 50.

```
1 caneta
2 caneta
3 lapis
4 borracha
5 lapis
6 caneta
7 lapis
8 borracha
9 borracha
10 caneta
11 lapis
12 borracha
```

Código 50: Resultado na tela do Código 49.

Basicamente, o que acontece no Código 49 é que o *Python* entra no primeiro laço e imprime o primeiro elemento, depois entra no segundo laço e imprime todos os elementos. Após isso, volta pro primeiro laço e imprime o segundo elemento e assim por diante. De fato, o comportamento de um `for` dentro do outro é conhecido como **laço encaixado** e pode ser algo desejado. Todavia, tome cuidado para não obter esse comportamento de forma indesejada.

## 8.2 while

Um laço formado por `for` não é o único laço que pode ser usado em Python. Um laço com `for` é executado uma quantidade específica de vezes, enquanto um laço com `while` é usado quando não se sabe a quantidade de repetições necessárias, mas sabe-se que ela deverá ser executada enquanto uma condição for satisfeita. Um exemplo de utilização do `while` está apresentado no 51. exemplo:

```

1 nome = input('Digite seu nome: ')
2 while nome != 'Maria':
3     print('Voce nao eh a Maria!')
4     nome = input('Digite seu nome: ')

```

Código 51: Exemplo de código com `while`.

Um laço em `while` é feito nos passos:

1. Checar a condição presente logo após o `while` (se variável `nome` não está armazenando o nome `Maria`).
2. Se verdadeira, executar o código no bloco (imprimir na tela 'Voce nao eh a Maria!' e requisitar ao usuário que digite seu nome).
3. Voltar ao Passo 1.

Ao utilizar `while` é **muito importante** lembrar de mudar a variável `nome` dentro do bloco de instruções executados pelo `while` para que o laço não seja executado para sempre, gerando o erro conhecido como **loop infinito**.

Outro exemplo de utilização de `while` está descrito no Código 52.

```

1 x = 45
2 y = 80
3 while x < 50 and y < 100:
4     x = x + 1
5     y = y + 1
6     print(x, y)

```

Código 52: Outro exemplo de código com `while`.

O laço do Código 52 será executado enquanto `x` for menor que 50 e `y` for menos que 100. Como `x` torna-se igual a 50 antes do `y` chegar a 100, nesse momento o laço é interrompido.

1. Numa eleição existem três candidatos. Faça um programa que leia na tela o número total de eleitores. Então, peça para cada eleitor votar e ao final, mostre o número de votos de cada candidato.
2. O Departamento Estadual de Meteorologia lhe contratou para desenvolver um programa que leia um conjunto indeterminado de temperaturas (conjunto indeterminado significa que é preciso perguntar na tela qual é a quantidade de temperaturas a serem lidas, antes de propriamente lê-las), e informa ao final a menor e a maior temperaturas registradas, bem como a média das temperaturas.

### 8.3 break

A palavra-chave `break` pode ser utilizada tanto no `while` quanto no `for` para que o laço seja interrompido. Basicamente, o programa lê uma condição e caso essa seja verdade e dentro desta tenha um `break`, o loop é quebrado e se passa para a próxima parte do código. Um exemplo de utilização do `break` está representado no Código 53.

```

1 i = 10
2 while i > 2
3     print(i)
4     if i == 6:
5         break
6     i = i - 1

```

Código 53: Exemplo de código com `break`.

O resultado na tela da execução do Código 53 está representado no Código 54.

```

1 10
2 9
3 8

```

Código 54: Resultado na tela do Código 53.

Note que no Código 54, apesar da condição dentro do `while` ser executar o código enquanto `i > 2`, existe a condição `i == 6` que quando satisfeita executa o `break`, que para a execução do laço e impede que a variável `i` chegue em 2.



1. Faça um programa que leia um nome de usuário e a sua senha e não aceite a senha igual ao nome do usuário, mostrando uma mensagem de erro e voltando a pedir as informações.
2. Os números primos possuem várias aplicações dentro da Computação, por exemplo na Criptografia. Um número primo é aquele que é divisível apenas por um e por ele mesmo. Faça um programa que peça um número inteiro e determine se ele é ou não um número primo.
3. Faça um programa que peça um numero inteiro positivo e em seguida mostre este número invertido.



## 9 Funções

No cotidiano, você já deve ter ouvido ou utilizado bastante o termo **função**. Normalmente, exercemos alguma função quando queremos atingir algum objetivo. Na programação em *Python* ocorre o mesmo: utilizamos funções para escrevermos um conjunto de comandos a fim de obtermos algum resultado.

No universo da programação, funções, também chamadas de **sub-rotinas**, **sub-programas** ou **métodos**, são trechos de um algoritmo (sequência de instruções) que encerram em si um pedaço da solução de um problema maior. Para atingirem determinados objetivos, tais funções podem ou não necessitar de **entradas** (também chamadas de parâmetros). Os **parâmetros** especificam qual informação você deve providenciar para que uma função possa ser utilizada. Ademais, as funções podem ou não retornar valores como resultados.

Por exemplo, imagine que você deseja criar uma função denominada `elevarQuadrado` cujo objetivo é elevar um número ao quadrado. O parâmetro desta função seria um número, que poderíamos chamar de `x`. Então, bastaria, dentro da função, elevar `x` a potência 2 e retornar esse resultado. O Código 55 é uma possível declaração em *Python* da função descrita.

```

1 def elevarQuadrado(x):
2     y = x ** 2
3     return y

```

Código 55: Declaração da função `elevarQuadrado`.

Após declarar uma função, para utilizá-la, basta fazer uma **chamada** em seu código. Por exemplo, uma chamada para a função presente no Código 55 está descrita no Código 56.

```

1 potencia = elevarQuadrado(3)
2 print(potencia)

```

Código 56: Chamada da função `elevarQuadrado`.

Note que, no Código 56 estamos armazenando o resultado obtido a partir da função `elevarQuadrado`, chamada com parâmetro 3, na variável `potencia`. O resultado na tela do Código 56 consiste na impressão do valor 9 (3 elevado ao quadrado).

No diagrama de blocos da Figura 3, demonstra-se um exemplo de sequência de sub-rotinas, cada uma executando uma função específica após receberem diferentes parâmetros de entrada. Nesse caso, as funções seguintes recebem como entrada o valor retornado pela função anterior.

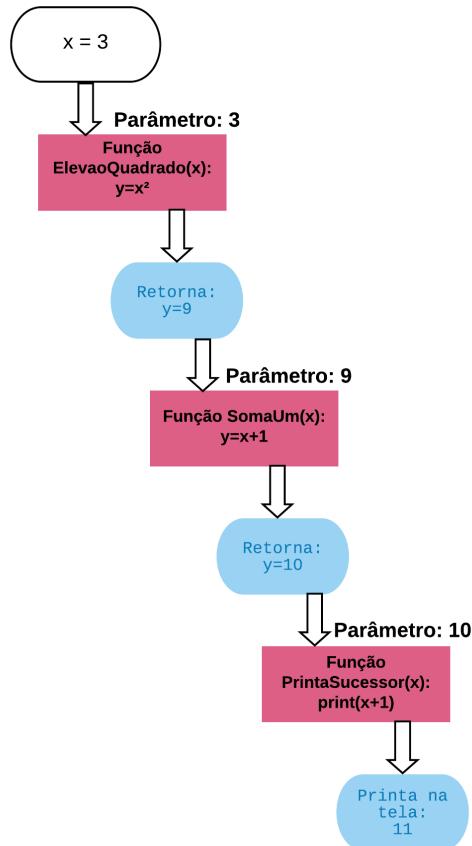


Figura 3: Exemplos de funções que atingem objetivos indicados nos blocos em azul.

## 9.1 Como criar funções em *Python*?

Para criarmos funções em *Python*, precisamos utilizar um formato, ou uma sintaxe específica. A sintaxe de uma função escrita nessa linguagem é composta de três partes: **nome**, **parâmetros** e **corpo**. Observe a função exemplificada no Código 57.

```

1 def funcãoExemplo (x):
2     if x>10:
3         y = x/10
4     else:

```

```
5     y = x
6     return y
```

Código 57: Declaração de função com retorno

O primeiro passo necessário para definir uma função está presente na linha 1. Note a presença do termo `def`. Ele é um comando em *Python* indispensável para definir uma função. Ao lado dele, coloca-se o nome da função que, nesse caso, chama-se `funcaoExemplo`. Em seguida, dentro dos parêntesis, coloca-se o(s) parâmetro(s) de entrada da função, que, neste caso, é um número qualquer, indicado por `x`. Caso a função precisasse receber mais de um parâmetro, bastaria colocá-los dentro dos parêntesis separados por vírgulas.

O comando `return` é necessário quando queremos retornar algum resultado (neste caso, o número `y`). Veja que, nesta função, o `y` retornado é igual a `x` quando `x` é menor ou igual a 10. Caso contrário, retorna-se um `y` igual a `x` dividido por 10. A sequência de comandos que efetuam esse algoritmo, isto é, o corpo da função, estão presentes nas linhas 2 a 5.

Nem todas as funções possíveis de serem criadas em *Python*, contudo, precisam receber parâmetros, nem retornar valores por meio do comando `return`. Um exemplo disso é a função `codeLikeAGirl()` descrita no Código 58. O corpo dela consiste apenas no comando que imprime uma frase na tela, sem a necessidade de receber parâmetros ou retornar algo.

```
1 def codeLikeAGirl():
2     print("Let's code!")
```

Código 58: Declaração de função sem retorno



1. Faça um programa, com uma função que necessite de três argumentos, e que forneça a soma desses três argumentos.
2. Escreva uma função que recebe os catetos de um triângulo retângulo e retorna o comprimento da sua hipotenusa.
3. Faça uma função que informe a quantidade de dígitos de um determinado número inteiro informado.
4. Faça uma função que informe a quantidade de dígitos de um determinado número inteiro informado.
5. Faça uma função que retorne o reverso de um número inteiro informado. Por exemplo: 127 -> 721.

## 9.2 Funções prontas em *Python*

Nem todas as funções precisam ser criadas por nós, pois algumas delas já encontram-se disponíveis para uso de forma automática. Elas são denominadas de funções **Built-in** e estão incorporadas na linguagem. Um exemplo de função desse tipo é a função `abs`, que retorna o valor absoluto (ou módulo) de um número. Por exemplo, o Código 9.2 utiliza a função `abs` para imprimir na tela o módulo do número digitado pelo usuário na tela.

```
1 x = int(input('Digite um numero: '))
2 mod = abs(x)
3 print('O modulo deste numero eh {}'.format(mod))
```

Além das funções **Built-in**, existe um grande universo de funções externas ao *Python* que podem ser encontradas na internet. Tais funções pertencem a **bibliotecas**, cujos acervos contemplam arquivos `.py`, denominados **módulos**. Geralmente, esses módulos são organizados em **pacotes**. Vamos explicar mais detalhadamente o que esses conceitos significam na Subseção a seguir.

### 9.3 Funções de bibliotecas externas

Basicamente, uma **biblioteca** é um conjunto de módulos, os quais podem conter funções, classes ou métodos, cujos significados serão explanados mais adiante, na Seção 12. Geralmente, em projetos de computação que lidam com grandes quantidades de arquivos, organizam-se os códigos em módulos específicos, de acordo com suas funcionalidades. Nesse contexto, módulos são arquivos .py que contêm em sua estrutura definições de funções e comandos quaisquer.

Por exemplo, em um projeto de um jogo, pode-se desenvolver alguns módulos para desenhar elementos na tela e outros para lidar com a transmissão de som, como ilustramos na Figura 4. Nesse exemplo, os subpacotes abrigam diversos módulos “.py”.

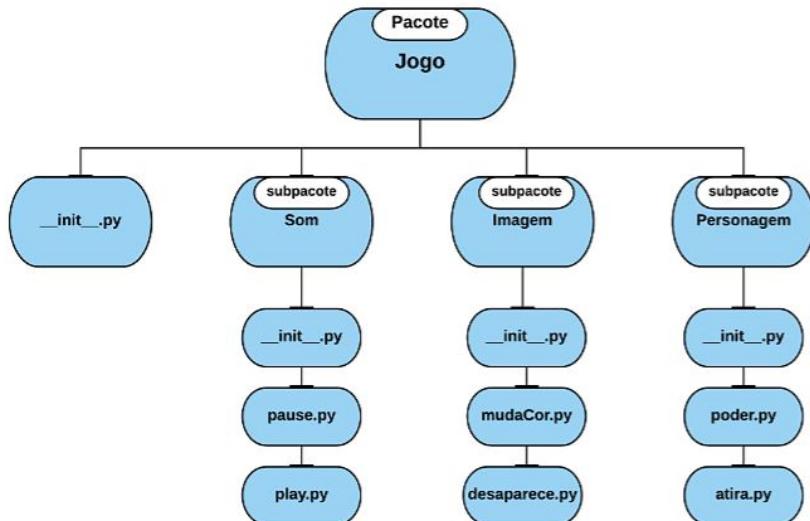


Figura 4: Exemplo de organização de módulos e funções em um jogo.

É comum organizar esses módulos em pacotes, que são diretórios (ou pastas) utilizados para abrigar, também, pacotes menores. Obrigatoriamente, pacotes devem conter um arquivo com o nome `__init__.py`, usado para indicar que ele se trata de um pacote e que pode ser "importado", assim como os módulos. A organização usual de classes e funções em módulos dentro de pacotes encontra-se ilustrada na Figura 5.

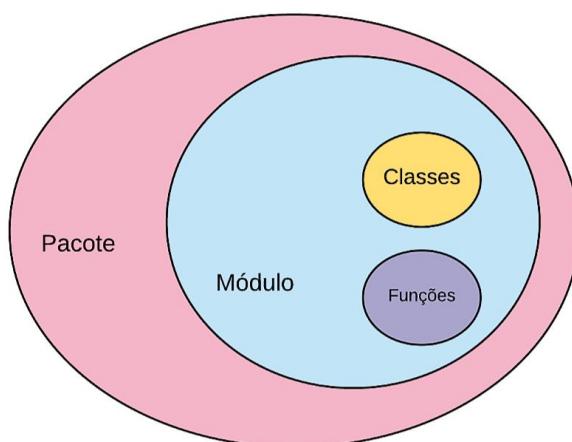


Figura 5

#### 9.3.1 O que significa importar um módulo, ou um pacote?

Quando uma função não pertence ao conjunto de funções **built-in** do *Python*, precisamos, após a instalação da biblioteca necessária, importar o módulo do qual ela pertence para depois "chamá-la"

propriamente. No código 59 exemplificamos o uso da função `sqrt`, pertencente ao módulo `math`, que está contido na biblioteca padrão do *Python*. A importação de funções é realizada pelo comando `import`, seguido do nome do módulo.

```
1 import math  
2 print(math.sqrt(25))
```

Código 59: Exemplo de utilização da função `sqrt` do módulo `math`

Uma forma mais prática de se importar funções de módulos em *Python* é demonstrada no código 60. Nesse caso, o comando `import *` permite a importação de todas as funções pertencentes a ao módulo `math`. Para utilizá-las após esse tipo de importação, torna-se desnecessário escrever o nome do módulo na chamada da função, como em `math.sqrt`.

```
1 from math import *  
2 print(sqrt(25))
```

Código 60: Exemplo de utilização da função `sqrt` do módulo `math`

Caso queiramos importar um módulo de um pacote, basta utilizarmos a estrutura `NomedoPacote.NomedoModulo`. Por exemplo, suponha que queremos importar o módulo `poder.py` do subpacote `Personagem`, pertencente ao pacote `Jogo`, da Figura 4. Nesse caso, utilizariammos a sequência de comandos demonstrada no código 61 e, se precisássemos usar uma função (por exemplo, `invisivel`) desse módulo, poderíammos usar o código 62.

```
1 import Jogo.Personagem.poder  
2 Jogo.Personagem.poder.invisivel()
```

Código 61: Importando um módulo do pacote `Jogo`

```
1 from Jogo.Personagem.poder import invisivel  
2 invisivel()
```

Código 62: Uso de função pertencente ao módulo `poder.py`

## 9.4 Vantagens de se utilizar funções no desenvolvimento de código

Você pode estar se perguntando: para quê devemos criar funções, se podemos escrever livremente comandos ao longo do código para atingir os mesmos objetivos? Já vamos te mostrar! Suponha que você queira imprimir na tela o resultado da `funcaoExemplo` (Código 57) para 5 valores diferentes de parâmetro: 8, 9, 10, 20, 21 e 22. Sem utilizar uma função, provavelmente o código que você escreveria corresponderia ao Código 63.

```
1 i = 8  
2 while (i <= 10):  
3     if i>10:  
4         y = i/10  
5     else:  
6         y = i  
7         print(y)  
8     i=i+1  
9  
10 i = 20  
11 while (i <= 22):  
12     if i>10:  
13         y = i/10  
14     else:  
15         y = i  
16         print(y)  
17     i=i+1
```

Código 63: Obtenção do resultado da `funcaoExemplo` para 5 valores distintos de entrada sem utilizar funções.

Note que o Código 63 é longo e repetitivo. Dessa maneira, se utilizarmos uma chamada da função `funcaoExemplo` dentro dos laços de repetição, poderemos fazer o mesmo trabalho com muito menos esforço, conforme explicitado no Código 64.

```
1 i = 8
2 while (i <= 10):
3     print(funcaoExemplo (i))
4     i=i+1
5 i = 20
6 while (i <= 22):
7     print(funcaoExemplo (i))
8     i=i+1
```

Código 64: Obtenção do resultado da `funcaoExemplo` para 5 valores distintos de entrada utilizando chamadas de funções.

Perceba que o uso da chamada de função nesse caso nos poupou linhas de código.

## 10 Vetores

Até então, nosso modo de armazenamento de informações se deu pelas variáveis, as quais armazenam um valor de determinado tipo. Agora, introduziremos o conceito de **vetor**.

Um vetor, também chamado de **lista**, é uma variável que nos permite guardar diversos valores de diversos tipos diferentes, como uma sequência de dados ocupando posições consecutivas de memória e, por isso, segue uma ordem natural: o primeiro elemento, o segundo e assim por diante. As posições consecutivas numeradas são denominadas *índices*. Os índices do vetor vão de 0, primeira posição, à `len(vet) - 1`, última posição.

A grande vantagem de usar vetores é poder trabalhar com um grande número de variáveis utilizando um único nome.

### 10.1 Construção de um vetor

Um vetor pode ser construído explicitando todos seus elementos, conforme mostrado no Código 65.

```
1 vet = [12, 34, 23, 7]
2 notas = [8.3 ,7.5 , 10]
3 nome = ["fernanda", "luiza", "marcela"]
4 lista = [12, 'livro', 7.332, 'casa']
```

Código 65: Métodos de construção de um vetor fornecendo seus elementos.

Pode-se adicionar elementos ao vetor, utilizando a função `append()`. No exemplo apresentado no Código 66, vamos adicionar o número 10 ao vetor `vet`, definido no Código 65.

```
1 vet = [12, 34, 23, 7]
2 vet.append(10)
3 print(vet)
```

Código 66: Código para adicionar elementos a um vetor.

O resultado da execução do Código 66 é o seguinte vetor: [12, 34, 23, 7, 10]. Ademais, pode-se gerar um vetor de modo iterativo, agregando novos elementos a cada passo usando o comando `for` e a função `append()`, conforme descrito no Código 67.

```
1 vet = [] # define um vetor vazio
2 for i in range(5) # a funcao range(n) gera numeros de 0 a n-1
3     vet.append(i)
4 print(vet)
```

Código 67: Código de um método de construção de um vetor de modo iterativo.

O resultado da execução do Código 67 é o seguinte vetor: [0, 1, 2, 3, 4]. Uma informação relevante sobre um vetor é o seu tamanho, isto é, o número de elementos que ele possui. O tamanho do vetor pode ser obtido por meio da utilização da função `len()`, conforme descrito no Código 68.

```
1  vet = [0, 1, 2, 3, 4]
2  print(len(vet))
```

Código 68: Utilização da função `len()` para obter o tamanho de um vetor.

O resultado na tela da execução do Código 68 é 5, pois o vetor possui 5 elementos. Além disso, é relevante dizer que nestes exemplos denominamos o vetor de `vet`, mas você pode denominar o vetor da forma que quiser, assim como fazia com variáveis fixas.

## 10.2 Manipulação de um vetor

Como citado anteriormente, os vetores possuem posições enumeradas denominadas índices, o que nos permite acessar os valores dentro do vetor, podendo alterá-los ou utilizar o valor correspondente àquela posição para outra operação. Desta forma, o elemento de índice `i` do vetor `lista` é dado por `lista[i]`. Observe o Código 69.

```
1  lista = [12, 'livro', 7.332, 'casa']
2  print(lista[0])
3  print(lista[3])
```

Código 69: Acesso ao elemento de um vetor por meio de seu índice.

Na linha 2 do Código 69, solicitamos ao programa que imprimisse o primeiro elemento do vetor (índice 0), na linha 3 solicitamos ao programa que imprimisse o quarto elemento do vetor (índice 3). O resultado obtido pela execução do Código 69 está apresentado no Código 70.

```
1  12
2  casa
```

Código 70: Resultado na tela do Código 69.

Para extrair um subconjunto de elementos de um vetor, utilizaremos, inicialmente, a mesma notação para a obtenção de um único elemento, porém, ao invés de definirmos somente o elemento que desejamos, informaremos agora, o intervalo desejado, ou seja, o índice do limitante inferior e o índice logo após o limitante superior do intervalo a ser extraído, conforme exemplificado no Código 71.

```
1  lista = [1, 2, 3, 4, 5, 6, 7, 8]
2  print(lista[1:4])
```

Código 71: Seleção de determinado intervalo de um vetor.

Tome cuidado com o acesso de intervalos, pois o primeiro índice informado antes dos dois pontos indica o ponto de início do intervalo mas o segundo índice corresponde a um índice após o fim do intervalo (o elemento com esse índice não fará parte do intervalo). Dessa maneira, a execução do Código 71 gera na tela: [2, 3, 4], isto é, a lista contendo apenas os elementos cujos índices variam entre 1 e 3.

Também podemos redefinir um determinado elemento do vetor, conforme descrito no Código 72.

```
1  lista = [12, 'livro', 7.332, 'casa']
2  lista[1] = 3
3  print(lista)
```

Código 72: Redefinindo um elemento de um vetor utilizando seu índice.

O Código 72 modifica o segundo elemento do vetor: '`livro`' para 3. O resultado na tela da sua execução gera o vetor: [12, 3, 7.332, '`casa`']. Além disso, podemos realizar operações matemáticas com elementos do vetor, conforme explicitado no Código 73.

```
1  lista = [12, 3, 7.332, 'casa']
2  print(2.5 + lista[2])
```

Código 73: Acesso ao elemento de um vetor por meio de seu índice para realização de operação aritmética.

O resultado na tela da execução do Código 73 é 9.832. Podemos, ainda, retirar elementos de uma determinada posição ou adicioná-los em determinado índice por meio das funções `del()` e `insert()`, respectivamente. Ainda por cima, podemos excluir todos os elementos de um vetor com a função `clear()`, ou descobrir o tamanho de um vetor com o auxílio da função `len()`, conforme descrito no Código 74.

```

1 A = ["a", "b", "c"]
2 del(A[0])
3 print(A)
4 A.insert(0, "y")
5 print(A)
6 print(len(A))
7 A.clear()
8 print(A)

```

Código 74: Utilização das funções `del()`, `insert()` e `clear()`.

O resultado obtido pela execução do Código 74 está apresentado no Código 75.

```

1 ["b", "c"]
2 ["y", "b", "c"]
3 3
4 []

```

Código 75: Resultado na tela após execução do Código 74.

Note a diferença entre a função `insert()` e a função `append()`, vista anteriormente: a função `append()` adiciona um elemento no final do vetor, enquanto que a função `insert(i,elemento)` adiciona um `elemento` na posição de índice `i` do vetor.

A fim de apagar os elementos de um vetor em certo intervalo, utilizamos a função `del()`, indicando o índice do primeiro e do último elemento, conforme exemplificado no Código 76, cujo resultado na tela é `[1, 6, 7, 8]`, isto é, o vetor A sem os elementos com índices de 2 a 4.

```

1 A = [1, 2, 3, 4, 5, 6, 7, 8]
2 del(A[1:4])
3 print(A)

```

Código 76: Remoção de determinado intervalo de um vetor.

Na hora de manipular elementos dos vetores, contudo, **tome bastante cuidado** para não acessar posições não existentes no vetor como, por exemplo, posições negativas ou acima de `len(vet)`.



1. Faça um algoritmo que solicite ao usuário números e os armazene em um vetor de 20 posições. Crie uma função que recebe o vetor preenchido e substitua todas as ocorrências de valores negativos por zero, as ocorrências de valores menores do que 10 por 1 e as demais ocorrências por 2.
2. Faça uma função que receba duas listas e retorne True se são iguais ou False, caso contrário. Duas listas são iguais se possuem os mesmos valores e na mesma ordem.
3. Faça um programa que percorre uma lista e exiba na tela o valor mais próximo da média dos valores da lista. Exemplo: Dada a lista = [2.5, 7.5, 10.0, 4.0], a média dos elementos é 6.0 e o valor mais próximo da média é 7.5.
4. Faça um programa que simule um lançamento de dados. Lance o dado 100 vezes e armazene os resultados em um vetor . Depois, mostre quantas vezes cada valor foi conseguido. Dica: use um vetor de contadores(1-6) e uma função para gerar números aleatórios, simulando os lançamentos dos dados.

## 10.3 Operações com vetores

### 10.3.1 Soma

O operador `+` é utilizado para concatenar duas ou mais listas resultando em uma só. Concatenar significa juntar os dois operandos unindo o fim de um com o começo do outro, conforme explicitado no Código 77.

```

1 A = [1, 2, 3]
2 B = [4, 5, 6]
3 C = A + B
4 print(C)
5 C = A + [7]
6 print(C)

```

Código 77: Exemplos de soma de vetores.

O resultado obtido pela execução do Código 77 está apresentado no Código 78.

```

1 [1, 2, 3, 4, 5, 6]
2 [1, 2, 3, 7]

```

Código 78: Resultado na tela após execução do Código 77.



1. Faça uma função que receba duas listas e exiba a união destas listas.
2. Faça uma função que receba duas listas e exiba a interseção destas listas.
3. Faça uma função que receba duas listas e exiba a intercalação destas listas, isto é, 1º da 1ª lista, 1º da 2ª lista, 2º da 1ª lista, 2º da 2ª lista, etc.
4. Faça uma função que receba uma lista e exiba os elementos da última metade na frente dos elementos da primeira metade.

### 10.3.2 Multiplicação

O operador \* repete  $n$  vezes os elementos que já estão na lista. Fazer  $lista * n$  equivale a fazer  $lista + lista + \dots + lista$  ( $n$  vezes). Um exemplo de utilização do operador \* pode ser visto no Código 79, cujo resultado na tela é  $[1, 2, 3, 1, 2, 3, 1, 2, 3]$ .

```

1 A = [1, 2, 3]
2 print(A*3)

```

Código 79: Exemplo de multiplicação de vetores.

Atenção, observe que ao utilizarmos os operadores + e \* nós não estamos realizando uma soma ou produto algébricos dos elementos do vetor, mas sim manipulando a lista, ou seja, ou unindo listas, ou repetindo a mesma lista para gerar uma nova lista.

## 11 Strings

Uma *string*, em essência, serve para trabalharmos com textos. Ela é uma sequência de caracteres (letras, números, símbolos) e para ser declarada em Python basta colocar todo seu conteúdo dentro de aspas simples ou duplas, como podemos ver no exemplo abaixo:

```

1 nome1 = 'Ana Beatriz'
2 nome2 = "Marcela"
3 idade1 = "14"
4 idade2 = '16'

```

Código 80: Exemplo de criação de strings.

Ao dar print das variáveis de idade1 e idade2 eles aparecerão como '14' e '16', respectivamente, pois apesar de seu conteúdo ser numérico, elas são do tipo string. Porém, podemos convertê-las em variáveis numéricas utilizando as funções de conversão int() ou float(), que as transformam em tipos inteiro e decimal, conforme descrito no Código 81.

```

1 idade1 = int(idade1)
2 print(idade1)
3 idade2 = float(idade2)
4 print(idade2)

```

Código 81: Exemplo de conversão de string em variáveis numéricas.

O resultado na tela do Código 81 está descrito no Código 82.

```
1 14
2 16.0
```

Código 82: Resultado na tela do Código 81.

### 11.1 Manipulação de *strings*

As *strings* funcionam como vetores, podendo ser manipuladas como estes. Para se obter um caractere específico da *string*, basta colocar a posição desejada entre colchetes após o seu nome, sendo que a contagem começa no 0. Esse processo é conhecido como **indexação**.

Já para se obter uma subsequência de elementos da *string*, devemos colocar após o nome da *string*,  $[n_1:n_2]$ , em que  $n_1$  é o índice do primeiro caractere e  $n_2$  é o índice do último caractere do intervalo que deseja-se selecionar. Esse processo é conhecido como **fatiamento**. Um exemplo de indexação e fatiamento de *strings* está descrito no Código 83.

```
1
2 nome1 = 'Ana Beatriz'
3 print(nome1[4])
4 print(nome1[2:7])
```

Código 83: Exemplo de fatiamento de *strings*.

O resultado na tela do Código 83 está descrito no Código 84.

```
1 'B'
2 'a Bea'
```

Código 84: Resultado na tela do Código 83.

Note que um espaço conta como um caractere.

**CODE**like a girl

1. Faça um programa que solicite o nome do usuário e imprima-o na vertical em formato de escada, conforme a Figura 6.
2. Um palíndromo é uma sequência de caracteres cuja leitura é idêntica se feita da direita para esquerda ou vice-versa. Por exemplo: OSSO e OVO são palíndromos. Faça um programa que leia uma palavra e diga se é um palíndromo ou não.
3. Desenvolva um programa que solicite a digitação de um número de CPF no formato xxx.xxx.xxx-xx e indique se é um número de CPF válido ou inválido através da validação dos dígitos verificadores e dos caracteres de formatação.
4. DESAFIO: Escreva um programa que solicite ao usuário a digitação de um número até 99 e imprima-o na tela por extenso.

```
F
FU
FUL
FULA
FULAN
FULANO
```

Figura 6: Formatação de impressão em escada para o Exercício 2.

## 11.2 Concatenação e multiplicação de *strings*

Para concatenar e multiplicar *strings* utilizamos os operadores `+` e `*`, da mesma forma que fazíamos com os vetores. Desta maneira, ao executarmos o Código 85, obtemos na tela o resultado descrito pelo Código 86.

```
1 nome_3 = 'Code'
2 nome_4 = ' like a girl'
3 print(nome_1 + nome_2)
4 excl = '!，“
5 print(nome_3 + nome_4 + excl*3)
```

Código 85: Exemplo do uso dos operadores + e \* com strings.

```
1 'Code like a girl'  
2 'Code like a girl!!!'
```

Código 86: Resultado na tela do Código 85.

### 11.3 Funções próprias para manipulação de *strings*

Além das operações acima, existem várias funções em *Python* que facilitam a manipulação das strings. Nas subseções a seguir são mostrados algumas delas.

### 11.3.1 Função len()

Assim como em vetores, a função `len(nomeDaString)` retorna o tamanho de uma *string*, incluindo os espaços. Um exemplo de utilização da função `len()` está descrito no Código 87.

```
1 teste = 'Gosto muito de programar.'
2 tamano = len(teste)
3 print(tamano)
```

Código 87: Exemplo de uso da função len.

O resultado na tela da execução do Código 87 é 25, pois este é o número de caracteres da string `Gosto muito de programar..`

### 11.3.2 Função replace()

Essa função retorna outra `string` igual a original, mas com um trecho dela substituído por outro indicado. Basicamente, para utilizá-la, você deverá utilizar a seguinte sintaxe: `minhaString.replace(<trecho a ser substituído>, <substituição>)`. Um exemplo de utilização da função `replace()` está explicitado no Código 88.

```
1 teste = 'Gosto muito de programar.'
2 teste2 = teste.replace('programar', 'aprender')
3 print(teste2)
```

Código 88: Exemplo do uso do método `replace`.

O resultado na tela da execução do Código 88 será a *string* ‘Gosto muito de aprender.’, pois a função `replace` substituirá a *substring* ‘programar’ pela *substring* ‘aprender’, mantendo o resto da *string* inalterada.

**CODE** like a girl 

1. Faça um programa que leia um número de telefone e corrija o número no caso deste conter somente 7 dígitos, acrescentando o ‘3’ na frente. O usuário pode informar o número com ou sem o traço separador.
  - 2.
  3. Você e sua amiga criaram uma criptografia em que substitui as vogais (a, e, i, o, u) por (/, \*, -, +, =), respectivamente. Crie uma função que receba um texto nessa criptografia e passe-o para a forma usual, em seguida, usando a função criada, decifre a mensagem que sua amiga lhe enviou:  $j = nt/ss + m + sm/ - sf + rt * s$ .

### 11.3.3 Função count

Essa função conta o número vezes que um caractere ou conjunto deles aparece na *string*. Basicamente, para utilizá-la, você deverá utilizar a seguinte sintaxe: `minhaString.count(<conjunto de caracteres>)`. Um exemplo de utilização da função `count()` está explicitado no Código 89.

```
1 teste = 'Gosto muito de programar.'
2 qtd0 = teste.count('o')
3 print(qtd0)
4 qtdMuito = teste.count('muito')
5 print(qtdMuito)
```

Código 89: Exemplo do uso do método `count..`

O resultado na tela da execução do Código 89 será 4 (pois existem 4 letras ‘o’ na *string* ‘Gosto muito de programar.’) e 1 (pois a *substring* ‘muito’ só aparece uma vez na *string* ‘Gosto muito de programar.’).

### 11.3.4 Função find

Essa função busca uma *substring* dentro de uma *string*. Se essa função encontrar a *substring* dentro da *string*, ela retorna o índice em que a ocorrência da *substring* inicia dentro da *string*. Caso contrário, a função retorna -1.

Para utilizar a função `find`, você deverá utilizar a seguinte sintaxe: `minhaString.find(<conjunto de caracteres>)`. Um exemplo de utilização da função `find()` está explicitado no Código 90.

```
1 teste = 'Gosto muito de programar.'
2 temM = teste.find('m')
3 print(temM)
4 temK = teste.find('k')
5 print(temK)
```

Código 90: Exemplo do uso do método `find..`

O resultado na tela da execução do Código 89 será 6 (pois a letra ‘m’ foi encontrada na *string* ‘Gosto muito de programar.’, estando localizada, pela primeira vez, no índice 6 da *string*) e -1 (pois a letra ‘k’ não foi encontrada na *string* ‘Gosto muito de programar.’).



1. Dado uma *string* com uma frase informada pelo usuário (incluindo espaços em branco), conte:
  - a) quantos espaços em branco existem na frase.
  - b) quantas vezes aparecem as vogais a, e, i, o, u.

### 11.3.5 Função split

Essa função retorna uma lista composta por *strings* que resultam da divisão da *string* original por um separador. Para usar o método, colocamos dentro do parênteses o caractere (ou conjunto deles) que será o separador. Se não for indicado, o separador será o espaço. Um exemplo de utilização da função `split()` está explicitado no Código 91.

```
1 teste = 'Gosto muito de aprender.'
2 divisao1 = teste.split()
3 print(divisao1)
4 divisao2 = teste.split('de')
5 print(divisao2)
```

Código 91: Exemplo do uso do método `split..`

O resultado na tela do Código 91 está descrito no Código 92.

```
1 ['Gosto', 'muito', 'de', 'aprender.']
2 ['Gosto muito ', ' aprender.']}
```

Código 92: Resultado na tela do Código 91.



1. Faça um programa que solicite a data de nascimento (dd/mm/aaaa) do usuário e imprima a data com o nome do mês por extenso. Por exemplo: A entrada ‘17/10/1995’ teria como saída 17 de Outubro de 1995.

#### 11.3.6 Funções upper e lower

As funções `upper` e `lower` modificam as letras da *string* para serem todas maiúsculas e minúsculas, respectivamente. Basicamente, para utilizá-las, você deverá utilizar a seguinte sintaxe: `minhaString.upper()` ou `minhaString.lower()`. Um exemplo de utilização dessas funções está explicitado no Código 93.

```
1 teste = 'Gosto muito de aprender.'
2 maiusculo = teste.upper()
3 minusculo = teste.lower()
```

Código 93: Exemplo do uso dos métodos `upper` e `lower`.

O resultado na tela do Código 93 está descrito no Código 94.

```
1 'GOSTO MUITO DE APRENDER.'
2 'gosto muito de aprender.'
```

Código 94: Resultado na tela do Código 93.



1. Faça um programa que permita ao usuário digitar o seu nome e em seguida mostre o nome do usuário de trás para frente utilizando somente letras maiúsculas. Dica: lembre-se que ao informar o nome o usuário pode digitar letras maiúsculas ou minúsculas.

## 12 Introdução a Programação Orientada a Objetos

A **Programação Orientada a Objetos** (POO) diz respeito a um padrão de desenvolvimento (paradigma) que é seguido por muitas linguagens, como C++ e Java. Diferentemente da Programação Estruturada (PE), muito usada com a linguagem C, na qual criamos funções (ou procedimentos) que podem ser aplicadas em todo nosso código, em POO declaramos funções que só podem ser aplicadas a blocos específicos, como ilustrado na Figura 7.

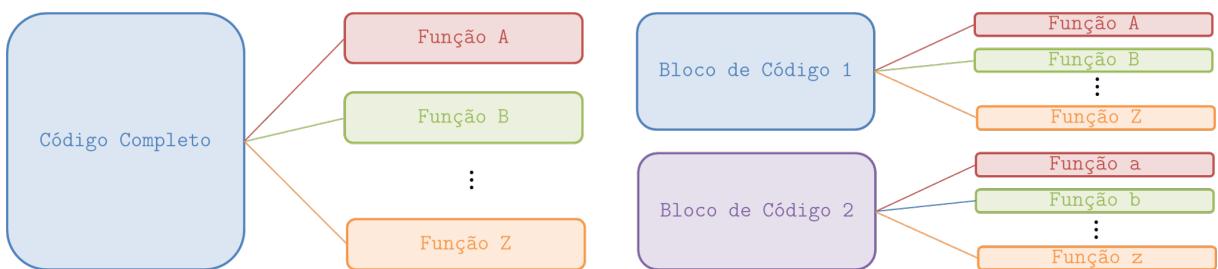


Figura 7: Comparação entre PE e POO.

Apesar de possuir um desempenho pior que o da programação estruturada, a programação orientada a objeto se difundiu muito por facilitar a reutilização do código, permitir uma maior organização e divisão desse e pela maior segurança.



## 12.1 Objetos

Para programarmos utilizando POO precisamos primeiramente abstrair a realidade para o nosso código dividindo-o em **objetos**, como o nome sugere. Para isso precisamos pensar como devemos nomear esses objetos dentro do nosso código (ex. Jogador, Telas, Inimigos), quais as características desses objetos que em POO chamaremos de **atributos** e quais ações esse objeto irá realizar ou receber, que são nomeadas **métodos**.

Para exemplificar, pensemos na protagonista do nosso jogo, podemos dizer que ela é um objeto e que entre seus atributos estão sua **imagem**, seu **tamanho**, sua **velocidade** e sua **posição**. Além disso ela possui alguns métodos para realizar algumas ações, como **pular** e **atirar**. Dessa maneira, o objeto criado para representar nossa personagem no código teria a estrutura mostrada na figura 8.

```
protagonista:
#Atributos
    protagonista.imagem = personagem.png
    protagonista.tamanho = (30,120)
    protagonista.velocidade = (15,0)
    protagonista.posicao = (0,500)
#Métodos
    protagonista.pular()
    protagonista.atirar()
```

Figura 8: Exemplo de um objeto.

Em *Python*, acessamos os atributos e métodos de um objeto pela sintaxe `nomeDoObjeto.atributo` e `nomeDoObjeto.metodo()`, como exemplificado no Código 95.

```
1 print(protagonista.tamanho) #imprime (30, 120)
2 protagonista.pular() # realiza as acoes descritas em pular
```

Código 95: Exemplo de acesso a atributos de objetos e da chamada de métodos dos objetos.

## 12.2 Classes

Os objetos podem possuir características e ações parecidas. Por exemplo, se pensarmos em personagens existem muitos possíveis objetos que são personagens, por exemplo as personagens Violeta, Esmeralda e Onix, entre outros. Para não escrevermos várias vezes os mesmos atributos e métodos podemos - e devemos - criar uma **Classe** que irá conter um modelo de como serão os objetos criados a partir dela. Por exemplo, a classe **Personagem** pode ter as características `nome`, `genero` e `idade` e os métodos `anda()`, `pula()` e `come()` e a partir dela podemos criar os objetos `violeta`, `esmeralda` e `onix` como representado no diagrama da Figura 9.

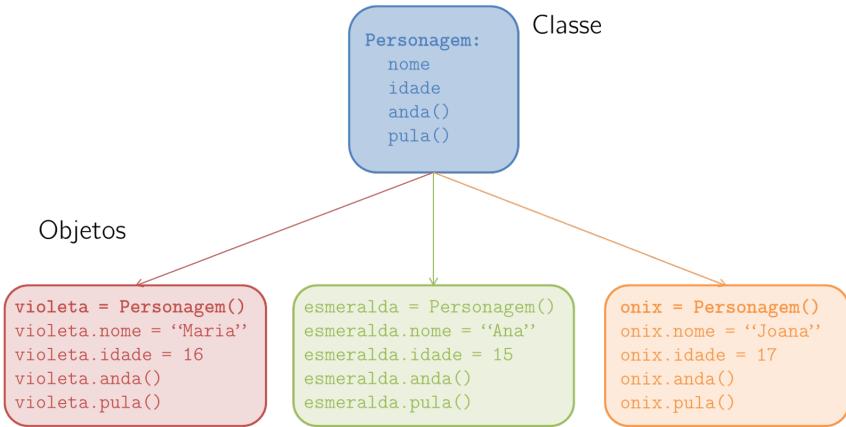


Figura 9: Exemplo de classe e objetos criados a partir dessa classe.

É importante ressaltar que todo objeto deve ser criado a partir de uma classe, mesmo que ele seja o único objeto gerado desta.

Em *Python* criamos uma nova classe utilizando a sintaxe descrita no Código 96.

```

1  class NomeDaClasse():
2      def __init__(self, parametrosDeInicializacao):
3          #atributos
4          self.atributo1 = 1
5          self.atributo2 = 2
6          [...]
7      #metodos
8      def primeiroMetodo(parametro):
9          [...]
10     def segundoMetodo(parametro):
11         [...]

```

Código 96: Sintaxe de uma classe em *Python*.

A palavra **self** presente no Código 96 é usada para se referir a própria classe. Por exemplo **self.atributo1** refere-se ao atributo **atributo1** da classe *Personagem*. Podemos usar **self** para invocar os próprios métodos da classe também.

Para declarar o objeto de uma classe devemos utilizar a sintaxe descrita no Código 97.

```
1  objeto = NomeDaClasse(parametro1, parametro2, ...)
```

Código 97: Criação do objeto de uma classe em *Python*.

No Código 97, os parâmetros passados para a criação do objeto correspondem aos parâmetros do método construtor da classe (responsável por inicializar os objetos da classe), o qual no Código 96 pode ser identificado como **\_\_init\_\_**

### 12.3 Herança

Como mencionado uma das principais vantagens de POO é a sua fácil reutilização, muito dessa característica deve-se a **herança**. Para exemplificar, pense em uma família, na qual uma criança herda as características da mãe que, por sequência, herdou as características da avó. Portanto, a criança também possuirá características da avó e assim sucessivamente, além de características novas próprias dela. De forma análoga ocorre na orientação a objetos, em que uma classe derivada de outra herda os métodos e atributos desta e isso estende-se a qualquer classe acima da classe pai.

Pensando no exemplo da classe *Personagem*, a partir dela podemos criar a classe filha *Protagonista* que possui o método **atira()**, assim como a classe *Figurante* com o método **fazNada()** e a classe *Inimigo* adicionando o método **defende()**. A partir da classe *Inimigo* podemos, ainda, criar uma classe derivada denominada *Boss*, que além dos métodos herdados de *Personagem*, possuirá o método **defende()** herdado de *Inimigo* e o novo método **ataca()**. Na Figura 10, exemplificamos como essa relação ocorre.

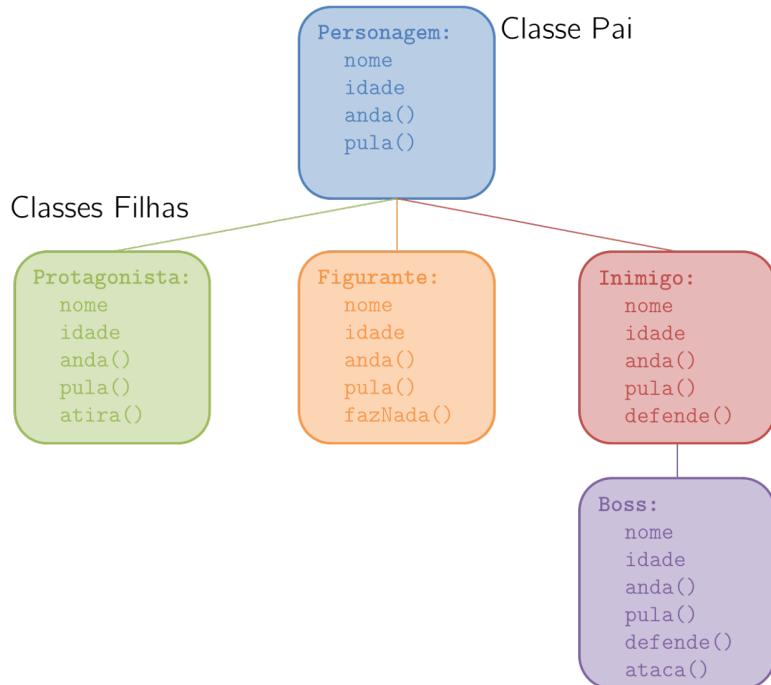


Figura 10: Exemplo de classes com relação de herança.

Para criar uma classe a partir de outra utilizaremos a sintaxe descrita no Código 98.

```

1 from NomeDaClassePai import *
2 class NomeDaClasseFilha(NomeDaClassePai):
3     def __init__(self, parametrosDeInicializacao):
4         #constroi a classe pai
5         super().__init__() # opcional, caso contrario deve-se repetir as partes
                           # de interesse do metodo __init__() da classe pai.
6 [...]
  
```

Código 98: Sintaxe utilizada para criar uma classe derivada (classe filha) de outra.

A palavra `super` é usada para se referir à classe ancestral (classe pai). O método `super().__init__()` é utilizado para invocar o construtor da classe pai, evitando que tenhamos que repetir o que foi feito naquela classe.



1. Crie uma classe para implementar uma conta corrente. A classe deve possuir os seguintes atributos: `numeroDaConta`, `nomeDoCorrentista` e `saldo`. Os métodos são os seguintes: `alterarNome`, `alterarNumero`, `deposito`, `saque`, `consultarSaldo`. No construtor, `saldo` possui valor padrão zero e os demais atributos estão vazios ([ ]).

2. Vamos criar um minijogo de charadas utilizando POO. Para isso:
  - a) implemente uma classe Palavra com um atributo palavra, que guardará o termo a ser adivinhado, e atributos pistas (pista1, pista2, ...) que guardarão pistas sobre essa palavra, bem como o atributo pontuacao que indica a pontuação do jogador.
  - b) implemente o método receberPistas que receba essas informações e limpe a tela.
  - c) implemente o método informarPistas que informe essas pistas conforme as escolhas do jogador, desconte 1 ponto da pontuação do jogador para cada cada pista informada.
  - d) implemente um arquivo main.py que permita que um jogador informe a palavra e as pistas e em seguida que um outro jogador escolha quais

Dica 1: Use a biblioteca `from os import system` no início do arquivo para importar a função `system('cls')` que te permite apagar o que está no prompt de comando.

Dica 2: Caso queira implementar a classe em um arquivo diferente use `from NomeDoArquivo import *` para obter acesso as classes desse arquivo no `main.py`.
3. Desafio: Use herança para criar modalidades específicas do jogo como adivinhar animais, objetos, filmes, entre outros.

## 13 Desenvolvimento do Jogo

### 13.1 Pygame

*Pygame* é uma biblioteca *open source* composta por diversos módulos projetados para se desenvolver videogames em *Python*. A maioria dos jogos separa a estrutura, o estilo e a lógica de programação em *HTML*, *CSS* e *JavaScript*, exigindo conhecimentos em todas essas linguagens. Embora essa separação seja melhor a longo prazo, é uma grande para iniciantes. Por causa disso, considera-se o *Pygame* uma boa alternativa para quem está entrando no universo de desenvolvimento de jogos.



Figura 11: Logotipo do *Pygame*.

O *Pygame* utiliza a biblioteca Simple DirectMedia Layer (SDL), uma biblioteca multimídia e multiplataforma escrita em C que possui interfaces para várias linguagens de programação, inclusive *Python*. A SDL gerencia vídeo, eventos, áudio digital, CD-ROM, som, processamento de objetos compartilhados, rede e tempo, tornando possível usar uma linguagem de programação de alto nível, como *Python*, para estruturar o jogo.

Apesar de ser uma biblioteca muito boa, a SDL não possui alguns elementos bem importantes no desenvolvimento de jogos, por causa disso, além do SDL, o *Pygame* reúne outras ferramentas como métodos de matemática vetorial, detecção de colisões, gerenciamento de gráficos de cenas 2D (como *sprites*), suporte para MIDI (Interface Digital para Instrumentos Musicais), manipulação de matrizes de pixels, filtragens, suporte para desenhos, entre outros.

Dadas todas as vantagens descritas, o *Pygame* será uma biblioteca muito utilizada no desenvolvimento do nosso jogo.

### 13.2 Telas

Para criarmos as telas do jogo, usamos a classe `Tela` (presente no arquivo `Tela.py`), que possui atributos e métodos comuns a todas elas. Esses métodos são responsáveis por definir os comportamentos dos botões de áudio e de sair e os comandos que permitem desenhar uma tela básica, isto é, uma tela contendo um plano de fundo e o botão de áudio. Um diagrama contendo os atributos e métodos desta classe está representado na Figura 12.

## Classe

Tela:  
  imagem de fundo  
  comportamentoBotaodeAudio()  
  comportamentoBotaoDeSair()  
  desenharTelaBasica()

Figura 12: Métodos e atributos da classe Tela.

A criação da classe **Tela** (arquivo **Tela.py**) é iniciada com a definição da função `__init__`, chamada de construtor. Observe o Código 99, no qual está descrito o construtor da classe **Tela**. Nele declaramos 3 atributos da classe (por isso eles apresentam o prefixo `self`): `imagemDeFundo` (para armazenar a imagem de fundo da tela), `imagemDeFundoX` e `imagemDeFundoX2` (variáveis auxiliares utilizadas para criar o efeito de paralaxe explicado mais detalhadamente na Subseção 13.2.6.3).

```
1 # declaracao de bibliotecas
2 from Configuracoes import *
3 import pygame
4 import os
5 class Tela:
6     def __init__(self):
7         # declaracao de atributos
8         self.imagemDeFundo = pygame.image.load(os.path.join('Imagens', 'cenario_1_extendido.png'))
9         self.imagemDeFundoX = 0
10        self.imagemDeFundoX2 = self.imagemDeFundo.get_width()
```

Código 99: Construtor da classe Tela.

Note que para carregarmos a imagem de fundo utilizamos a chamada da seguinte função: `pygame.image.load(<caminho do arquivo>)`. Essa função é própria do *Pygame* e é responsável por carregar uma nova imagem de um arquivo. Por causa disso, precisamos importar a biblioteca *Pygame* nesta classe, com a instrução `import pygame` no início do arquivo.

Além disso, para facilitar a escrita do caminho em que a imagem está salva, faremos uso da função `os.path.join(<nome da pasta>, <nome do arquivo>)`. Essa função é própria da biblioteca `os`, possui como argumentos o nome da pasta em que o arquivo está armazenado e o nome do arquivo em si, e retorna uma *string* com o caminho para se acessar o arquivo de imagem em questão. Por causa disso, precisamos importar a biblioteca `os` nesta classe, com a instrução `import os` no início do arquivo.

A partir de agora, toda vez que precisarmos carregar a imagem de algum elemento do jogo a ser desenhado na tela, utilizaremos a seguinte instrução: `pygame.image.load(os.path.join(<nome da pasta>, <nome do arquivo>))`.

Após inicializar os atributos da classe, precisamos declarar seus métodos responsáveis por desenhar a imagem de fundo e o botão de áudio na tela, bem como o método responsável por definir o comportamento dos botões de áudio e de sair.

Primeiramente, vamos declarar a função (ou método) responsável por desenhar a imagem de fundo e o botão de áudio na tela. Para alcançar esse objetivo, utilizaremos a seguinte função própria do *Pygame* `game.janela.blit(<imagem>, <posicao a ser inserida na tela>)`. Essa função é responsável por desenhar uma superfície (imagem) sobre outra superfície (janela do jogo). Ela recebe como argumentos a variável que armazena a imagem e a posição em que a imagem deve ser inserida na tela. Para saber como posicionar os objetos na tela corretamente, precisamos nos atentar às coordenadas que o *Pygame* utiliza para posicionar imagens na tela. Basicamente, as coordenadas da tela e das imagens iniciam sempre no canto superior direito, conforme ilustrado pelos eixos em vermelho da Figura 13, em que a posição (0, 0) corresponde ao encontro dos eixos vermelhos e os valores das coordenadas x e y crescem positivamente de acordo com as setas.

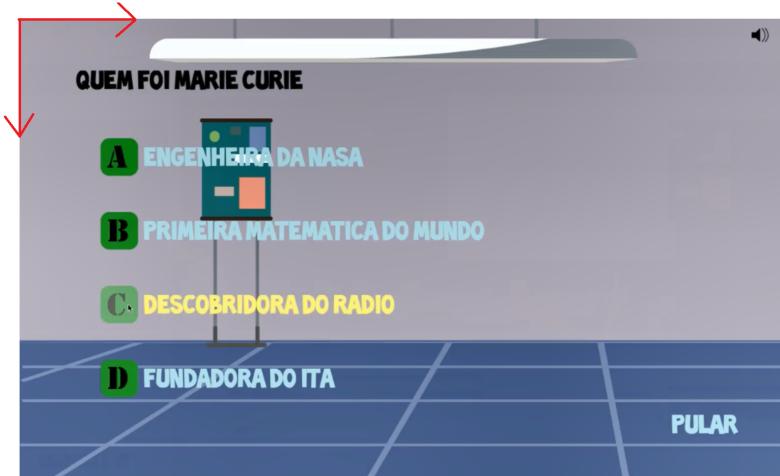


Figura 13: Coordenadas na Tela. A vertical representa o  $x$  e a horizontal representa o  $y$ .

O código completo que desenha a imagem de fundo e o botão de áudio nas telas do jogo está descrito no Código 100.

```

1 # desenha os elementos comuns a toda as telas, isto eh, a tela de fundo e o
2 # botao de audio
3 def desenharTelaBasica(self, game):
4     game.janela.blit(self.imagemDeFundo, (self.imagemDeFundoX, 0))
5     game.janela.blit(self.imagemDeFundo, (self.imagemDeFundoX2, 0))
6
7     # carrega a imagem do botao de audio de acordo com o status de audio do jogo
8     if game.comAudio:
9         self.botaoSom = pygame.image.load(os.path.join('Imagens', 'audio_ligado.
10                                         png'))
11    else:
12        self.botaoSom = pygame.image.load(os.path.join('Imagens', 'audio_desligado.png'))
13
14     game.janela.blit(self.botaoSom, (1200, 20))

```

Código 100: Método `desenharTelaBasica` da classe `Tela`.

Observe que no Código 100 existem comandos `if-else` para carregar a imagem correta do botão de áudio, de acordo com o comportamento do áudio do jogo naquele momento, o qual é definido pela variável booleana `game.comAudio`. Basicamente, se esta variável for verdadeira, carrega-se a imagem correspondente ao áudio ligado, caso contrário, carrega-se a imagem correspondente ao áudio desligado. Mais informações sobre a necessidade de inserir duas imagens de fundo na tela de jogo serão dadas na Subseção 13.2.6.3.

Agora, vamos implementar o comportamento do botão de áudio. Para conseguirmos implementar essa função, precisamos entender como o *Pygame* nos auxilia a perceber as interações do usuário com o jogo. Basicamente, cada vez que o usuário tenta interagir com o jogo, como pressionar alguma tecla ou botão do *mouse*, o *Pygame* adiciona um novo evento em uma fila de eventos.

Então, devemos utilizar uma função própria do *Pygame* para eventos, denominada `type` para identificar qual é o tipo daquele evento. Vamos considerar que, no caso do botão de áudio, existem dois principais tipos de eventos que podem ser identificados como tentativas de se alterar o comportamento do áudio:

- Se o tipo de evento for igual a `pygame.MOUSEBUTTONDOWN` (botão do *mouse* pressionado), devemos verificar a posição do *mouse* na tela. Se essa posição estiver restrita às dimensões da imagem do botão de áudio, isso significa que o usuário deseja alterar o comportamento do áudio do jogo.
- Se o tipo de evento for igual a `pygame.KEYDOWN` (tecla pressionada), devemos verificar qual foi a tecla pressionada. Se essa tecla corresponder a tecla ‘m’ (de *mute* em inglês), isso significa que o usuário deseja alterar o comportamento do áudio do jogo.

Toda vez que o usuário solicitar a alteração do comportamento do áudio do jogo, uma entre duas coisas podem ocorrer: se o jogo estava com áudio (variável booleana `game.comAudio` for verdadeira), o jogo deve

ser silenciado e para fazer isso, utilizaremos o método `game.administradorDeAudio.deixarSomMudo(game)` da classe `administradorDeAudio`<sup>5</sup>. Do contrário (se a variável booleana `game.comAudio` for falsa), o jogo deve voltar a ter sonorização e para fazer isso, utilizaremos o método `game.administradorDeAudio.deixarSomTocar(game)` da classe `administradorDeAudio`.

Desta maneira, o Código 101 apresenta o método `comportamentoBotaoDeAudio`, o qual implementa o comportamento esperado para o botão de áudio. Os principais argumentos deste método são `eventos` que recebe os eventos identificados pelo *Pygame*, `pos` que recebe a posição do cursor do *mouse* na janela do jogo e `game` que é o administrador do jogo.

```

1 # metodo para lidar com interacoes com o botao de audio, pode ser utilizado em
2 # qualquer tela
3 def comportamentoBotaoDeAudio(self, game, evento, pos):
4     if (evento.type == pygame.MOUSEBUTTONDOWN and pos[0] > 1200 and pos[0] <
5         1230 and pos[1] > 20 and pos[1] < 45) or (evento.type == pygame.KEYDOWN
6         and pygame.key.get_pressed()[pygame.K_m]):
7         if game.comAudio:
8             game.administradorDeAudio.deixarSomMudo(game)
9         else:
10            game.administradorDeAudio.deixarSomTocar(game)

```

Código 101: Método `comportamentoBotaoDeAudio` da classe `Tela`.

Já o método responsável pelo comando de saída do jogo é semelhante ao `comportamentoBotaoDeAudio`, todavia, os eventos identificados como intenção de o usuário sair do jogo são os seguintes:

- `pygame.QUIT`, isto é, quando o usuário clica no botão de fechar da janela do jogo;
- `pygame.KEYDOWN` com `pygame.key.get_pressed()[pygame.K_ESCAPE]`, isto é, tecla ESC pressionada.

Desta maneira, o Código 102 apresenta o método `comportamentoBotaoDeSair`, o qual implementa o comportamento esperado para que o usuário encerre a execução do jogo.

```

1 def comportamentoBotaoDeSair(self, game, evento):
2     if evento.type == pygame.QUIT or (evento.type == pygame.KEYDOWN and pygame.
3         key.get_pressed()[pygame.K_ESCAPE]):
4         game.usuarioSaiu = True

```

Código 102: Método `comportamentoBotaoDeSair` da classe `Tela`.

É válido ressaltar que no Código 102, setar a variável `game.usuarioSaiu` para `True` é suficiente para que o jogo pare de ser executado, pois a condição utilizada no laço de repetição que faz o jogo continuar a sendo executado - presente no método `run` da classe `AdministradorDoJogo` (arquivo `main.py`) - é `while not game.usuarioSaiu`.

Após completarmos a criação da classe `Tela`, podemos partir para as outras telas do jogo, que herdarão seus atributos e métodos.

### 13.2.1 Tela de Início

Nesta subseção vamos desenvolver a tela inicial do nosso jogo que será semelhante à tela inicial representada na Figura 14. Para atingir nosso objetivo, vamos utilizar conceitos relacionados à definição de classes e herança e seguir os seguintes passos:

- Inicializar a classe `TelaDeInicio`;
- Desenhar os elementos na tela;
- Definir o comportamento do botão de Instruções;
- Definir o comportamento do botão de *play*;
- Interpretar os eventos acionados pelo usuário;
- Implementar o laço de repetição que executa a tela.

<sup>5</sup>Esse método será melhor explicado na Subseção 13.7.1.



Figura 14: Tela inicial do jogo.

### 13.2.1.1 Inicializando a classe TelaDeInicio

O primeiro passo necessário para a construção da classe `TelaDeInicio` (arquivo `TelaDeInicio.py`) consiste em inicializar os atributos da classe (variáveis tipo `self`) que armazenarão conteúdos da tela. Assim como demonstrado na seção dedicada à classe `Tela`, deve-se iniciar a implementação de uma tela declarando o seu construtor.

Neste construtor, primeiramente, invocaremos o construtor da classe-pai de `TelaDeInicio`, isto é, a classe `Tela`. Por causa disso, utilizaremos a instrução `super().__init__()`.

Logo após isso, utilizaremos funções da biblioteca `Pygame` responsáveis por inicializar variáveis para imagens, fontes, e palavras que serão introduzidas tela. Para inicializar imagens, utilizaremos a função `pygame.image.load()`, conforme descrito na Subseção 13.2.

Com relação aos elementos de texto presentes na tela inicial, deve-se, primeiramente, definir as fontes que serão utilizadas para eles. Para inicializar uma fonte, deve-se utilizar a seguinte instrução: `pygame.font.FONT(<caminho do arquivo que armazena a fonte>, <tamanho da fonte>)`. Para obter o caminho do arquivo que armazena a fonte, utilizaremos a seguinte instrução: `os.path('Fontes', <nome da fonte>)`, exatamente como fizemos para obter o caminho dos arquivo de imagens, trocando apenas a pasta do caminho para ‘Fontes’, ao invés de ‘Imagens’.

Em seguida, para carregar as palavras que aparecerão na tela, deve-se usar a seguinte sintaxe: `self.nomeDaFonte.render('INSTRUÇÕES', True, <cor do texto>)`. Nesse caso, carregariammos a palavra ‘INSTRUÇÕES’ com a fonte `nomeDaFonte`, com a cor que digitássemos em `<cor de texto>`. É importante ressaltar que as cores que forem utilizadas devem ser declaradas no arquivo `Configuracoes.py` com seus respectivos códigos ***RGB***.

Mas o que é código ***RGB***? Em computação, as cores são identificadas como a combinação de diferentes valores de intensidade para as três cores primárias (Vermelho, Verde e Azul, ou *RGB*, do inglês, *Red-Green-Blue*) e, a partir delas, pode-se criar as demais cores.

Basicamente, cada uma dessas cores recebe um valor de intensidade que varia de 0 a 255. O vermelho, por exemplo, é obtido quando se tem o maior valor para a cor vermelha e o menor valor para as outras cores, ou seja, 255 para *Red*, e 0 para *Green* e *Blue*. Já o branco é obtido quando se tem o maior valor para as 3 cores, ou seja, 255 para *Red*, *Green* e *Blue*. Sendo assim, as declarações de algumas cores utilizando o código *RGB* estão exemplificadas no Código 103.

```

1 BRANCO = (255, 255, 255)
2 PRETO = (0, 0, 0)
3 VERMELHO = (255, 0, 0)

```

Código 103: Declaração de cores no arquivo `Configuracoes.py`.

Neste projeto, já definimos algumas cores para lhe auxiliar. No entanto, se você quiser utilizar outras cores, basta procurar na *internet* seu respectivo código *RGB* e declará-la no arquivo `Configuracoes.py`, conforme realizado no Código 103.

Dessa maneira, um diagrama esquemático das inicializações que devem ser realizadas no construtor da classe `TelaDeInicio` está representado na Figura 15. Ademais, uma declaração possível para o construtor da `TelaDeInicio` está registrada no Código 104.

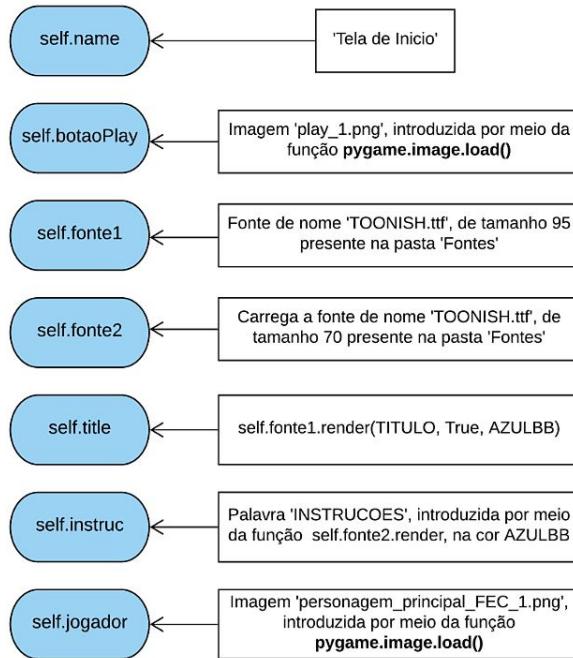


Figura 15: Diagrama esquemático das inicializações que devem ser realizadas no construtor da classe `TelaDeInicio`.

```

1 # declaracao de bibliotecas
2 from Tela import *
3 import pygame
4 import os
5
6 class TelaDeInicio(Tela):
7     def __init__(self, game):
8         # declaracao do construtor da classe-pai Tela
9         super().__init__()
10
11     # definindo o nome da tela
12     self.name = 'Tela de Inicio'
13
14     # carregando a imagem do botao de play
15     self.botaoPlay = pygame.image.load(os.path.join('Imagens', 'play_1.png'))
16
17     # carregando a fonte para o titulo do jogo
18     self.fonte1 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 95)
19     # carregando a fonte para o 'botao' de instrucoes
20     self.fonte2 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 70)
21
22     # carregando o texto do titulo do jogo
23     self.titulo = self.fonte1.render(TITULO, True, AZULBB)
24
25     # carregando o texto do botao de instrucoes
26     self.inst = self.fonte2.render('INSTRUCOES', True, AZULBB)

```

Código 104: Construtor da classe `TelaDeInicio`.

### 13.2.1.2 Desenhando a tela de início

O primeiro método que devemos implementar na classe `TelaDeInicio` é o `desenharTela` que consiste em desenhar os elementos que inicializamos no construtor da tela, isto é, desenhar o `botaoPlay`, `titulo` e `inst` (botão de instruções).



1. Implemente o método `desenharTela(self, game)`, tendo como base o método `desenharTelaBasica(self, game)`, descrito no Código 100. Lembre-se que já comen-tamos na Subseção 13.2 que para desenhar algo na tela do jogo, devemos utilizar a instrução `game.janela.blit(<imagem>, <posicao a ser inserida na tela>)`.

Você se lembra que nós já havíamos declarado um método lá na classe `Tela` responsável por desenhar a tela de fundo e o botão de áudio? Pois bem, precisamos chamá-lo antes do método `desenharTela` para que a imagem de fundo seja desenhada na janela do jogo antes dos itens que devem ficar sobre ela, como, por exemplo, o botão de *play*. Por causa disso, utilizaremos um método auxiliar denominado `desenhar` que chamará, primeiramente, o método `desenharTelaBasica` (comum a todas as telas e, portanto, declarado na classe `Tela`) e, posteriormente, o método `desenharTela` que você implementou. Por último, utilizamos a função `pygame.display.flip()` para confirmar o desenho conjunto de todos os elementos na janela do jogo. A implementação do método `desenhar` da classe `TelaDeInicio` está descrita no Código 105.

```
1 def desenhar(self, game):
2     self.desenharTelaBasica(game)
3     self.desenharTela(game)
4     pygame.display.flip()
```

Código 105: Método `desenhar` da classe `TelaDeInicio`.

### 13.2.1.3 Comportamento do botão de instruções

Outro método que deve ser implementado na classe `TelaDeInicio` é o responsável pelo comportamento do botão de instruções. Queremos que este botão se comporte da seguinte forma: quando o jogador colocar o cursor sobre a palavra ‘INSTRUÇÕES’, ela deve adquirir uma cor amarela e quando o cursor sair da área correspondente ao botão, ela deve voltar a ter sua cor original. Além disso, se o usuário clicar na palavra ‘INSTRUÇÕES’, a tela atual do jogo deve ser atualizada para a tela de instruções. Um exemplo de implementação do comportamento do botão de instruções está descrito no Código 106.

```
1 # metodo para lidar com interacoes com o botao que direciona para a tela de
2 # instrucoes
3 def comportamentoBotaoDeInstrucoes(self, game, evento, pos):
4     # se o cursor estiver posicionado sobre a palavra 'INSTRUÇÕES'
5     if pos[0] > 445 and pos[0] < 805 and pos[1] > 540 and pos[1] < 600:
6         # se o usuario nao tiver clicado nesta regiao
7         if evento.type != pygame.MOUSEBUTTONDOWN:
8             # alterar a cor da palavra 'INSTRUÇÕES'
9             self.inst = self.fonte2.render('INSTRUÇÕES', True, AMARELO)
10            # do contrario, mudar de tela
11        else:
12            game.ultimaTela = 'Tela de Inicio'
13            game.telaAtual = 'Tela de Instrucoes'
14            # do contrario
15        else:
16            # retornar a cor da palavra 'INSTRUÇÕES' para a cor original
17            self.inst = self.fonte2.render('INSTRUÇÕES', True, AZULBB)
```

Código 106: Método `comportamentoBotaoDeInstrucoes` da classe `TelaDeInicio`.

No Código 106, note que `pos[0]` e `pos[1]` indicam, respectivamente as coordenadas x e y do cursor na tela. Por causa disso, para identificar se o cursor está sobre a palavra ‘INSTRUÇÕES’, devemos ver se a coordenada x está entre as extremidades esquerda e direita da palavra. Além disso, precisamos verificar se a coordenada y está entre as extremidades superior e inferior da palavra. Essas verificações estão presentes na linha 3 do Código 106.

Além disso, é importante notar que para alternar entre as telas do jogo, basta trocar o nome armazena-do na variável `game.telaAtual` pelo nome da tela que você deseja ser redirecionado para. Isso porque, as telas do jogo são alternadas pelo administrador do jogo (classe `AdministradorDoJogo` declarada no arquivo `main.py`), o qual possui o método `run`, responsável por manter o jogo funcionando, alternando entre as telas. A implementação do método `run` da classe `AdministradorDeJogo` (arquivo `main.py`) está descrita no Código 107.

```

1 # metodo que executa o jogo, alternando entre as telas do jogo, de acordo com os
2 # comandos do usuario
3 def run(self):
4     while not self.usuarioSaiu:
5         if self.telaAtual == 'Tela de Inicio':
6             self.tela = TelaDeInicio(self)
7         elif self.telaAtual == 'Tela de Instrucoes':
8             self.tela = TelaDeInstrucoes()
9         elif self.telaAtual == 'Tela de Jogo':
10            self.tela = TelaDeJogo(self)
11        elif self.telaAtual == 'Tela de Perguntas':
12            self.tela = TelaDePerguntas(self)
13        elif self.telaAtual == 'Tela Resultado da Pergunta':
14            self.tela = TelaResultadoDaPergunta()
15        else:
16            self.tela = TelaDeFim(self)
17
18        self.tela.run(self)
19
20 pygame.quit()

```

Código 107: Método `run` da classe `AdministradorDeJogo` (arquivo `main.py`).

### 13.2.1.4 Comportamento do botão *play*

Outro método importante para o funcionamento da tela de início consiste em implementar o comportamento do botão *play*. Queremos que ele se comporte da seguinte forma: quando o jogador colocar o cursor sobre o botão, ele deve mudar sua imagem para outra com o brilho alterado (alternar entre a imagem `play_x.png` e `play_brilho_x.png`, em que `x` é o número do botão que você está utilizando) e quando o cursor sair da área correspondente ao botão, ele deve voltar a ter sua imagem original. Além disso, se o usuário clicar no botão *play*, a tela atual do jogo deve ser atualizada para a tela de jogo.



1. Implemente o método `comportamentoBotaoDeJogar(self, game, evento, pos)`, tendo como base o método `comportamentoBotaoDeInstrucoes(self, game, evento, pos)`, descrito no Código 106.

### 13.2.1.5 Interpretando os eventos disparados pelo usuário

Além dos métodos já descritos, precisamos inserir um método para interpretar os eventos realizados pelo usuário e acionar os comportamentos definidos para esta tela, como `comportamentoBotaoDeInstrucoes` e `comportamentoBotaoDeJogar`, ou ainda os comportamentos definidos para todas as telas, como `comportamentoBotaoDeSair` e `comportamentoBotaoDeAudio`.

Para pegar os eventos, utilizaremos o método `pygame.event.get()` que retorna uma lista de eventos disparados pelo usuário. Então, para cada evento nesta lista, acionaremos os comportamentos dos botões da tela. É importante lembrar que a maioria dos métodos que definem comportamentos de botões da tela também precisam saber a posição do cursor na tela. Para registrar essa posição, utilizaremos a instrução `pygame.mouse.get_pos()`. Uma possível implementação para o método `interpretarEventos` da classe `TelaDeInicio` está registrada no Código 108.

```

1 def interpretarEventos(self, game):
2     game.clock.tick(game.fps)
3
4     # verificando cada um dos eventos acionados pelo usuario
5     for evento in pygame.event.get():
6         # armazenando a posicao do cursos do mouse na janela do jogo
7         pos = pygame.mouse.get_pos()
8
9         # checa se o usuario quer sair do jogo
10        self.comportamentoBotaoDeSair(game, evento)

```

```

11
12     # checa se o usuario quer tirar o som
13     self.comportamentoBotaoDeAudio(game, evento, pos)
14
15     # checa se o usuario clicou no botao para abrir a tela de instrucoes
16     self.comportamentoBotaoDeInstrucoes(game, evento, pos)
17
18     # checa se o usuario quer jogar
19     self.comportamentoBotaoDeJogar(game, evento, pos)

```

Código 108: Método `interpretarEventos` da classe `TelaDeInicio`.

Repare que no Código 108, na linha 2, utilizamos a função: `game.clock.tick(game.fps)`. Para entender melhor o objetivo de utilização desta instrução precisamos entender o que é taxa de **quadros por segundo** (do inglês, *Frames Per second*, abreviado para *FPS*).

Basicamente, qualquer ação que visualizamos em filmes, vídeos ou jogos é composta por quadros. Eles são nada mais do que imagens sequenciais que, ao serem reproduzidas em velocidade, dão a sensação de movimento. É possível fazer um teste simples em casa para entender como funcionam os vídeos. Tente desenhar várias figuras similares em um bloco e em cada folha deslocá-las um pouco. Ao folhear o bloco, você terá impressão que a figura está em movimento. Com os vídeos funciona da mesma forma. No nosso jogo, o *FPS* é de 60 quadros por segundo o que significa, efetivamente, que o código do jogo é executado 60 vezes por segundo. Essa taxa de *FPS* está declarada no arquivo `Configuracoes.py`.

Agora que você já sabe o que significa taxa de quadros por segundo *FPS*, você vai conseguir entender a necessidade de utilização da função `game.clock.tick(game.fps)`. Basicamente, ela é responsável por manter o relógio (do inglês, *clock*) sendo atualizado com a taxa de quadros por segundo *FPS* que desejamos. Este método deve ser chamado uma vez por quadro. Ele calculará quantos milissegundos se passaram desde a chamada do quadro anterior e ao passar como argumento a taxa de quadros por segundo *FPS* desejada para a execução do jogo, este método poderá atrasar a velocidade de execução de um jogo com o intuito de manter a taxa de *FPS* desejada constante. Por exemplo, ao chamar `clock.tick(60)` uma vez por quadro, o programa nunca será executado em uma taxa maior que 60 quadros por segundo.

O método `interpretarEventos` apresentado no Código 108 será utilizado em várias outras telas do jogo com o objetivo de interpretar as interações do usuário.

### 13.2.1.6 Executando a tela de início

Por último, o método presente nas telas do jogo que faz com que as telas sejam executadas é o método `run` que executa a tela, isto é, interpreta os eventos acionados pelo usuário e desenha os elementos na tela, enquanto o nome da tela atual for igual ao nome desta tela e enquanto o usuário ainda não mostrou interesse em sair do jogo. A implementação do método `run` da classe `TelaDeInicio` está registrada no Código 109.

```

1 def run(self, game):
2     while game.telaAtual == self.name and not game.usuarioSaiu:
3         self.interpretarEventos(game)
4         self.desenhar(game)

```

Código 109: Método `run` da classe `TelaDeInicio`.

O método `run` apresentado no Código 109 será utilizado em várias outras telas do jogo com o objetivo de executar as funcionalidades das telas.

### 13.2.2 Tela de Fim

Nesta subseção desenvolveremos uma tela de fim de jogo semelhante à representada na Figura 16.



Figura 16: Tela de fim de jogo.

Para atingir este objetivo, seguiremos o seguinte roteiro:

- Inicializar a classe `TelaDeFim`;
- Desenhar os elementos na tela;
- Definir o comportamento do botão de *replay*.

#### 13.2.2.1 Inicializando a classe `TelaDeFim`

Assim como fizemos na criação da tela de início, devemos, primeiramente, inicializar os atributos (variáveis do tipo `self`) da classe `TelaDeFim` (arquivo `TelaDeFim.py`) que guardarão as fontes utilizadas, os textos, o botão de `replay`, o `score` e o melhor `score`. Uma declaração incompleta para o construtor da classe `TelaDeFim` está descrita no Código 110.

```

1 # declaracao de bibliotecas
2 from Configuracoes import *
3 from TelaDeJogo import *
4 from Tela import *
5 import pygame
6 import os
7 class TelaDeFim(Tela):
8     def __init__(self, game):
9         # declaracao do construtor da classe -pai Tela
10        super().__init__()
11
12        # definindo o nome da tela
13        self.name = 'Tela de Fim'
14
15        # carregando as fontes
16        self.fonte1 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 95)
17        self.fonte2 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 65)
18
19        # carregando a imagem do botao de replay
20        self.botaoReplay = pygame.image.load(os.path.join('Imagens', 'replay_2.
21          png'))
22
23        # carregando a mensagem 'FIM DE JOGO', na cor amarela com a fonte 1
24        self.fimJogo = self.fonte1.render('FIM DE JOGO!', True, AMARELO)
25
26        # atualizando a melhor pontuacao do jogo
27        if game.pontuacao > game.melhorPontuacao:
28            game.melhorPontuacao = game.pontuacao
29
30        # carregando os valores da ultima pontuacao e da melhor pontuacao em
31          forma de texto a ser desenhado na tela

```

```

31     self.pontuacaoFinalNum = self.fonte1.render(str(game.pontuacao), True,
32                                         AMARELO)
      self.melhorPontuacaoNum = self.fonte1.render(str(game.melhorPontuacao),
      True, AMARELO)

```

Código 110: Declaração incompleta para o construtor da classe *TelaDeFim*.



- Logo abaixo da definição do texto de ‘FIM DE JOGO’, da classe *TelaDeFim*, inicialize as variáveis que guardarão outros textos a serem exibidos nesta tela: “SCORE FINAL” e “MELHOR SCORE”, com a cor de sua preferência, tendo como referência o Código 110.

**Dica:** Lembre-se também de utilizar nomes facilmente legíveis para as variáveis; podemos convencionar em chamá-las de *fimJogo*, *scoreFinal* e *melhorScore*.

Note que nas linhas 21 e 22 do Código 110 verificamos se a pontuação do último jogo (a qual fica armazenada na variável *game.pontuacao*) é maior que a melhor pontuação conquistada dentre todas as partidas executadas (a qual fica armazenada na variável *game.melhorPontuacao*). Se isso for verdade, atualizamos a variável *game.melhorPontuacao* com o valor da pontuação da última partida. Além disso, nas linhas 26 e 27, carregamos os valores da pontuações em forma de texto capaz de ser mostrado na tela.

### 13.2.2.2 Desenhando a tela de fim

Da mesma forma que existe o método *desenharTela* lá na classe *TelaDeInicio*, na tela de fim esse método também deverá ser implementado com o intuito de desenhar os elementos na tela.



- Implemente o método *desenharTela(self, game)*, tendo como base o método *desenharTelaBasica(self, game)*, descrito no Código 100. Lembre-se de desenhar todos os elementos desta tela, isto é: os textos ‘FIM DE JOGO’, ‘SCORE FINAL’ e ‘MELHOR SCORE’, assim como os textos das pontuações e o botão de *replay*.

### 13.2.2.3 Comportamento do botão *replay*

O método responsável pelo comportamento do botão *replay* funciona de forma semelhante ao do comportamento do botão *play* da tela de ínicio: quando o cursor do *mouse* estiver nos limites da imagem do botão, trocamos a imagem do botão para aquela em que ele possui brilho diferente (*replay\_brilho\_X.png*) e se o usuário clicar na imagem do botão, trocamos a tela atual para a Tela de Jogo.



- Implemente o método *comportamentoBotaoDeReplay(self, game, evento, pos)*, tendo como base o método *comportamentoBotaoDeInstrucoes(self, game, evento, pos)*, descrito no Código 106.

### 13.2.3 Tela de Instruções

Nessa etapa, vamos criar uma tela de instruções de jogo, para auxiliar o usuário a entender o funcionamento do jogo. A tela que desenvolveremos nesta subseção será semelhante à representada na Figura 17.

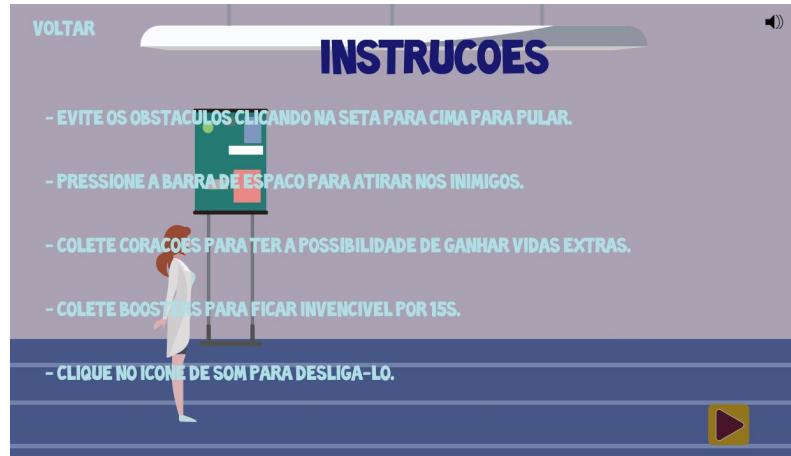


Figura 17: Tela de Instruções

Para atingir esses objetivos, seguiremos os seguintes passos:

- Inicializar a classe `TelaDeInstrucoes`;
- Desenvolver um método auxiliar para imprimir as instruções na tela de forma igualmente espaçada;
- Desenhar os elementos na tela;
- Definir o comportamento do botão de *play*.
- Definir o comportamento do botão de *voltar*.

#### 13.2.3.1 Inicializando a classe `TelaDeInstrucoes`

Assim como fizemos na criação da tela de fim, devemos, primeiramente, inicializar os atributos (variáveis do tipo `self`) da classe `TelaDeInstrucoes` (arquivo `TelaDeInstrucoes.py`) que guardarão as fontes utilizadas, os textos das instruções, o botão de *play* e o botão de *voltar*. Uma declaração para o construtor da classe `TelaDeInstrucoes` está descrita no Código 111.

```

1 # declaracao de bibliotecas
2 from Tela import *
3 import pygame
4 import os
5
6 class TelaDeInstrucoes(Tela):
7     def __init__(self):
8         # declaracao do construtor da classe -pai Tela
9         super().__init__()
10
11     # definindo o nome da tela
12     self.name = "Tela de Instrucoes"
13
14     # carregando as fontes
15     self.fonte1 = self.font = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 70)
16     self.fonte2 = self.font = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 30)
17
18     # carregando o titulo da tela
19     self.titulo = fonte1.render('INSTRUÇÕES', True, NAVY)
20
21     # carregando a mensagem 'VOLTAR' para o botao voltar
22     self.botaoVoltar = self.fonte2.render('VOLTAR', True, AZULBB)
23
24     # carregando a imagem do botao play

```

```

25     self.botaoPlay = self.play = pygame.image.load(os.path.join('Imagens', 'play_1.png'))

```

Código 111: Construtor da classe `TelaDeInstrucoes`.

### 13.2.3.2 Método auxiliar `imprimirInstrucoes`

Na classe `TelaDeInstrucoes` iremos implementar um método auxiliar para imprimir os textos das instruções de forma igualmente espaçada. Este método será denominado `imprimirInstrucoes` e terá como argumentos o administrador do jogo (`game`), o número da instrução (`num`) e o texto da instrução (`tex`). Dados estes argumentos, ele imprimirá na tela, na posição correta, o texto da instrução. Uma implementação possível para este método está descrita no Código 112.

```

1 def imprimirInstrucoes(self, game, num, text):
2     # carrega a instrucao com a fonte 2 e a cor azul bebe
3     instrucao = self.fonte2.render(text, True, AZULBB)
4
5     # posiciona a instrucao na tela
6     game.janela.blit(instrucao, (70, 170 + 100*(num - 1)))

```

Código 112: Método `imprimirInstrucoes` da classe `TelaDeInstrucoes`.

No Código 112, note que a instrução é posicionada com a sua coordenada `x` (posição horizontal) valendo 70 *pixels*. Além disso, a sua coordenada `y` (posição vertical) varia de acordo com a seguinte função:  $170 + 100 * (num - 1)$ . Perceba que, utilizando esta função, a primeira instrução (`num = 1`) será posicionada na coordenada `y` igual a  $170 + 100 * (1 - 1) = 170$ , já a segunda instrução (`num = 2`) será posicionada na coordenada `y` igual a  $170 + 100 * (2 - 1) = 170 + 100 = 270$ , e assim por diante. Dessa maneira, como usamos sempre o mesmo tamanho de fonte para todas as instruções, elas aparecerão na tela de forma igualmente espaçadas.

Para utilizar método auxiliar `imprimirInstrucoes` e, efetivamente, imprimir as instruções na tela deveremos chamá-lo dentro do método `desenharTela` que desenvolveremos na próxima atividade. Essa chamada deverá ser feita da seguinte forma:

```
self.imprimirInstrucoes(game, <numero da instrucao>, <string com o texto da instrucao>).
```

Um exemplo de chamada do método auxiliar `imprimirInstrucoes` está descrito no Código 113.

```

1 # imprimindo a primeira instrucao do jogo com o auxilio do metodo
2     #   imprimirInstrucoes
3 self.imprimirInstrucoes(game, 1, '- Evite os obstaculos clicando na seta para
4     #   cima para pular.')

```

Código 113: Chamada do método `imprimirInstrucoes` da classe `TelaDeInstrucoes`.

### 13.2.3.3 Desenhando a tela de instruções

O segundo método que devemos implementar na classe `TelaDeInstrucoes` é o `desenharTela` que consiste em desenhar os elementos que inicializamos no construtor da tela, isto é, desenhar o `botaoPlay`, o `botaoVoltar`, o `titulo` e os textos das instruções, com o auxílio do método `imprimirInstrucoes`.

**CODElike a girl**

1. Implemente o método `desenharTela(self, game)`, tendo como base o método `desenharTelaBasica(self, game)`, descrito no Código 100. Lembre-se de utilizar o método auxiliar `imprimirInstrucoes` para imprimir as instruções de forma igualmente espaçada.

### 13.2.3.4 Comportamento do botão *play*

Dentro da tela de instruções, possuímos um botão *play*, para que o usuário possa jogar logo após ler as instruções. O método responsável pelo comportamento do botão *play* da tela de instruções funciona de forma semelhante ao comportamento do botão *play* da tela de início: quando o cursor do `mouse` estiver nos limites da imagem do botão, trocamos a imagem do botão para aquela em que ele possui brilho

diferente (`replay_brilho_X.png`) e se o usuário clicar na imagem do botão, trocamos a tela atual para a Tela de Jogo.



1. Implemente o método `comportamentoBotaoDeJogar(self, game, evento, pos)`, tendo como base o método `comportamentoBotaoDeInstrucoes(self, game, evento, pos)`, descrito no Código 106.

#### 13.2.3.5 Comportamento do botão voltar

Além do botão *play*, a tela de instruções também possui o botão voltar, representado pelo texto ‘VOLTAR’. A funcionalidade deste botão é retornar para a tela de início do jogo. De fato, da forma que este jogo será desenvolvido, o usuário não ganha muito retornando à tela de início, dado que os únicos elementos interativos da tela de início são o botão que leva a tela de instruções e o botão *play*, que leva a tela de jogo. Contudo, resolvemos implementar o botão de voltar na tela de instruções para que, no futuro, você possa aprimorar seu jogo e colocar mais opções na tela de início, tornando, então, o botão voltar útil.

Sendo assim, vamos implementar o método responsável pelo comportamento do botão voltar. Nossa intenção aqui é que ele se comporte da seguinte forma: quando o jogador colocar o cursor sobre a palavra ‘VOLTAR’, essa palavra deve adquirir uma cor amarela (como fizemos com o botão de instruções da classe `TelaDeInicio`). Ademais, quando o cursor sair da área correspondente à palavra ‘VOLTAR’, o ícone deve voltar a ter sua cor original. Por último, se o usuário clicar na palavra ‘VOLTAR’, devemos redirecionar a tela atual do jogo para a tela de início.



1. Implemente o método `comportamentoBotaoVoltar`, tendo como referência o Código 106.

#### 13.2.4 Tela de Perguntas

Para deixar nosso jogo mais interessante, implementaremos a classe `TelaDePerguntas` com um pequeno *quiz*, semelhante ao representado pela Figura 18.

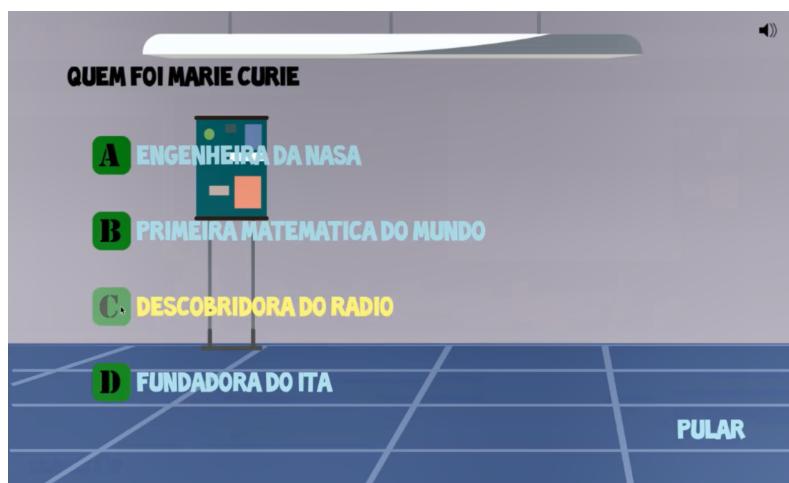


Figura 18: Tela de Perguntas com uma alternativa selecionada.

O objetivo da tela de perguntas é deixar o jogo com um viés educativo, apesar da ação. A nossa ideia é que esta tela seja disparada toda vez que o jogador colidir com um ícone de vida extra, de forma que, se o usuário quiser essa vida extra ele deverá acertar a pergunta. Se ele errar, o jogo termina. Daremos ainda, uma “colher de chá” deixando o usuário pular a questão caso não saiba, com a penalidade de não ganhar a vida extra.

Para implementar essa tela, seguiremos os seguintes passos:

- Definir quais perguntas queremos colocar em nosso *quiz*;
- Declarar imagens e fontes de textos que usaremos para exibir corretamente a tela e todo o seu conteúdo;
- Implementar métodos convenientes para a execução da tela;
- Escrever um algoritmo que lerá no arquivo `main.py` uma pergunta e suas respectivas alternativas de maneira aleatória;
- Enviar o resultado da pergunta para o arquivo `TelaResultadoDaPergunta.py`, a fim de exibir o resultado: se está correta a resposta do usuário ou não.

#### 13.2.4.1 Inserção de perguntas na tela

Para inserir as perguntas em nosso *quiz*, suas respectivas alternativas e resposta, iremos implementar o método `inserirPerguntas` na classe `AdministradorDoJogo` (arquivo `main.py`), conforme descrito no Código 114.

```

1 # dentro da classe AdministradorDoJogo ,
2 # colocamos, por exemplo a pergunta de numero 1
3 def inserirPerguntas(self):
4     # adiciona a pergunta 1 no vetor de perguntas
5     self.perguntas.append("Quem foi Ada Lovelace?")
6
7     # adiciona as alternativas da pergunta 1 nos vetores de alternativas
8     self.alternativaA.append("Engenheira")
9     self.alternativaB.append("Primeira programadora do mundo")
10    self.alternativaC.append("Matematica")
11    self.alternativaD.append("Descobridora do DNA")
12
13    # adiciona a resposta correta para a pergunta 1 no vetor de respostas
14    self.respostas.append('B')
```

Código 114: Método `inserirPerguntas` da classe `AdministradorDoJogo` (arquivo `main.py`).

No Código 114, cada pergunta é armazenada no vetor `self.perguntas`, assim como cada uma das quatro alternativas da pergunta é armazenada nos seus respectivos vetores `self.alternativaA`, `self.alternativaB`, `self.alternativaC` e `self.alternativaD`. Além disso, a resposta correta da pergunta é armazenada no vetor `self.respostas`.



1. Replique a estrutura mostrada no Código 114 para outras perguntas que você deseja inserir em seu jogo.

#### 13.2.4.2 Inicializando a classe TelaDePerguntas

A inicialização da classe `TelaDePerguntas` (arquivo `TelaDePerguntas.py`) é realizada com a declaração do método `__init__`, chamado de construtor. Nele, devemos:

- Declarar o nome desta tela;
- Carregar as imagens dos ícones das alternativas;
- Inicializar variáveis auxiliares que irão armazenar a pergunta e as alternativas a serem mostradas na tela;
- Carregar as fontes que serão utilizadas nos textos.
- Declarar o texto do botão de pular.

Observe o Código 115, no qual está descrito a implementação completa do construtor da classe `TelaDePerguntas`.

```

1  from Configuracoes import *
2  from Tela import *
3  import random
4  import pygame
5  import os
6
7  class TelaDePerguntas(Tela):
8      def __init__(self, game):
9          # declaracao do construtor da classe-pai Tela
10         super().__init__()
11
12         # definindo o nome da tela
13         self.name = 'Tela de Perguntas'
14
15         # carregando os icones das alternativas
16         self.altA = pygame.image.load(os.path.join('Imagens', 'alternativa_A.png'))
17         self.altB = pygame.image.load(os.path.join('Imagens', 'alternativa_B.png'))
18         self.altC = pygame.image.load(os.path.join('Imagens', 'alternativa_C.png'))
19         self.altD = pygame.image.load(os.path.join('Imagens', 'alternativa_D.png'))
20
21         # inicializando variaveis auxiliares que guardarao as perguntas e
22         # alternativas a serem expostas na tela
23         self.pergunta = 0
24         self.altATexto = 0
25         self.altBTexto = 0
26         self.altCTexto = 0
27         self.altDTexto = 0
28
29         # carregando as fontes que serao utilizadas
30         self.fonte1 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 40)
31         self.fonte2 = self.font = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 40)
32         self.pular = self.fonte2.render("Pular", True, AZULBB)

```

Código 115: Construtor da classe Tela

### 13.2.4.3 Escolha aleatória da pergunta que aparecerá na tela

Você se lembra que declaramos lá na classe `AdministradorDoJogo` (arquivo `main.py`) as perguntas do nosso *quiz* nos vetores `self.perguntas`, `self.alternativaA`, `self.alternativaB`, `self.alternativaC` e `self.alternativaD`? Então, agora utilizaremos estes vetores para carregar uma pergunta aleatória na tela.

Para atingir esse objetivo, implementaremos um método denominado `carregarPerguntas`, o qual selecionará aleatoriamente, com o auxílio da função `random.randrange(0, len(game.perguntas) - 1)`, uma posição do vetor que armazena as perguntas.

Após selecionado um índice, carregaremos as variáveis `self.pergunta` e `self.altXTexto` (com X sendo a alternativa), com as *strings* da pergunta e das alternativas presentes naquele índice. O Código 116 exemplifica como carregar uma pergunta e a alternativa A na tela de perguntas. Este código também armazena na variável `game.respostaCorreta` a alternativa correta para aquela pergunta.

```

1  def carregarPergunta(self, game):
2      # a variavel 'game.i' guarda um numero aleatorio cujo valor varia entre 0 e
3      # o ultimo indice do vetor de perguntas
4      game.i = random.randrange(0, len(game.perguntas) - 1)
5      # esse valor armazenado em 'game.i' correspondera ao indice da pergunta
       # selecionada para ser mostrada naquela execucao da tela de perguntas

```

```

6     #variavel 'self.pergunta' recebe qual a pergunta a ser lida do vetor de
7         perguntas e carrega-a com a fonte1, na cor preta
8     self.pergunta = self.fonte1.render(game.perguntas[game.i], True, PRETO)
9
10    #variavel 'self.altATexto' recebe qual a alternativa A correspondente a
11        pergunta game.perguntas[game.i] e carrega-a com a fonte2, na cor azul
12        bebe
13    self.altATexto = self.fonte2.render(game.alternativaA[game.i], True, AZULBB)
14
15    # armazena qual a resposta correta da pergunta que foi lida e esta na tela
16        para que a tela de respostas saiba se a resposta do usuario esta
17        correta ou nao
18    game.respostaCorreta = game.respostas[game.i]

```

Código 116: Exemplo de como carregar uma pergunta e uma alternativa na classe `TelaDePerguntas`.



- Carregue as alternativas B, C e D da pergunta selecionada no método `carregarPergunta`, conforme realizado para a alternativa A no Código 116.

#### 13.2.4.4 Desenhando a tela de perguntas

Devemos implementar na classe `TelaDePerguntas` o método `desenharTela` que consiste em desenhar os elementos que inicializamos no construtor da tela, isto é, desenhar o texto da pergunta selecionada (`pergunta`), os ícones das alternativas (`altA`, `altB`, `altC` e `altD`), os textos das alternativas (`altATexto`, `altBTexto`, `altCTexto` e `altDTexto`) e o texto ‘PULAR’ (botão de pular).



- Implemente o método `desenharTela(self, game)`, tendo como base o método `desenharTelaBasica(self, game)`, descrito no Código 100.

#### 13.2.4.5 Comportamento dos botões das alternativas

Nesta subseção implementaremos os métodos responsáveis pelo comportamento dos botões das alternativas. Nossa intenção aqui é que eles se comportem da seguinte forma: quando o jogador colocar o cursor sobre o ícone de uma alternativa, o texto da alternativa deve adquirir uma cor amarela (como fizemos com o botão de instruções da classe `TelaDeInicio`), assim como o ícone da alternativa deve ter seu brilho alterado (como fizemos com o botão de *play* da classe `TelaDeInicio`). Ademais, quando o cursor sair da área correspondente ao ícone da alternativa, o ícone deve voltar a ter seu brilho inicial, assim como o texto da alternativa deve retornar à sua cor original. Por último, se o usuário clicar em uma alternativa, devemos armazenar na variável `game.respostaUsuario` qual é a alternativa escolhida por ele, com o intuito de poder compará-la com a resposta correta, assim como devemos redirecionar a tela atual do jogo para a tela de respostas de perguntas. Um exemplo de implementação do comportamento do botão da alternativa A está descrito no Código 117.

```

1 # metodo para lidar com a escolha da alternativa A
2 # se a escolha do usuario for igual a resposta correta, da tudo certo
3 # se nao, aparece mensagem negativa na outra tela, a de resultado da pergunta
4 def comportamentoBotaoAlternativaA(self, game, evento, pos):
5     if pos[0] > 140 and pos[0] < 200 and pos[1] > 200 and pos[1] < 260:
6         if evento.type != pygame.MOUSEBUTTONDOWN:
7             self.altA = pygame.image.load(os.path.join('Imagens', ,
8                                         'alternativa_brilho_A.png'))
8             self.altATexto = self.fonte2.render(game.alternativaA[game.i], True,
9                                         AMARELO)
9
10        else:
11            game.respostaUsuario = 'A'

```

```

11         game.ultimaTela = 'Tela de Perguntas'
12         game.telaAtual = 'Tela Resultado da Pergunta'
13     else:
14         self.altA = pygame.image.load(os.path.join('Imagens', 'alternativa_A.png
15             '))
15         self.altATexto = self.fonte2.render(game.alternativaA[game.i], True,
AZULBB)

```

Código 117: Método `comportamentoBotaoDeInstrucoes` da classe `TelaDeInicio`.

**CODE like a girl**

1. Implemente os métodos `comportamentoBotaoAlternativaB`, `comportamentoBotaoAlternativaC` e `comportamentoBotaoAlternativaD`, tendo como referência o Código 117.

#### 13.2.4.6 Comportamento do botão pular

Nesta subseção implementaremos o método responsável pelo comportamento do botão pular. Nossa intenção aqui é que ele se comporte da seguinte forma: quando o jogador colocar o cursor sobre a palavra ‘PULAR’, essa palavra deve adquirir uma cor amarela (como fizemos com o botão de instruções da classe `TelaDeInicio`). Ademais, quando o cursor sair da área correspondente à palavra ‘PULAR’, o ícone deve voltar a ter sua cor original. Por último, se o usuário clicar na palavra ‘PULAR’, devemos redirecionar a tela atual do jogo para a tela de jogo.

**CODE like a girl**

1. Implemente o método `comportamentoBotaoPular`, tendo como referência o Código 106.

#### 13.2.5 Tela de Resposta de Perguntas

Nesta subseção, nós daremos sequência à tela de perguntas. Aqui, receberemos a resposta que o usuário deu e revelaremos se ela está correta ou não. Dado que na tela de perguntas armazenamos a resposta correta e a resposta do usuário em variáveis do administrador de jogo, não será tão difícil implementar esta tela.

Basicamente, nosso intuito com a tela de resposta de perguntas é:

- Se o usuário acertou a resposta da pergunta, ele deve receber uma vida extra e continua a jogar. Então, uma mensagem positiva deve aparecer na tela de resposta de perguntas, conforme representado na Figura 19.

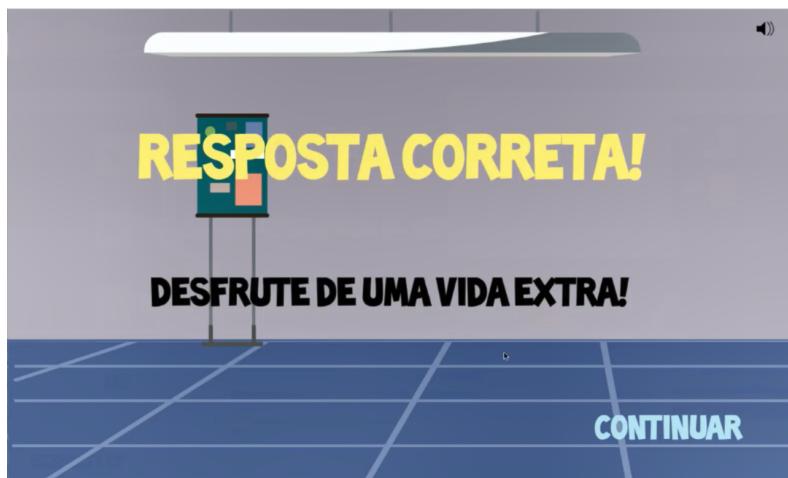


Figura 19: Tela de resposta de perguntas com mensagem positiva.

- Já se o usuário errou a resposta da pergunta, ele perderá o jogo. Então, uma mensagem negativa deve aparecer na tela de resposta de perguntas, conforme representado na Figura 20.

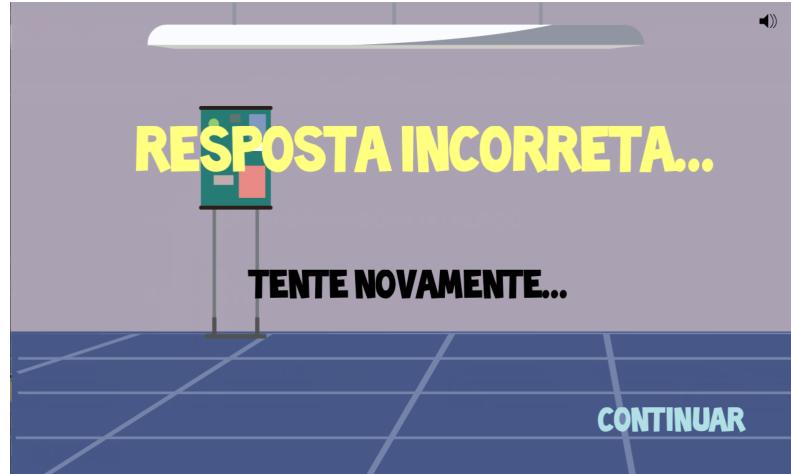


Figura 20: Tela de resposta de perguntas com mensagem negativa.

Para atingir esses objetivos, seguiremos os seguintes passos:

- Inicializar a classe `TelaResultadoDaPergunta`;
- Desenhar os elementos na tela;
- Definir o comportamento do botão de *continuar*.
- Definir o comportamento do botão de *voltar*.

#### 13.2.5.1 Inicializando a classe `TelaResultadoDaPergunta`

Assim como fizemos na criação da tela de perguntas, devemos, primeiramente, inicializar os atributos (variáveis do tipo `self`) da classe `TelaResultadoDaPergunta` (arquivo `TelaResultadoDaPergunta.py`) que guardarão as fontes utilizadas, as mensagens de resposta correta ou incorreta e o botão de *continuar*. Uma declaração para o construtor da classe `TelaResultadoDaPergunta` está descrita no Código 118.

```

1 # declaracao de bibliotecas
2 from Tela import *
3 import pygame
4 import os
5
6 class TelaResultadoDaPergunta(Tela):
7     def __init__(self):
8         # declaracao do construtor da classe -pai Tela
9         super().__init__()
10
11     # definindo o nome da tela
12     self.name = 'Tela Resultado da Pergunta'
13
14     # carregando as fontes que serao utilizadas
15     self.fonte1 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 95)
16     self.fonte2 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 60)
17     self.fonte3 = pygame.font.Font(os.path.join('Fontes', 'TOONISH.ttf'), 50)
18
19     # carregando as mensagens de resposta correta e incorreta
20     self.respCorreta = self.fonte1.render('RESPOSTA CORRETA!', True, AMARELO)
21     self.mensCorreta = self.fonte2.render('Desfrute de uma vida extra!', True,
22                                         PRETO)
23     self.respIncorreta = self.fonte1.render('RESPOSTA INCORRETA...', True,
24                                         AMARELO)
```

```

23     self.mensCorreta = self.fonte2.render('Tente novamente...', True, PRETO
24         )
25
26     # carregando o texto 'CONTINUAR' para o botao continuar
27     self.botaoContinuar = self.fonte3.render('CONTINUAR', True, AZULBB)

```

Código 118: Construtor da classe TelaResultadoDaPergunta.

### 13.2.5.2 Desenhando a tela de resposta de perguntas

O primeiro método que devemos implementar na classe `TelaResultadoDaPergunta` é o `desenharTela` que consiste em:

- Desenhar o `botaoContinuar`;
- Imprimir as mensagens positivas `respCorreta` e `mensCorreta` se a resposta do usuário para a pergunta (`game.respostaUsuario`) for igual à resposta correta (`game.respostaCorreta`) ou, do contrário, imprimir as mensagens negativas `respIncorrecta` e `mensIncorrecta`.



1. Implemente o método `desenharTela(self, game)`, tendo como base o método `desenharTelaBasica(self, game)`, descrito no Código 100.

### 13.2.5.3 Comportamento do botão *continuar*

Após imprimir as mensagens positiva ou negativa para o resultado da pergunta, devemos viabilizar que o usuário saia da tela de resultado de perguntas. Para isso, implementaremos o comportamento do botão continuar. Nossa intenção aqui é que ele se comporte da seguinte forma: quando o jogador colocar o cursor sobre a palavra ‘CONTINUAR’, essa palavra deve adquirir uma cor amarela (como fizemos com o botão de instruções da classe `TelaDeInicio`). Ademais, quando o cursor sair da área correspondente à palavra ‘CONTINUAR’, o ícone deve voltar a ter sua cor original. Por último, se o usuário clicar na palavra ‘CONTINUAR’, devemos redirecionar a tela atual do jogo para outra tela, conforme as seguintes condições:

- Se a resposta do usuário para a pergunta (`game.respostaUsuario`) for igual à resposta correta (`game.respostaCorreta`), devemos redirecionar o usuário para a tela de jogo.
- Do contrário, devemos redirecionar o usuário para a tela de fim de jogo.



1. Implemente o método `comportamentoBotaoContinuar`, tendo como referência o Código 106.

### 13.2.5.4 Incrementando o número de vidas extras do usuário quando ele acerta a pergunta

Por último, faltou implementarmos a grande vantagem de se acertar uma pergunta no jogo: incrementar o número de vidas extras do usuário quando ele acerta a pergunta. Para fazer isso, acrescentaremos uma condição antes do laço de repetição que executa a tela de resultado de perguntas (método `run` da classe `TelaResultadoDaPergunta`). Basicamente, se a resposta do usuário para a pergunta (`game.respostaUsuario`) é igual à resposta correta (`game.respostaCorreta`). Caso isso seja verdadeiro, aumentaremos em 1 o número de vidas extras do usuário. É relevante saber que o número de vidas extras que o usuário possui fica armazenado na variável `game.vidasExtras`.

1. Implemente o incremento do número de vidas extras do usuário quando ele acertar a pergunta. Faça isto dentro do método `run`, antes do laço de repetição.

### 13.2.6 Tela de Jogo

#### 13.2.6.1 Elementos do cenário

Todos os elementos do cenário do jogo (obstáculos, inimigos, tiros, vidas e impulsionadores) são classes derivadas da classe `Cenario`. Essa classe possui três métodos comuns a todos os elementos do cenário do jogo. O primeiro método é o seu construtor, o qual deve ser chamado toda vez que um elemento do cenário deseja ser criado. Esse método possui quatro argumentos:

- `x`: coordenada x em que se deseja posicionar o elemento na tela;
- `y`: coordenada y em que se deseja posicionar o elemento na tela;
- `imagem`: imagem do elemento;
- `vel`: “velocidade” com que o elemento se moverá na tela;

Além de utilizar essas variáveis para inicializar a imagem do elemento do cenário na tela e a sua velocidade de deslocamento, o construtor também é responsável por calcular as variáveis `self.largura` e `self.altura` da imagem do personagem, bem como gerar o retângulo de colisões do elemento<sup>6</sup>. O código correspondente à declaração da classe `Cenario` e o seu construtor está explicitado no Código 119.

```
1 # declaracao de bibliotecas
2 import pygame
3 from Configuracoes import *
4
5 class Cenario:
6     def __init__(self, x, y, imagem, vel):
7         self.x = x
8         self.y = y
9         self.image = imagem
10        self.largura = imagem.get_width()
11        self.altura = imagem.get_height()
12        # retangulo de colisoes
13        self.rect = pygame.Rect(self.x, self.y, self.largura, self.altura)
14        # velocidade de atualizacao horizontal na tela
15        self.vel = vel
```

Código 119: Declaração da classe `Cenario`.

Para movimentar os elementos, a classe `Cenario` possui o método `atualizacaoBasica`, o qual é responsável por atualizar a posição horizontal dos elementos e de seus retângulos de colisão na tela. Basicamente, toda vez que esse método é chamado, a coordenada x do elemento é diminuída de acordo com a variável que define a velocidade de deslocamento na tela do elemento. Essa diminuição na coordenada x ocorre porque os elementos do cenário são gerados no extremo direito da tela, de modo que é necessário diminuir a coordenada x do objeto para que ele se desloque para a esquerda.

Além disso, é importante notar que a coordenada x do elemento só é atualizada enquanto a imagem do objeto ainda está aparecendo na tela, de modo que, se a coordenada x do elemento for menor que o oposto da largura de sua imagem, não há mais razão de fazer essa atualização. Para entender melhor por que a coordenada x do elemento não pode ser menor que o oposto da largura de sua imagem, é preciso perceber que se o topo superior direito começar a ter coordenadas x negativas, ele está completamente fora da tela. Todavia, o *Pygame* utiliza como referência de posição da imagem as coordenadas x e y do topo superior esquerdo da imagem, de forma que, para utilizar a mesma lógica de condição de parada com a referência estando no topo superior direito da imagem, é preciso utilizar a condição de parada sendo quando a coordenada x do elemento for menor que o oposto da largura de sua imagem.

Ademais, as coordenadas x e y do retângulo de colisões são atualizadas de acordo com as coordenadas mais atuais do elemento. O código correspondente à declaração do método `atualizacaoBasica` está explicitado no Código 120.

```
1 # atualizacaoBasica eh o metodo que descreve como o item do cenario tem
2 # sua posicao horizontal atualizada na tela
3 def atualizacaoBasica(self):
4     if self.x > -self.largura:
```

<sup>6</sup>Será abordado mais sobre a necessidade de um retângulo de colisões na Subseção 13.6.

```

5         self.x -= self.vel
6     self.rect.x = self.x
7     self.rect.y = self.y

```

Código 120: Método padrão de atualização da posição horizontal dos elementos do cenário.

Para desenhar na tela os elementos, a classe `Cenario` possui o método `desenhar`, o qual é responsável por adicionar o elemento na janela do jogo, por meio da chamada de um método específico do *Pygame*, denominado `blit`, que requer a imagem e a posição do objeto.

Dentro do método `desenhar`, ainda é possível chamar a função `pygame.draw.rect` com o intuito de desenhar na janela do jogo o retângulo de colisões do elemento, algo que poderá ser interessante para enxergar como as colisões do jogo estão implementadas e auxiliar a encontrar erros nesta parte do desenvolvimento do jogo. O código correspondente à declaração do método `desenhar` está explicitado no Código 121.

```

1 # desenha o item do cenario na tela
2 def desenhar(self, game):
3     game.janela.blit(self.image, (self.x, self.y))
4     # pygame.draw.rect(game.janela, (255, 0, 0), self.rect, 2)

```

Código 121: Método para desenhar na tela os elementos do cenário.

### 13.2.6.2 Aparecimento de elementos do cenário

Para implementar o aparecimento dos elementos do cenário do jogo, utilizaremos o método `criarCenario` da classe `TelaDeJogo`. O aparecimento de elementos no cenário será feito de forma aleatória, com o auxílio da função `random`. A ideia aqui será atribuir para uma variável inteira um número aleatório entre 0 e determinado valor. Então, dependendo do valor que esta variável inteira assumir, por meio da utilização de expressões condicionais, será possível selecionar qual elemento do cenário (obstáculo, vida ou impulsionador) será criado. Um exemplo de implementação de aparecimento de obstáculos está descrito no Código 122.

```

1 # cria itens do cenario na tela
2 def criarCenario(self, game):
3     # geracao de numero aleatorio
4     # game.aparecimentoElementos eh uma constante que vale, inicialmente, 50
5     r = random.randrange(0, game.aparecimentoElementos)
6     # se o numero gerado for menor que 8
7     if r < 8:
8         if len(game.obstaculos) == 0 or (len(game.obstaculos) != 0 and game.
9             obstaculos[-1].x + game.obstaculos[-1].largura + self.tolerancia <
10                LARGURA_DA_TELA):
11             # adiciona um obstaculo no jogo, na posicao (LARGURA_DA_TELA, 562)
12             # com velocidade 5
13             game.obstaculos.append(Obstaculo(LARGURA_DA_TELA, 562, pygame.image
14                 .load(os.path.join('Imagens', 'obstaculo_1_1.png'))), 5))

```

Código 122: Método `criarCenario` com aparecimento aleatório de obstáculos.

Sobre o Código 122 é importante notar que a condição presente na linha 8 verifica dois aspectos:

- A primeira parte da condição verifica a tentativa de se inserir um obstáculo no jogo quando não há nenhum outro obstáculo, isto é, quando `len(game.obstaculos) == 0`. Note que, se isso for verdade, é possível inserir o obstáculo no jogo, sem precisar se preocupar com a sobreposição de imagem deste com a de outro obstáculo.
- A segunda parte da condição verifica a tentativa de se inserir um obstáculo no jogo quando já existem outros obstáculos, isto é, quando `len(game.obstaculos) != 0`. Se isso ocorrer é preciso verificar mais uma condição: se a coordenada x da extremidade direita da imagem do último obstáculo inserido na tela, acrescida de uma pequena tolerância, é menor que a `LARGURA_DA_TELA`. Isso porque, os objetos são criados com coordenada x igual a `LARGURA_DA_TELA`, de modo que se o último obstáculo inserido não estiver completamente na tela, o outro obstáculo será criado em cima dele, sobrepondo as imagens, o que não é bom. Para verificar se a coordenada x da extremidade direita da imagem do último obstáculo inserido na tela, acrescida de uma pequena

tolerância, é menor que a `LARGURA_DA_TELA` utiliza-se a expressão apresentada logo após o `and` presente na linha 8, em que `game.obstaculos[-1]` acessa o último elemento do vetor de obstáculos do jogo, `game.obstaculos[-1].x + game.obstaculos[-1].largura` calcula a coordenada x da extremidade direita da imagem do obstáculo e `self.tolerancia` representa o mínimo espaçamento desejado entre um obstáculo e outro.

Para deixar o jogo mais interessante, você pode implementar o aparecimento de um obstáculo que se move com alta velocidade quando a tela não possui nenhum outro obstáculo, como descrito no Código 123.

```

1 # cria itens do cenario na tela
2 def criarCenario(self, game):
3     # geracao de numero aleatorio
4     # game.aparecimentoElementos eh uma constante que vale, inicialmente, 50
5     r = random.randrange(0, game.aparecimentoElementos)
6     # se o numero gerado for menor que 8
7     if r < 8:
8         if len(game.obstaculos) == 0:
9             # adiciona um obstaculo no jogo, na posicao (LARGURA_DA_TELA, 562)
10            com velocidade 10
11            game.obstaculos.append(Obstaculo(LARGURA_DA_TELA, 562, pygame.image
12               .load(os.path.join('Imagens', 'obstaculo_1_1.png')), 10))
13        elif game.obstaculos[-1].x + game.obstaculos[-1].largura + self.
14            tolerancia < LARGURA_DA_TELA:
15            # adiciona um obstaculo no jogo, na posicao (LARGURA_DA_TELA, 562)
16            com velocidade 5
17            game.obstaculos.append(Obstaculo(LARGURA_DA_TELA, 562, pygame.image
18               .load(os.path.join('Imagens', 'obstaculo_1_1.png')), 5))

```

Código 123: Método `criarCenario` com aparecimento de obstáculo com velocidade alta quando não há outros obstáculos no jogo.

Para que o aparecimento de obstáculos ocorra efetivamente, é preciso chamar o método `criarCenario` no método que efetivamente executa a tela de jogo, isto é, no método `run` da classe `TelaDeJogo`. Não obstante, é preciso lembrar que o método `run` é executado conforme a taxa de quadros por segundo do jogo (FPS) declarada no arquivo `Configuracoes.py`.

Você se lembra que nosso jogo, o FPS é de 60 quadros por segundo o que significa, efetivamente, que o código é executado 60 vezes por segundo? Perceba, todavia, que se chamarmos o método `criarCenario` 60 vezes por segundo, a taxa de aparecimento de objetos será muito grande. Por causa disso, a chamada deste método deve estar dentro de uma condição que só é satisfeita poucas vezes por segundo, como pode ser visto no Código 124.

```

1 def run(self, game):
2     self.tempo = 1
3
4     while game.telaAtual == self.name and not game.usuarioSaiu:
5         if self.tempo % 30 == 0:
6             self.criarCenario(game)
7
8             self.interpretarEventos(game)
9             self.checarColisoes(game)
10            self.atualizar(game)
11            self.desenhar(game)
12
13            self.tempo += 1

```

Código 124: Método `run` da classe `TelaDeJogo` com a chamada do método `criarCenario` sendo realizada duas vezes por segundo.

Note que no Código 124, existe uma variável chamada `self.tempo` que é inicializada com o valor 1 logo antes do laço principal do método começar a ser executado e, a cada iteração do laço, é incrementada em 1. Desta forma, essa variável conta quantas vezes o código é executado e, unindo essa informação com a taxa de FPS do jogo, é possível criar uma lógica para chamar a função `self.criarCenario` apenas duas vezes por segundo. Basta pensar que, se a variável `self.tempo` conta quantas vezes o laço foi executado,

com uma taxa de FPS de 60 quadros por segundo, é possível perceber que, cada vez que `self.tempo` for múltiplo de 60, haverá passado um tempo múltiplo de segundo no jogo. Seguindo essa linha de raciocínio, se `self.tempo` for múltiplo de 30, haverá passado um tempo múltiplo de meio segundo no jogo. Sendo assim, criou-se a condição que verifica se `self.tempo % 30 == 0`, isto é, se o resto da divisão inteira entre `self.tempo` e 30 é nulo (maneira matemática de se verificar se `self.tempo` é múltiplo de 30), e quando ela é satisfeita, chama-se o método `criarCenario`, o que corresponde a chamar o método `criarCenario` a cada meio segundo de jogo.

## CODE like a girl

### 1. Implemente o aparecimento do segundo tipo de obstáculo no método `criarCenario` da classe `TelaDeJogo`.

Para tornar o aparecimento de elementos no jogo ainda mais interessante, podemos aumentar a velocidade com a qual os objetos são inicializados ao longo do tempo, bem como aumentar a taxa de aparecimento de obstáculos.

Para aumentar a taxa de aparecimento de obstáculos, decrementaremos a variável `game.aparecimentoElementos`. Essa variável é o limite superior do intervalo de números possíveis de serem gerados pela função `random` que existe dentro do método `criarCenario`. Desta forma, quanto menor for o intervalo, maior serão as probabilidades de serem gerados números que atendem às condições para aparecimento de elementos do cenário.

Já para aumentar a velocidade com a qual os objetos são inicializados ao longo do tempo, pode-se utilizar uma variável, por exemplo, `game.dvel` que é somada à velocidade inicial dos objetos criados no jogo. Então, no método `run`, incrementa-se o valor desta variável de tempos em tempos.

Sabendo disso, no Código 125 estão descritos os métodos `criarCenario` e `run` com a variável `game.aparecimentoElementos` garantindo o maior aparecimento de obstáculos ao longo do tempo e com a variável `game.dvel` garantindo que os objetos criados tem sua velocidade de deslocamento horizontal aumentadas progressivamente.

```

1 # cria itens do cenario na tela
2 def criarCenario(self, game):
3     # geracao de numero aleatorio
4     # game.aparecimentoElementos eh uma constante que vale, inicialmente, 50
5     r = random.randrange(0, game.aparecimentoElementos)
6     # se o numero gerado for menor que 8
7     if r < 8:
8         if len(game.obstaculos) == 0:
9             # adiciona um obstaculo no jogo, na posicao (LARGURA_DA_TELA, 562)
10            com velocidade 10
11            game.obstaculos.append(Obstaculo(LARGURA_DA_TELA, 562, pygame.image
12                .load(os.path.join('Imagens', 'obstaculo_1_1.png')), 10 + game.
13                dvel))
14
15
16 def run(self, game):
17     self.tempo = 1
18
19     while game.telaAtual == self.name and not game.usuarioSaiu:
20         # aumentar a taxa de aparecimento de elementos do cenario
21         if game.aparecimentoElementos > 25 and self.tempo % 300 == 0:
22             game.aparecimentoElementos -= 1
23         if self.tempo % 1200 == 0:
24             game.dvel += 1
25
26         if self.tempo % 30 == 0:

```

```

27         self.criarCenario(game)
28
29     self.interpretarEventos(game)
30     self.checarColisoes(game)
31     self.atualizar(game)
32     self.desenhar(game)
33
34     self.tempo += 1

```

Código 125: Métodos `criarCenario` e `run` da classe `TelaDeJogo` com maior probabilidade de aparecimento de obstáculos ao longo do tempo e com os obstáculos sendo criados com maior velocidade de deslocamento horizontal ao longo do tempo.

É importante setar um valor inicial para as variáveis `game.aparecimentoElementos` e `game.dvel` a cada novo jogo, com o intuito de garantir que elas assumirão os valores esperados. Por causa disso, na classe `AdministradorDoJogo` do arquivo `main.py`, faremos as inicializações no método `novoJogo`, conforme descrito no Código 126.

```

1 # esse metodo inicializa as constantes do jogo a cada novo jogo
2     def novoJogo(self):
3         self.pontuacao = 0
4         self.aparecimentoElementos = 50
5         self.vidasExtras = 3
6         self.ehInvencivel = False
7         self.respostaCorreta = 0
8         self.respostaUsuario = 0
9         self.tempoDeInvencibilidade = 15
10        self.dvel = 0
11        self.jogador = Jogador(self)
12        self.obstaculos.clear()
13        self.inimigos.clear()
14        self.vidas.clear()
15        self.impulsionadores.clear()
16        self.tiros.clear()
17        self.tirosInimigo.clear()

```

Código 126: Método `novoJogo` da classe `AdministradorDoJogo` com a inicialização das variáveis `game.aparecimentoElementos` e `game.dvel`.



1. Altere os métodos `criarCenario` e `run` da classe `TelaDeJogo`, de modo a incluir o efeito de aumento da velocidade de inicialização dos obstáculos ao longo do tempo, bem como de aumento da taxa de aparecimento de obstáculos ao longo do tempo.
2. Implemente o aparecimento de vidas no método `criarCenario` da classe `TelaDeJogo`. Lembre-se de deixar este aparecimento mais raro que o de obstáculos. Além disso, restrinja o aparecimento a uma única vida ao longo da tela.
3. Implemente o aparecimento de impulsionadores no método `criarCenario` da classe `TelaDeJogo`. Lembre-se de deixar este aparecimento mais raro que o de obstáculos. Além disso, restrinja o aparecimento a um único impulsionador ao longo da tela.

### 13.2.6.3 Efeito de rolagem de paralaxe

Para dar a ideia de movimento e progressão no jogo, além de implementar um algoritmo para o aparecimento, utilizaremos uma técnica de computação gráfica conhecida como **rolagem de paralaxe**. Essa técnica consiste em mover as imagens de plano de fundo (tela de fundo) mais lentamente que imagens em primeiro plano (personagem principal, inimigos, vidas, etc.), criando uma ilusão de profundidade em uma cena 2D e aumentando a sensação de imersão na experiência virtual.

Para fazer isso, trabalhamos com uma imagem de fundo cuja largura é praticamente o dobro da largura da tela de jogo e, sempre, adicionamos à tela duas imagens de fundo: uma iniciando na posição

`(imagemDeFundoX, 0)` da tela e outra iniciando na posição `(imagemDeFundoX2, 0)`, em que `imagemDeFundoX` corresponde à 0 e `imagemDeFundoX2` corresponde à largura da imagem de fundo. Isso pode ser observado no método `desenharTelaBasica` da classe `Tela` (Código 127), o qual desenha os elementos comuns a todas as telas do jogo.

```

1 # desenha os elementos comuns a toda a tela, isto eh, a tela de fundo e o botao
2     de audio
3
4     def desenharTelaBasica(self, game):
5
6         game.janela.blit(self.imagemDeFundo, (self.imagemDeFundoX, 0))
7         game.janela.blit(self.imagemDeFundo, (self.imagemDeFundoX2, 0))
8
9         # carrega a imagem do botao de audio de acordo com o status de audio do
10        jogo
11
12        if game.comAudio:
13            self.botaoSom = pygame.image.load(os.path.join('Imagens', 'audio_ligado.png'))
14
15        else:
16            self.botaoSom = pygame.image.load(os.path.join('Imagens', 'audio_desligado.png'))
17
18        game.janela.blit(self.botaoSom, (1200, 20))

```

Código 127: Método `desenharTelaBásica` da classe `Tela` o qual desenha os elementos comuns a todas as telas do jogo.

Essa adição das imagens de fundo na tela utilizando os parâmetros `imagemDeFundoX` e `imagemDeFundoX2` viabiliza que implementemos a rolagem de paralaxe na classe que representa a tela de jogo (arquivo `TelaDeJogo.py`). Para cumprir esse objetivo, enquanto a tela de jogo estiver sendo executada, faremos as seguintes instruções:

- Decrementar os valores das variáveis `imagemDeFundoX` e `imagemDeFundoX2`, com o intuito de fazer a imagem de fundo se mover para esquerda ao longo do tempo.
- Alterar os valores das coordenadas `imagemDeFundoX` e `imagemDeFundoX2` quando toda a largura da imagem estiver em posições negativas para posições positivas novamente, gerando uma espécie de laço de repetição de imagens de fundo.

No Código 128, a variável `imagemDeFundoX` é alterada de modo a seguir o comportamento descrito acima.

```

1 def atualizar(self, game):
2
3     ...
4
5     # making background move
6     self.imagemDeFundoX -= 2
7
8     if self.imagemDeFundoX < self.imagemDeFundo.get_width() * -1:
9         self.imagemDeFundoX = self.imagemDeFundo.get_width()

```

Código 128: Método `atualizar` da classe `TelaDeJogo` com a variável `self.imagemDeFundoX` sendo alterada de modo a viabilizar o efeito de rolagem paralaxe.



1. Complete o método descrito pelo Código 128, de modo a viabilizar o mesmo comportamento para a variável `imagemDeFundoX2`.

#### 13.2.6.4 Pontuação

É bastante interessante imprimir a pontuação que o usuário está obtendo ao longo da execução do jogo, para que ele possua um *feedback* da sua performance até então. Por causa disso, implementaremos os métodos `computarPontuacao` e `imprimirPontuacao` da `TelaDeJogo` (arquivo `TelaDeJogo.py`).

Para computar a pontuação do usuário, basta que alteremos a variável `game.pontuacao`, incrementando-a ao longo do tempo, conforme descrito no Código 129.

```

1 def computarPontuacao(self, game):
2     game.pontuacao += 1

```

Código 129: Método `computarPontuacao` da classe `TelaDeJogo`.

Para que a pontuação seja computada efetivamente, é preciso chamar o método `computarPontuacao`, de tempos em tempos, no método que efetivamente executa a tela de jogo, isto é, no método `run` da classe `TelaDeJogo`, conforme descrito no Código 130.

```

1 def run(self, game):
2     self.time = 1
3
4     while game.telaAtual == self.name and not game.usuarioSaiu:
5
6         # aumentar a taxa de aparecimento de elementos do cenário
7         if game.aparecimentoElementos > 25 and self.tempo % 300 == 0:
8             game.aparecimentoElementos -= 1
9         # aumentar a velocidade com a qual os elementos do cenário são
10        inicializados
11        if self.tempo % 1200 == 0:
12            game.dvel += 1
13
14        if self.time % 30 == 0:
15            self.computarPontuacao(game)
16            self.criarCenario(game)
17
18            self.interpretarEventos(game)
19            self.checarColisoes(game)
20            self.atualizar(game)
21            self.desenhar(game)
22
23        self.time += 1

```

Código 130: Método `run` da classe `TelaDeJogo` com chamada do método `computarPontuacao`.



**1. Personalize o método `computarPontuacao` e a chamada do método `computarPontuacao` dentro do método `run`, de forma que o incremento da pontuação do seu jogo e a taxa com que a pontuação é incrementada sejam realizadas da forma que você deseja.**

Note que o método `computarPontuacao` e a sua chamada no método `run` apenas incrementam a pontuação, não mostrando-a na tela. Para viabilizar a visualização da pontuação na tela de jogo, implementaremos o método `imprimirPontuacao`.

Neste método, carregaremos duas variáveis de texto, uma com a palavra “*SCORE*” (pontuação, em inglês), e outra com o valor da pontuação, a qual, como já vimos, está armazenada na variável `game.pontuacao`. Posteriormente, basta desenhar estes textos na tela, com o auxílio do método `blit`, conforme pode ser visto no Código 131.

```

1 def imprimirPontuacao(self, game):
2     self.pontuacao = self.fonte1.render("SCORE: ", True, ROXO)
3     self.pontuacaoNum = self.fonte1.render(str(game.pontuacao), True, ROXO)
4     game.janela.blit(self.pontuacao, (35, ALTURA_DA_TELA - 50))
5     game.janela.blit(self.pontuacaoNum, (140, ALTURA_DA_TELA - 50))

```

Código 131: Método `imprimirPontuacao` da classe `TelaDeJogo`.

### 13.3 Jogador

Nossa personagem, nomeado de `jogador`, é representada como um objeto derivado da classe `Jogador()` declarado dentro do método `novoJogo()` da classe `AdministradorDoJogo` (`main.py`):

```

1     self.jogador = Jogador(self)

```

A classe `Jogador()` no arquivo `Jogador.py` possui os atributos e métodos apresentados abaixo:

```
1 class Jogador():
2     def __init__(self, game):
3         self.x = X_CHAO
4         self.y = Y_CHAO
5         self.carregarImagenPersonagem(game)
6         self.largura = self.image.get_width()
7         self.altura = self.image.get_height()
8         self.rect = pygame.Rect(self.x, self.y, self.largura, self.altura)
9             # retangulo de colisoes
10        self.rect.center = (self.x + self.largura/2, self.y + self.altura/2)
11        self.pos = vec(self.x + self.largura/2, self.y + self.altura/2)
12        self.vel = vec(0.0, 0.0)
13        self.acc = vec(0.0, 0.0)
14
15        # carrega a imagem da personagem de acordo com a escolha do usuario
16        def carregarImagenPersonagem(self, game):
17            pass
18        # esse metodo atualiza as posicoes do jogador para que ele pule
19        def pular(self, game):
20            pass
21        # esse metodo faz carregar o tiro do jogador na tela
22        def atirar(self, game):
23            pass
24        # esse metodo atualiza as posicoes do jogador
25        def atualizar(self, game):
26            pass
27        # desenha o jogador na tela com a imagem correspondente ao seu estado
28        atual
29        def desenhar(self, game, tela):
30            pass
```

O `__init__` na linha 2 é uma palavra reservada de Python que faz o papel de construtor definido como o método que será chamado assim que o objeto for criado. Nele definimos os atributos da classe e o que desejamos que seja feito quando criarmos um objeto a partir dessa classe. Em `Jogador()`, por exemplo, inicializamos os atributos `x`, `y`, `largura`, `altura`, `rect`, `rect.center`, `pos`, `vel` e `acc` e chamamos o método `carregarImagenPersonagem()`.

Com esse construtor obtemos e definimos alguns valores que irão ser importantes para objetos criados a partir de `Personagem` e através do método `carregarImagenPersonagem()` importamos as imagens, que serão guardadas no vetor `imagens` e que usaremos para os movimentos de nossa personagem.

### 13.3.1 Movimentos do jogador

Nosso jogador possui basicamente três movimentos: pular, atirar e andar no mesmo lugar. Podemos criar muitos outros comandos, como voar, abaixar, pulo duplo, entre outros, mas vamos ensinar os conceitos necessários para fazer esses três movimentos básicos a partir dos quais vocês poderão desenvolver muitos outros.

#### 13.3.1.1 Identificação das teclas pressionadas

Os movimentos do jogador inicializam-se pela identificação das teclas pressionadas. No arquivo `TelaDeJogo.py`, temos a classe `TelaDeJogo` cujo método `checlarComportamentoJogador()` verifica os comandos enviados pelo jogador.

```
1 def checlarComportamentoJogador(self, game, evento):
2     # verificar se o usuario pediu para o jogador fazer algum comando (
3         # atirar ou pular)
4     if evento != [] and evento.type == pygame.KEYDOWN: #verificar se ha algo
5         na fila de eventos e se ha teclas pressionadas
6         if evento.key == pygame.K_UP:
7             game.jogador.pular(game)
8         elif evento.key == pygame.K_SPACE:
9             game.jogador.atirar(game)
```

Esse método recebe como parâmetros a própria classe `TelaDeJogo`, o objeto `game` criado a partir da classe `AdministradorDoJogo` e um evento que em `Pygame` pode ser qualquer interferência externa ao jogo como pressionarmos uma tecla, mexermos o mouse ou fecharmos o jogo. Esses eventos ficam guardados em uma fila e pelo método `interpretarEventos()` da classe `TelaDeJogo()` pegamos cada um dos eventos com `pygame.event.get()` e checamos qual ação o jogador quer realizar:

```

1 def interpretarEventos(self, game):
2     game.clock.tick(game.fps)
3
4     for evento in pygame.event.get():
5         pos = pygame.mouse.get_pos()
6
7         # checa se o usuario quer sair do jogo
8         self.comportamentoBotaoDeSair(game, evento)
9
10        # checa se o usuario quer mover o jogador
11        self.checarComportamentoJogador(game, evento)
12
13        # checa se o usuario quer tirar o som
14        self.comportamentoBotaoDeAudio(game, evento, pos)
```

Para todos os eventos recebidos verificamos primeiramente se ele não é vazio `evento !=`

e se ele é do tipo `KEYDOWN` (tecla pressionada). Se ele for uma tecla pressionada, verificamos se essa tecla é `K_UP` (seta para cima) ou `K_SPACE` (barra de espaço), para cada uma chamamos o método do objeto `jogador` correspondente (`pular()` para `K_UP` e `atirar()` para `K_SPACE`).

### 13.3.1.2 Andar

Para o movimento de andar, dado que não será nossa personagem que se deslocará, mas sim o plano de fundo, vamos implementar apenas trocas de imagens para gerar a ilusão de que ela está caminhando. Cada personagem possui 3 *frames* de movimento como os apresentados na figura 21:



(a) Frame 0.



(b) Frame 1.



(c) Frame 2.

Figura 21: Sequência de *frames* que forma a caminhada da personagem.

A intercalação entre os *frames* deve ser feita na sequência 0-1-2-1 em intervalos aproximadamente constantes para que o movimento pareça mais natural. Vamos implementar essa alternância no método `desenhar()` da classe `Jogador`, para isso vamos utilizar o atributo `time` da classe `TelaDeJogo` que é passada como parâmetro para o método `desenhar()` como `tela`. Para acessar `time` dentro do método `desenhar()` basta usarmos `tela.time`. Esta variável funciona como um contador e é incrementada a cada execução do método `run()` da classe `TelaDeJogo` que fica em loop durante o jogo até entrar uma tela nova ou o usuário sair, assim ela pode nos fornecer uma aproximação do tempo decorrido no jogo que nos será útil para estimar os intervalos para as trocas de imagens.

Dado um intervalo de tempo  $\Delta t$ , durante o primeiro quarto de  $\Delta t$  a imagem da personagem deve ser o frame 0, no próximo quarto o frame 1, em seguida o frame 2 e novamente o frame 1 e assim durante todo o tempo. Pensando no decorrer do jogo, podemos dividir o tempo em intervalos de duração  $\Delta t$  e para cada quarto desse intervalo definir qual imagem a personagem vai possuir. Uma boa solução matemática para esse problema é usando módulo. Se pensarmos no tempo do jogo módulo  $\Delta t$  ( $\Delta t$  inteiro) estamos realizando uma partição do tempo e dado que os possíveis valores de  $(\text{tempo} \bmod \Delta t)$  são  $(0, 1, \dots, \Delta t - 1)$  podemos verificar para cada valor obtido em qual quarto de  $\Delta t$  estamos, por exemplo para  $\Delta t = 12$  temos a situação:

$$\begin{aligned} (\text{tempo} \bmod 12) < 3 &\Rightarrow \text{frame 0} \\ 3 \leq (\text{tempo} \bmod 12) < 6 &\Rightarrow \text{frame 1} \\ 6 \leq (\text{tempo} \bmod 12) < 9 &\Rightarrow \text{frame 2} \\ 9 \leq (\text{tempo} \bmod 12) &< 12 \Rightarrow \text{frame 1} \end{aligned}$$

A imagem da personagem que aparecerá no jogo é armazenada na variável `imagem` da classe `Jogador`. Ela é declarada no método `carregarImagenPersonagem` que também carrega das pasta `Imagens` os três frames mostrados na figura 21 e os três equivalentes para o modo invencível. Para facilitar a manipulação dessas imagens podemos guarda-las em um vetor em que o índice 0 terá o frame 0, o índice 1 o frame 1 e o índice 3 o frame 3, como para o modo normal e o modo invencível as imagens são diferentes, primeiramente

devemos verificar se a personagem está no modo invencível (`game.ehInvencivel`), se falso criar o vetor com as imagens padrões, caso verdadeiro com as imagens do modo invencível. Em seguida devemos fazer a verificação mostrada acima na variável `tempo` e associar ao atributo `imagem` o frame correto, não se esquecendo de verificar se a personagem está no chão, pois caso ela esteja pulando, não faz sentido ela estar se movimentando como se estivesse andando.

## CODE like a girl

1. **Implemente as transições de imagens necessárias para criar o efeito de que a personagem está andando. Dica 1: O operador % é usado em Python para realizar a operação de módulo, por exemplo  $13\%3 = 1$ . Dica 2: Para definir qual valor de  $\Delta t$  vai usar, dê preferência para múltiplos de 4 e comece com o exemplo mostrado ( $\Delta t = 12$ ), teste e em seguida mude para outros valores até obter o efeito que mais lhe agrade.**

### 13.4 Pular

O método `pular()` é responsável por atualizar a posição vertical da personagem criando o pulo. Para construção desse método pensemos primeiramente como é dada a ação de pular. Primeiramente devemos estar sobre uma superfície, se estivermos sobre uma realizamos um impulso para cima a partir do qual ganhamos uma velocidade que nos move para cima e vamos desacelerando, devido a ação da gravidade, até começarmos a cair.

Nossa personagem segue os mesmos princípios. Por isso devemos implementar no método pulo i) uma verificação se a personagem está no chão, ii) dizermos qual a velocidade inicial do pulo e iii) alterarmos sua aceleração para a aceleração da gravidade.

Além disso, precisamos atualizar com bastante frequência a posição e a velocidade da personagem, o que iremos fazer no método `atualizar()` que é chamado de tempos em tempos pelo objeto `tela` pertencente a classe `TelaDeJogo()`. Para recriarmos a equação de movimento no jogo, dado que não é possível atualizar de forma continua a posição, precisamos criar uma variável auxiliar para representar o intervalo de tempo entre cada atualização, podemos dar a ela o valor 1. A cada intervalo a velocidade do jogador vai ser acrescida da aceleração vezes o intervalo de tempo definido (atenção ao sentido dessas grandezas) e a posição da jogadora vai ser somada ao quando ele se deslocou desde de a última atualização (velocidade vezes o intervalo de tempo). Em algum momento, devido a aceleração da gravidade, o jogar ira começar a cair, para que ele não ultrapasse o chão, precisamos verificar cada nova posição vertical, “desligar” a gravidade e zerar a velocidade quando o jogador alcançar o chão.

## CODE like a girl

1. **Implemente o método pular. Dica: O atributo `vel` refere-se a velocidade do jogador, se escrevermos `vel.y` estaremos nos referindo a sua velocidade vertical. O mesmo vale para `pos` (posição) e `acc` (aceleração). Sempre que necessário, consulte o capítulo 12.**

Para aumentar o desafio do nosso jogo cada obstáculo ou inimigo aparece com uma velocidade diferente que vai aumentando com o passar do jogo, por isso precisamos fazer uma adição no método criado para que o personagem consiga pular por todos os obstáculos. Antes de mais nada precisamos fazer a análise física desse movimento (SIM! Física! Jogos em geral envolvem muita física e matemática).

Apesar de só alterarmos a posição vertical de nosso personagem, dado que os obstáculo estão se aproximando com uma velocidade  $v_o$ , podemos mudar o referencial para nosso personagem, para facilitar a análise, e considerar que nosso jogar se movimenta na horizontal com a mesma velocidade do obstáculo, mas em sentido contrário, assim será equivalente a situação de o obstáculo estar parado e estarmos saltando sobre ele, como ilustrado na figura 22.

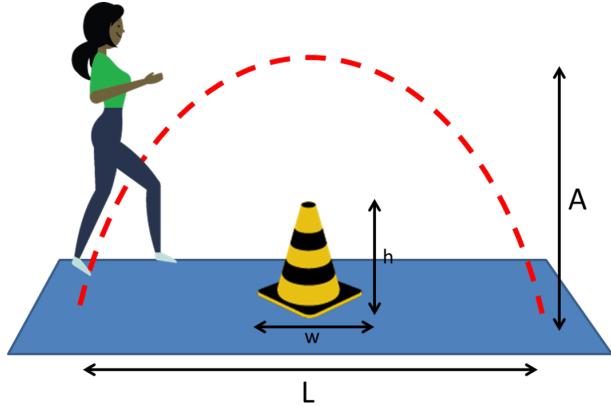


Figura 22: Movimento realizado pela personagem ao saltar sobre um obstáculo, no referencial do obstáculo.

Para começarmos essa análise temos que ter em mente que a largura e a altura desses obstáculos serão constantes e iguais a  $w$  e  $h$ , respectivamente, assim, a altura e a distância que a personagem atingir devem ser suficientes para ultrapassar o obstáculo. De cinemática temos que a altura e a distância horizontal máximas atingidas em uma trajetória parabólica são:

$$A_{max} = \frac{v_v^2}{2a} \quad (1)$$

$$L_{max} = \frac{2v_v * v_h}{a} \quad (2)$$

Em que  $v_v$  é a velocidade vertical inicial da nossa personagem,  $v_h$  é a velocidade horizontal que ela vai ter no referencial do obstáculo que será igual a  $-v_{obstaculo}$  e  $a$  é a aceleração vertical que a personagem sofre.

Para facilitar nossos cálculos vamos assumir que basta  $A_{max}$  e  $L_{max}$  ser maior que a largura e a altura do obstáculo do maior dos obstáculos mais uma margem de segurança que estimaremos em 50%. Assim:

$$A_{max} = \frac{v_v^2}{2a} \geq 1.5h$$

$$L_{max} = \frac{2v_v v_h}{a} \geq 1.5w$$

Considerando o caso de igualdade:

$$A_{max} = \frac{v_v^2}{2a} = 1.5h \quad (3)$$

$$\begin{aligned} L_{max} &= \frac{2v_v v_h}{a} = 1.5w \Rightarrow \\ \frac{v_v}{a} &= \frac{1.5w}{2v_h} \end{aligned} \quad (4)$$

Substituindo 4 em 3:

$$\begin{aligned} \frac{v_v}{2} \cdot \frac{v_v}{a} &= 1.5h \\ \frac{v_v}{2} \cdot \frac{1.5w}{2v_h} &= 1.5h \Rightarrow \\ v_v &= \frac{4v_h h}{w} \end{aligned} \quad (5)$$

Voltando 5 em 4:

$$\begin{aligned} a &= \frac{2v_h v_v}{1.5w} \\ a &= \frac{2v_h}{1.5w} \cdot \frac{4v_h h}{w} \Rightarrow \\ a &= \frac{8v_h^2 h}{1.5w^2} \end{aligned} \quad (6)$$

Assim a velocidade vertical da personagem precisará ser aproximadamente  $\frac{4v_h h}{w}$  e a aceleração  $\frac{8v_h^2 h}{1.5w^2}$ , em que  $v_h$  é  $-v_{obstaculo}$ .

Aproximando o obstáculo para um quadrado de 100x100px:

$$v_v \approx -4v_{obstaculo}$$

$$a \approx 0.053v_{obstaculo}^2$$

Assim para fazermos as correções necessárias para variarmos a velocidade do jogo e nossa personagem ainda conseguir pular os obstáculos, temos que adicionar no método pular um trecho que verifique se há obstáculos aproximando e ajustar a aceleração e a velocidade da personagem de acordo com as relações encontradas.

Mais para frente veremos que os inimigos e obstáculos criados ficam em vetores chamados `obstaculos` e `inimigos` dentro do objeto `game`, assim podemos usar a função `len(vetor)` que retorna o tamanho de um vetor para verificar se há obstáculos/inimigos nesses vetores e caso hajam obter a velocidade do obstáculo de posição zero (mais próximo de nós) para calcular as correções.

Além dessa correção, o pulo com a personagem parada é pouco natural, por isso precisamos atualizar sua imagem para que fique mais próximo de como é um pulo na vida real. Para isso, no mesmo método em que implementamos o efeito de andar, vamos adicionar verificações para que de acordo com a altura em que a personagem está sua imagem seja o frame 0 (para quando ela ainda estiver no chão), o frame 1 (para as posições iniciais do salto) ou o frame 2 (para quando ela não estiver tão baixa na posição vertical).



1. Implemente as correções no método pular e adicione as transições de imagem para o pulo. Dica 1: Lembre-se que no Pygame o eixo y está voltado para baixo, assim, quanto maior o valor da posição em y da personagem, mais próxima do chão ela está.

#### 13.4.1 Tiro

O método `atirar()` é invocado da mesma forma que o método `pular()` pelo método `checkComportamentoJogador()` da classe `TelaDoJogo`, só mudando a tecla que o aciona. Dado que podemos ter vários tiros consecutivos, para cada um deles criaremos um objeto pertencente a classe `Tiro` que serão armazenados no vetor `tiros` da classe `AdministradorDoJogo`.

```
1 self.tiros = []
```

Assim, no método `atirar()` da classe `Jogador` precisaremos apenas implementar um comando que adicione novos objetos, no entanto, para adicionar esses novos objetos, precisamos primeiramente implementar a classe `Tiro`. No arquivo `Tiro.py` já temos um esqueleto do que será a classe filho.

```
1 class Tiro(Cenario):
2     def __init__(self, x, y, tipo, vel):
3         self.carregarImagenTiro(tipo)
4         super().__init__(x, y, self.imagem, vel)
5
6         # carregar a imagem do tiro
7     def carregarImagenTiro(self, tipo):
8         self.imagem = pygame.image.load(os.path.join('Imagens', 'tiro.
9             png'))
10
11         # Inverter a imagem para o tiro do inimigo
12         if tipo == 'I':
13             self.imagem = pygame.transform.flip(self.imagem, 1, 0)
14
15         # atualiza a posicao do tiro apos disparado
16         def atualizar(self, tela):
17             self.atualizacaoBasica()
18
19         # verifica as colisoes do tiro com o inimigo
20         def checarColisoes(self, game, tiro):
21             pass
```

Observem na linha 1 que a classe `Tiro` é uma classe filha da classe `Cenario`, portanto ele já possui os métodos `atualizacaoBasica()` e `desenhar()`, além dos atributos `x`, `y`, `image`, `largura`, `altura`, `rect` e `vel` que inicializamos com o contrutor de `Cenario` na linha 4 com o `super().__init__(x, y, self.image, vel)`.

Não será necessário alterar o método `carregarImagenTiro()` - que carrega a imagem do tiro e a inverte horizontalmente quando o tiro vier do inimigo - nem `atualizar()` - que dado a velocidade `vel` do tiro e a coordenada `(x,y)` a cada chamada decrementa `vel` de `x`. Portanto, trabalharemos apenas com o `checarColisoes` e com as manipulações necessárias para esse objeto.

#### 13.4.1.1 Disparo

Quando o usuário envia o comando de tiro, como mencionado, o método `atirar()` da classe `Jogador` adiciona um novo objeto pertencente a classe `Tiro` ao vetor `tiros`. Para isso vamos usar o método `append` próprio do *Python*, cuja função é adicionar um elemento qualquer a uma lista sem alterar os que já estão nela. A sintaxe para usar o `append` é:

```
1 nomeDaLista.append(elementoQueSerahAdicionado)
```

Em nosso caso, a lista é `tiros` que por pertencer ao objeto `game` deve ser acessada como `game.tiros` e o elemento que adicionaremos é um objeto da classe `Tiro` que deve ser criado por `Tiro(valorDeX, valorDeY, tipoDeTiro, velocidadeDoTiro)`. Para os tiros do jogador queremos que ele saí exatamente da frente da personagem e vamos definir a sua origem em `y` do pé dessa, facilitando na hora de acertar os inimigos, ademais o tiro da personagem é do tipo A e vamos começar com a velocidade `-10` que devido a mudança de velocidade ao longo do jogo deve ser acrescida em módulo da variável `dvel`, pertencente ao objeto `game`, que guarda o aumento da velocidade.



1. Implemente o método `atirar()` da classe `Jogador`. Dica : Lembre-se de como usar o método `append` e criar um objeto da classe `Tiro`.

```
1 game.tiros.append(Tiro(self.x + self.largura, 0.93*self.pos.y, 'A', -10  
- game.dvel))
```

#### 13.4.1.2 Movimento do tiro

O movimento do tiro será exclusivamente horizontal a uma velocidade constante, portanto usaremos o método `atualizacaoBasica` herdado de `Cenario`. Apesar do movimento ser simples, precisamos realizar uma verificação a mais já implementada no método `atualizar` da classe `TelaDeJogo`.

```
1 for tiro in game.tiros:  
2     tiro.atualizar(game)  
3     if tiro.x > LARGURA_DA_TELA:  
4         game.tiros.pop(game.tiros.index(tiro))
```

Esse método verifica se a posição do tiro é maior que a largura da tela e caso seja, ele remove (método `pop`) esse objeto da lista `tiros`.

### 13.5 Inimigos

Os inimigos, objetos pertencentes a classe `Inimigo()`, são personagens que aparecem a cada 1800 contagens da variável `time` e são apresentados no modo batalha. Esses, diferentemente dos obstáculos, apresentam duas dificuldades a mais: se movimentam também na vertical e atiram, assim como nossa personagem principal.

#### 13.5.1 Movimentos do inimigo

Assim como a classe `Tiro`, `Inimigo` também é uma classe filha da classe `Cenario`, portanto ele possui os mesmos métodos e atributos herdados da classe pai, mas diferentemente da classe `Tiro`, os inimigos se movimentarão, não apenas horizontalmente, mas saltando também e possui atributos a mais como `vida` e `vely`.

No arquivo `Inimigo.py` temos o esboço da classe `Inimigo`:

```

1   class Inimigo(Cenario):
2       def __init__(self, x, y, imagem, vel, vida, tipo):
3           super().__init__(x, y, imagem, vel)
4           self.vida = vida
5           self.tempo = pygame.time.get_ticks()
6           self.velY = -2.5 * vel
7           self.tipo = tipo
8
9           # metodo responsavel pela movimentacao do inimigo na tela
10          def atualizar(self, game):
11              pass
12
13          # verifica as colisoes do personagem com o inimigo
14          def checarColisoes(self, game):
15              pass

```

Para implementarmos o método `atualizar()` primeiramente devemos chamar o método `atualizacaoBasica()` que ficará responsável pelo movimento horizontal e para o movimento vertical vamos implementar a mesma equação de movimento utilizada na classe `Jogador` usando ao invés da velocidade horizontal do objeto que está se aproximando, a própria velocidade horizontal do inimigo que é fixa, lembrando-se de realizar as verificações nas posições do objeto para saber se ele está no chão e pode pular ou se na queda do salto ele já chegou no chão.



1. Implemente o método `atualizar()` da classe `Inimigo`. Dica: Lembre-se de verificar se são válidas as posições dos inimigos e de quando o inimigo voltar ao chão, redefinir `velY` como `-2.5 * self.vel`.

### 13.5.2 Tiros do inimigo

O segundo ponto que torna os inimigos um desafio são seus tiros que só nos permite saltar sobre eles ou anular com nossos próprios tiros. Para que esses não sejam igualmente espaços no tempo, o que os tornariam previsíveis, podemos criar uma variável dentro do método `atualizar()` que recebe um valor aleatório entre  $a$  e  $b$  usando a função `random.randrange(a,b)` e quando ocorrer um valor específico desse intervalo criamos um tiro do inimigo.

Assim, como ocorre com os tiros da personagem principal, todos os tiros dos inimigos são armazenados em uma fila do objeto `game` chamada `tirosInimigo`. De forma análoga ao que fizemos para adicionar um tiro para o jogador, vamos fazer para o inimigo, exceto pelo tipo de tiro que para os inimigos será 'T' e da velocidade que será no sentido contrário ao tiro da protagonista, ainda considerando a correção usando `dvel`.

Assim como ocorre com os tiros do jogador, também verificamos a cada atualização a posição dos tiros dos inimigos e caso esses saíram da tela, chamamos a função `pop` para remove-los.

```

1   for tiroInimigo in game.tirosInimigo:
2       tiroInimigo.atualizar(game)
3       if tiroInimigo.x < - tiroInimigo.largura - 20:
4           game.tirosInimigo.pop(game.tirosInimigo.index(tiroInimigo))

```



1. Adicione o tiro aleatório no método `atualizar()`. Dica : Vá ajustando os valores de  $a$  e  $b$  até obter uma frequência de tiros desafiadora, mas não impossível e, caso não esteja, adicione a linha `import random` ao cabeçalho do arquivo `Inimigo.py` para poder usar a função `random.randrange(a,b)`.

## 13.6 Colisões

Os métodos e funções de colisão são aqueles que recebem imagens ou vetores de imagens e identificam quando essas se sobrepõem, permitindo assim que seja criada uma condição quando esse evento ocorrer. No nosso jogo usaremos uma combinação do método rect.spritecollide e de classes do módulo sprite.collide, como pode ser visto no exemplo a seguir:

```
1 if self.rect.colliderect(objeto1):
2     collisions = pygame.sprite.spritecollide(objeto1, objeto2, False)
3     callback = pygame.sprite.collide_mask
4     collide = pygame.sprite.spritecollideany(objeto1, collisions,
callback)
```

- rect.spritecollide: com a função pygame.rect que é chamada para todas as imagens do jogo, é construído um retângulo envolvente à elas. Na interação do objeto1 e objeto2, ao usarmos objeto1.rect.spritecollide(objeto2) é checado se os retângulos envolventes de suas imagens se sobrepueram.
- pygame.sprite.spritecollide(objeto1, objeto2, False): o objeto1 deve ter como atributo uma imagem e o objeto2 deve ter um vetor de imagem, é retornado, então, uma lista contendo todas as imagens do objeto2 que colidiram com a imagem do objeto1.
- pygame.sprite.spritecollide.collide\_mask: método de detecção de colisão à ser passado para uma função que verifica se essa ocorreu. Esse método foi escolhido porque ele verifica a colisão analisando o contorno da imagem o que traz mais realidade ao jogo do que realizar a colisão entre os retângulos envoltórios.
- pygame.sprite.spritecollideany: verifica se houve colisão usando o método passado, que no caso é o collide\_mask, retornando o um booleano True ou False. Perceba que no nosso exemplo essa verificação é feita não mais com todas as imagens, mas sim com aquelas que já apresentaram a colisão de seus retângulos envolventes, isso é feito porque o método de colisões com collide\_mask é mais dispendioso do que o colliderect.

Agora é sua vez. Implemente as colisões listadas abaixo:

### 13.6.1 Colisões com obstáculos

Ao jogador entrar em contato com o obstáculo, se o jogador não tiver o poder de invencibilidade ou vidas extras, ele morre. Caso contrário, o obstáculo deve desaparecer. Se o jogador possuir vidas extras, mas não tem o poder de invencibilidade, ele deve perder uma de suas vidas. Se o jogador morrer deve tocar por um tempo a música de fim de jogo e deve aparecer a 'Tela de Fim'. Todas essas ações devem ser implementadas no arquivo Obstaculo.py

Dica: As imagens do objeto estão em um vetor, para remover a imagem desse vetor que colidiu use o método pop, passando como parâmetro o seu índice no vetor, o que, por sua vez, será feito utilizando o método index. Por exemplo, se temos um objeto da classe livros que possui um vetor de imagens e que foi declarado dentro de outro objeto chamado escola. Ao implementarmos a função de colisão, essa deverá receber como argumentos self (que é o objeto da classe livro que está sendo trabalhado) e biblioteca. Desse modo, para excluir a imagem com a qual foi verificada a colisão constrói-se uma função do tipo: checarColisoes(self, biblioteca) e dentro dela o trecho que irá excluir a imagem de um vetor de imagens, com a qual foi checada a colisão, será: escola.livros.pop(escola.livros.index(self))

### 13.6.2 Colisões com vidas

Ao jogador entrar em contato com a vida essa deve sumir (assim como os obstáculos ela tem um vetor de imagens), deve aparecer a 'Tela de Perguntas' e a quantidade de vidas extra do jogador deve ser aumentada em 1 unidade. Essas ações devem ser implementada no arquivo Vida.py.

### 13.6.3 Colisões com impulsionadores

Ao jogador entrar em contato com o impulsionador, esse deve sumir (também possui um vetor de imagens) e a variável de invencibilidade do jogador deve se tornar verdadeira (= True). A invencibilidade tem um prazo para durar então, após terminado esse tempo, ela deve voltar a ser falsa (= False) e enquanto ela ainda é verdadeira a música do jogo será diferente. Essas ações devem ser implementadas no arquivo Impulsionador.py.

### 13.6.4 Colisões com inimigos

Ao jogador entrar em contato com o inimigo (também possui um vetor de imagens), se não tiver o poder de invencibilidade, o jogador morre. Caso isso aconteça, irá ocorrer a mesma série de eventos descritos na parte de colisões com obstáculos. Essas ações devem ser implementadas no arquivo Inimigo.py.

### 13.6.5 Colisões com tiros

Para implementarmos um método para checar as colisões com os tiros (`checkColisoes()`) precisamos pensar nas duas possíveis situações de colisão do tiro: tiro-tiro e tiro-inimigo.

Analisando a primeira situação, sabemos que não necessariamente o primeiro tiro da fila de tiro vai colidir com o primeiro tiro da fila de tiros do Inimigo, por essa razão para todos os objetos de `tiros` precisamos verificar se houve colisão com os objetos de `tirosInimigo` ou não. Uma boa função para passar por todos esses elementos é a `for .. in ..` que obedece a seguinte sintaxe:

```
1 # a variavelAuxiliar a cada iteracao recebe um valor da listaDeElementos
2 for variavelAuxiliar in listaDeElementos:
```

Assim podemos usar `for .. in ..` para dado um tiro, percorrer a lista de inimigos e verificar se o tiro acertou algum inimigo, lembrando que tanto os tiros da protagonista, quanto os tiros dos inimigos utilizaram esse mesmo método, por tanto precisamos diferenciar quando um tiro do inimigo está acertando o próprio inimigo e quando é da protagonista. Um método simples de fazer isso é verificando a velocidade horizontal dos objetos dado que os tiros da nossa personagem vão ter velocidade negativa (por causa de como é implementado o método `atualizacaoBasica()`) e os tiros dos inimigos vão ter velocidades positivas.

Após verificarmos se o tiro acertou um inimigo, devemos fazer o mesmo com os tiros dos inimigos, dessa vez usando o `for .. in ..` para percorrer a fila `tiroInimigo` e verificando se houve colisão entre o tiro analisado e os tiros da fila, lembrando da mesma verificação anterior para saber se o tiro partiu da protagonista ou do inimigo.



1. Implemente o método `checkColisoes` da classe `Tiro`. Dica 1: Lembre-se de que quando o jogador está no modo invencível ele não recebe dados por ser atingido, verifique isso antes de decrementar o atributo `vidasExtras` ou encerrar o jogo.

## 13.7 Sonorização

Para adicionarmos a sonorização do jogo, usamos a classe `AdministradorDeAudio`. Nela definimos o número de canais a serem utilizados, o método para tocar a música de fundo e os efeitos sonoros, e os métodos para mutar e desmutar o jogo.

```
1 class AdministradorDeAudio():
2     def __init__(self):
3         pygame.mixer.init()
4         pygame.mixer.set_num_channels(2)
```

A função `pygame.mixer.init()` inicializa o módulo do mixer para carregamento e reprodução de som. Os argumentos padrão podem ser substituídos para fornecer mixagem de áudio específica, no entanto não será necessário para o nosso código.

A função `pygame.mixer.set_num_channels()` define o número de canais disponíveis para o mixer, permitindo que sons diferentes sejam reproduzidos simultaneamente. Como parâmetro, passamos o número de canais desejados, sendo no máximo 8. No caso escolhemos 2: um para a música de fundo e outro para os efeitos sonoros.

### 13.7.1 Deixando o jogo mudo

O método descrito a seguir é responsável por tirar os sons do jogo (torná-lo mudo).

```
1 # deixar o som mudo
2 def deixarSomMudo(self, game):
3     pygame.mixer.Channel(0).set_volume(0)
4     pygame.mixer.Channel(1).set_volume(0)
5     game.comAudio = False
```

Nesse método, apenas zeramos o volume dos canais utilizados por meio da função `pygame.mixer.Channel().set_volume()`. O primeiro parâmetro é o canal e o segundo é a porcentagem do volume, valor que varia de 0 a 1. Por último, a variável booleana `game.comAudio = False` armazena a informação de que o jogo está sem áudio.



1. Implemente o método `deixarSomTocar` da classe `AdministradorDeAudio`, responsável por “desmutar” o jogo.

### 13.7.2 Tocando efeitos sonoros e a música de fundo

O método descrito abaixo é responsável por implementar os efeitos sonoros.

```

1 def tocarEfeitoSonoro(self, efeito, game):
2     # lembrar de checar se o jogo esta com audio ou nao
3     # usar 1 channel para a musica de fundo e outro para os efeitos sonoros
4     if (game.comAudio):
5         pygame.mixer.Channel(0).set_volume(0.3)
6         pygame.mixer.Channel(1).play(pygame.mixer.Sound(efeito), 0)
7         pygame.mixer.Channel(0).set_volume(1)

```

Esse método é dividido em três ações: diminui o volume da música de fundo, reproduz o efeito sonoro e aumenta o volume da música de fundo.

Como visto, a função `pygame.mixer.Channel().set_volume()` determina o volume de determinado canal. No caso, inicialmente reduzimos o volume do canal 0 para 0.3, ou seja, 30% do volume máximo.

A função `pygame.mixer.Channel().play(pygame.mixer.Sound())` é utilizada para tocar determinado som em determinado canal. A função `pygame.mixer.Channel().play()` tem por parâmetros o som que será reproduzido, o qual é definido pela função `pygame.mixer.Sound()`, e o número de loops, vezes que o som será repetido depois de tocar uma vez. Ou seja, se o número de loops for 3, o som será reproduzido 4 vezes (na primeira vez e mais três). Se o loop for -1, a reprodução será repetida indefinidamente.

Em seguida, retornamos o volume da música de fundo para o volume máximo.



1. Implemente o método `tocarMusicaDeFundo` da classe `AdministradorDeAudio`.

### 13.7.3 Adicionando efeitos sonoros e a música de fundo

Os efeitos sonoros serão utilizados nas colisões da personagem com os objetos do jogo, portanto, o método `tocarEfeitoSonoro` será utilizado no impulsionador, no inimigo, no obstáculo e no tiro. Ou seja, todas as funções que tiverem colisões, adicionaremos um efeito sonoro.

Como exemplo, vamos ver a implementação do efeito sonoro do impulsionador.

```

1 def checarColisoes(self, game):
2     if self.rect.collidelist(game.jogador):
3         colisoes = pygame.sprite.spritecollide(game.jogador, game.
4             impulsionadores, False)
5         callback = pygame.sprite.collide_mask
6         colisao = pygame.sprite.spritecollideany(game.jogador, colisoes,
7             callback)
8         if colisao:
9             game.impulsionadores.pop(game.impulsionadores.index(self))
10            # toca o efeito sonoro de boost
11            game.administradorDeAudio.tocarEfeitoSonoro(os.path.join('Musica', ,
12                'boost.wav'), game)
13            # da uma pausa para que todo o efeito seja tocado corretamente
14            pygame.time.wait(1500)
15            game.ehInvencivel = True

```

A instrução que utilizaremos para inserir o efeito sonoro é a seguinte:  
`game.administradorDeAudio.tocarEfeitoSonoro(os.path.join('Musica', <nome do efeito>), game).`  
Os parâmetros da função `os.path.join()` são a pasta onde estão localizados os sons do jogo e o nome do som que será utilizado. O formato do arquivo de som deve possuir extensão `.wav`.

É interessante ressaltar que a função `pygame.time.wait(1500)` pausa a tela pelo tempo determinado como parâmetro em ms. No caso, a tela foi pausado por 1500 ms. O tempo é determinado de acordo com a duração do efeito sonoro. Essa função é de extrema importância para que o som seja reproduzido apropriadamente e o jogo não fique travado.

## **CODElike a girl**

- 1. Adicione os efeitos sonoros para as colisões com o inimigo, com o obstáculo e com o tiro.**

Diferentemente dos efeitos sonoros, a música de fundo será utilizada nas telas: tela de fim, de início, de jogo e de perguntas.

Para demonstração, vejamos a implementação da música de fundo na tela de fim de jogo.

```
1 game.administradorDeAudio.tocarMusicaDeFundo(os.path.join('Musica', 'menuLoop.wav'), game)
```

A instrução que utilizaremos para inserir a música de fundo em uma tela é a seguinte:  
`game.administradorDeAudio.tocarMusicaDeFundo(os.path.join('Musica', <nome da musica>), game)`  
na qual, analogamente ao efeito sonoro, passamos como parâmetros a pasta onde o som está localizado e o nome do seu arquivo com extensão `.wav`.

## **CODElike a girl**

- 1. Adicione uma música de fundo para as demais telas.**

## Glossário

**\_\_init\_\_** Palavra reservada em Python para o método construtor. 68

**atributos** Características de um objeto. 38

**C++** Linguagem de programação derivada de C com multi paradigmas, entre eles Programação Estruturada e POO. 37

**Classe** Estrutura que define como será o objeto criado a partir dela. 38

**concatenar** juntar listas ou *strings*. 32

**construtor** Método em POO responsável por inicializar um objeto assim que ele é criado. 40, 68

**divisão inteira** ao realizar uma divisão, considera-se apenas a parte inteira do quociente. 15

**herança** Característica em POO na qual uma classe derivada de outra herda os mesmos métodos e atributos diretamente ou indiretamente. 39

**indentação** Formatação para a qual as linhas de código respeitam a um espaçamento do início do texto em relação à margem ("organização" do código) de modo que a hierarquia dos comandos seja de fácil visualização. Em Python, um código não endentado não funciona. 12

**Java** Linguagem de programação orientada a objetos para diversos propósitos. 37

**laços de repetição** comandos são repetidos enquanto uma condição é satisfeita. Normalmente, utilizam-se os comandos **while** e **for**. 13

**linguagem C** Linguagem de programação estruturada de baixo nível. 37

**métodos** Ações de um objeto. 38

**operador** símbolo especial que representa um cálculo. 15

**paradigma** Estilo de programação classificado de acordo com sua estrutura e execução. 37

**Programação Estruturada** Um modelo de programação baseado no uso de sub-rotinas (funções), laços de repetição (*while*, *for*), condicionais (*if*, *switch*) e estruturas em bloco. 37

**Programação Orientada a Objetos** Um modelo de programação baseado no uso de sub-rotinas (funções), laços de repetição (*while*, *for*), condicionais (*if*, *switch*) e estruturas em bloco. 37

**self** Palavra chave para se referir a própria classe em que ela está sendo usada. 39

**software** programa de computador. 6

**super** Palavra chave para se referir a classe pai da classe em que ela está sendo usada. 40

**variável** recurso que armazena dados. 9

**vetor** variável que armazena vários valores. 30

## Siglas

**CPU** Central Processing Unit (Unidade Central de Processamento). 7

**IDE** Integrated Development Environment (Ambiente de Desenvolvimento Integrado). 8

**PE** Programação Estruturada. 37

**POO** Programação Orientada a Objetos. 37