

Trilhas Python

Programação multiparadigma e desenvolvimento Web com Flask



ISBN

Impresso e PDF: 978-85-94188-69-4

EPUB: 978-85-94188-70-0

MOBI: 978-85-94188-71-7

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Quero agradecer à minha esposa Regla e à minha filha Victória, pelo apoio e carinho, pois, sem isso, não teria sido possível escrever este livro. Também quero agradecer profundamente à equipe da Casa do Código, em especial à Vivian Matsui, que acompanhou de perto o desenvolvimento do livro, e ao Adriano Almeida. Deixo aqui meus agradecimentos aos colegas Volney Casas e Charles Tenório, que auxiliaram na revisão técnica do livro.

SOBRE O AUTOR

Eduardo S. Pereira é mestre e doutor em Astrofísica, pelo Instituto Nacional de Pesquisas Espaciais (INPE). Possui graduação em Física pela Universidade Federal de São João Del-Rei. Também realizou pós-doutorado em Astrofísica, pelo INPE, e em Astronomia Observacional/Computacional, pela Universidade de São Paulo (USP).

Ele atua principalmente nos seguintes temas de pesquisa: cosmologia, ondas gravitacionais, astrofísica computacional, processamento de imagens e inteligência artificial. Atualmente, é professor do curso de Ciência da Computação da Faculdade Anhanguera, e Tutor no curso de física na Universidade Virtual de São Paulo (UNIVESP). Trabalha com Python há mais de 6 anos.

SOBRE O LIVRO

Com o avanço das tecnologias Mobile e Web, juntamente com o crescimento do movimento de criação de empresas digitais emergentes, as chamadas startups, a necessidade de desenvolver protótipos de programas que sejam consistentes e que demandem um ciclo mais curto de desenvolvimento acabou por abrir um grande espaço para as linguagens de programação interpretadas de altíssimo nível, tal como a linguagem Python.

O uso da programação Python se estende desde a criação de programas scripts, para o gerenciamento e automação de tarefas de sistemas operacionais, passando pelo desenvolvimento de aplicações Web e por ferramentas de Inteligência Artificial e Visão Computacional. Logo, conhecer os fundamentos da linguagem Python será essencial para quem quiser trabalhar com um desses tópicos.

Neste livro, passaremos por tópicos de programação em Python que vão do básico ao avançado. O material está dividido em três partes que reforçam o aspecto multiparadigma da linguagem.

Na primeira, são apresentados os elementos básicos de Python, como estrutura de dados, funções e decoradores e Programação Estruturada.

Na segunda parte, é dada ênfase à Programação Orientada a Objetos, apresentando de forma prática quais as motivações para usar tal paradigma. Além disso, são abordados tópicos mais avançados da linguagem, como compreensões e geradores, que são

fundamentais para a Programação Funcional em Python.

Na terceira parte, veremos como aplicar o conhecimento de programação Python para o desenvolvimento Web. Para isso, usamos o framework Web chamado Flask, juntamente com uma Camada De Abstração de Banco de Dados ou *Database Abstraction Layer* (PyDAL), para a integração da aplicação com banco de dados. Como o desenvolvimento Web acaba necessitando do auxílio da linguagem JavaScript, para facilitar o acompanhamento do desenvolvimento do projeto da terceira parte do livro, um anexo de introdução ao JavaScript foi adicionado.

A didática escolhida neste livro para apresentar a linguagem Python se dá por meio da solução de problemas. Para isso, usaremos conceitos de desenvolvimento de jogos para poder tornar o conteúdo mais dinâmico. Na primeira parte do livro, desenvolveremos um Jogo da Velha, usando apenas Programação Estruturada. Em seguida, veremos como rescrever esse jogo, aplicando os conceitos de Programação Orientada a Objetos. Na terceira parte, é desenvolvido um aplicativo Web para streaming de áudio, com sistema de login e player de áudio usando JavaScript, HTML5 e CSS.

Pré-requisitos e público-alvo

Este livro foi escrito para aqueles que estão iniciando sua jornada como programador e querem conhecer mais profundamente a linguagem Python. Procurou-se fazer uma abordagem que começa com os fundamentos da linguagem e o uso da Programação Estruturada, passando por outros paradigmas de programação. Assim, uma base sobre algoritmos e algum

conhecimento prévio sobre Python ajudará bastante na compreensão do conteúdo do livro. Como o livro possui uma abordagem prática, apresentando conceitos fundamentais, seguidos de projetos práticos, a leitura pode ser produtiva tanto para os iniciantes em programação quanto para os já iniciados na linguagem, que procuram aprimorar os seus conhecimentos.

Os exemplos do livro foram desenvolvidos em Python 3.6, com compatibilidade para a versão 2.7. A versão usada do Flask foi a 0.12.2.

Sumário

| | |
|---|-----------|
| Python | 1 |
| 1 Introdução | 2 |
| 2 A linguagem Python | 7 |
| 2.1 Configurando o ambiente de trabalho | 8 |
| 2.2 Comentários e variáveis | 12 |
| 2.3 Palavras reservadas e tipos primitivos | 13 |
| 2.4 Operadores matemáticos e lógicos | 24 |
| 2.5 Estruturas condicionais e loops | 25 |
| 2.6 Funções | 28 |
| 2.7 Os argumentos args e kwargs | 30 |
| 2.8 Decoradores | 33 |
| 2.9 Tratamento de exceções | 38 |
| 2.10 Conclusão | 40 |
| 3 Aprendendo na prática: criando jogos | 41 |
| 3.1 Jogo da velha | 45 |
| 3.2 Desenhando a tela do jogo | 49 |

| | |
|---|-----------|
| 3.3 Interação com o usuário | 51 |
| 3.4 O tabuleiro | 54 |
| 3.5 Movendo as peças do jogo | 57 |
| 3.6 Marcando o x da jogada | 60 |
| 3.7 Criando uma Inteligência Artificial | 62 |
| 3.8 Determinando o ganhador | 66 |
| 3.9 Reiniciando a partida | 69 |
| 3.10 Conclusão | 76 |
| Paradigmas de programação | 77 |
| 4 Uma linguagem, muitos paradigmas | 78 |
| 4.1 Paradigmas de programação | 78 |
| 4.2 Conclusão | 84 |
| 5 Programação Orientada a Objetos | 86 |
| 5.1 Classes, objetos e instâncias | 87 |
| 5.2 Herança | 89 |
| 5.3 Público, privado e protegido | 91 |
| 5.4 Descobrimos métodos e atributos | 92 |
| 5.5 Módulos como objetos | 94 |
| 5.6 Pacotes | 96 |
| 5.7 Conclusão | 98 |
| 6 Orientação a Objetos na prática | 99 |
| 6.1 A tela do jogo | 106 |
| 6.2 Os jogadores | 111 |
| 6.3 Função principal | 121 |

| | |
|--|----------------|
| 6.4 Conclusão | 129 |
| 7 Programação Funcional | 131 |
| 7.1 Elementos de Programação Funcional | 131 |
| 7.2 Compreensões | 137 |
| 7.3 Geradores | 142 |
| 7.4 Conclusão | 150 |
| Desenvolvimento Web com Python | 151 |
| 8 Aplicativos Web | 152 |
| 8.1 Frameworks Web | 153 |
| 8.2 Conclusão | 155 |
| 9 Aplicações Web com Python e Flask | 156 |
| 9.1 Nosso projeto | 159 |
| 9.2 Gerando templates dinâmicos | 161 |
| 9.3 Acessando as músicas | 168 |
| 9.4 Manipulando metadados das músicas | 172 |
| 9.5 Full stack: do back-end para o front-end | 177 |
| 9.6 Criando ações no front-end com JavaScript. | 185 |
| 9.7 Conclusão | 203 |
| 10 Persistência de dados | 205 |
| 10.1 Conexão com o banco de dados | 206 |
| 10.2 Criando os modelos de tabelas | 210 |
| 10.3 Integração entre aplicação e banco de dados | 215 |
| 10.4 Conclusão | 223 |

| | |
|---|----------------|
| 11 Sistema de login | 225 |
| 11.1 Organizando o nosso código | 226 |
| 11.2 Cadastro de usuários | 230 |
| 11.3 Tela de login | 234 |
| 11.4 Login | 238 |
| 11.5 Conclusão | 242 |
| 12 LAF - Linux, Apache e Flask | 243 |
| 12.1 Dialogo entre Flask e Apache | 243 |
| 12.2 Conclusão | 248 |
| Considerações finais | 249 |
| 13 Considerações finais | 250 |
| 14 Referências bibliográficas | 252 |
| Apêndice | 255 |
| 15 Apêndice A: JavaScript | 256 |
| 15.1 Comentários, palavras reservadas e tipos | 256 |
| 15.2 Ferramenta de desenvolvimento do navegador | 260 |
| 15.3 Escopo de variáveis | 261 |
| 15.4 Strings e conversão de tipos | 262 |
| 15.5 Arrays ou listas | 265 |
| 15.6 Objetos | 267 |
| 15.7 Operadores lógicos | 269 |
| 15.8 Estruturas condicionais e de laço | 269 |

| | |
|--------------|-----|
| 15.9 Funções | 276 |
|--------------|-----|

| | |
|-----------------|-----|
| 15.10 Conclusão | 277 |
|-----------------|-----|

Python

Na primeira parte do livro, vamos iniciar nossa jornada no universo da programação com a linguagem Python.

Vamos estudar, Python?



INTRODUÇÃO

Antes de começar, precisamos pensar no que nos leva a escolher uma linguagem em vez de outra. Por que escolher Python no lugar de C++ ou Java? Quais motivos levam à criação dos mais diversos tipos de linguagens de programação? Para tentar encontrar respostas a essas questões, primeiro precisamos entender que as linguagens de programação possuem níveis e que cada uma acaba sendo construída tendo em mente filosofias particulares.

Python é uma linguagem interpretada, logo, precisamos da máquina virtual para executar nossos programas, disponível em <https://www.python.org/downloads/>. Diferente da linguagem Java, que também, roda em uma máquina virtual (Java Virtual Machine), Python é fracamente tipada. Isso significa que não é preciso ficar definindo os tipos da variável. Além disso, o controle de bloco é por indentação.

Aqui vale abrir uma pequena observação acerca da classificação do nível de uma linguagem, que pode ser feita de acordo com sua proximidade com a linguagem humana. Quanto mais próxima ela for da linguagem de máquinas, mais baixo será o seu nível. Nesse aspecto, portanto, o código Python é de mais alto nível, pois ele é mais fácil de ser compreendido por um ser humano.

Como exemplo de linguagem de baixo nível, temos a Assembly. Nesse caso, uma instrução do programa equivale a uma operação do computador. Já para linguagens de alto nível, como FORTRAN e C/C++, temos comandos mnemônicos, nos quais instruções em linguagem mais próxima da humana (como `for`, `do` e `while`) são convertidas posteriormente por um compilador em linguagem de máquina.

No caso do Java, temos uma conversão da linguagem para o chamado bytecode, que nada mais é do que um código de mais baixo nível interpretado pela JVM. A grande diferença aqui está na portabilidade. Ou seja, enquanto, nas linguagens como FORTRAN e C/C++, é necessário compilar e modificar o código de acordo com cada arquitetura de hardware e cada sistema operacional, nas linguagens interpretadas basta apenas portar a máquina virtual, facilitando o desenvolvimento de soluções portáteis.

Por outro lado, a semântica da linguagem Java é bem parecida com a do C/C++, por isso ela é uma linguagem de alto nível. Já a linguagem Python, além de ser portátil, ela é mais fácil de ser interpretada por seres humanos, o que a torna de altíssimo nível.

EXEMPLOS DE "OI MUNDO" EM LINGUAGENS DE DIFERENTES NÍVEIS

Hoje, dizemos que uma linguagem de programação pode ser de baixo nível, alto nível ou altíssimo nível. Quanto mais próximo da linguagem de máquina, mais baixo é o nível da linguagem. Como a sintaxe da linguagem Python permite que, com apenas uma breve visualização, uma pessoa consiga identificar os diversos blocos (como funções e iterações), a linguagem é dita de altíssimo nível.

Oi Mundo em Assembly:

```
lea si, string ; Atribui SI ao endereço de string
call printf    ; Coloca o endereço atual na pilha e chama o
processo printf

hlt            ; Encerra o computador
string db "Oi mundo!", 0

printf PROC
    mov AL, [SI] ; Atribui a AL o valor no endereço SI
    cmp AL, 0    ; Compara AL com nulo
    je pfend     ; Pula se a comparação for igual

    mov AH, 0Eh
    int 10h      ; Executa uma função da BIOS que imprime o ca
racter em AL
    inc SI       ; Incrementa o valor de SI em um
    jmp printf   ; Pula para o início do processo

pfend:
    ret          ; Retorna para o endereço na posição atual da
pilha
printf ENDP
```

Oi Mundo em Java:

```
public class OlaMundo{
    public static void main(String[] args){
        System.out.println("Oi Mundo");
    }
}
```

Oi Mundo em Python:

```
print("Oi Mundo")
```

Em geral, as linguagens compiladas apresentam um melhor desempenho (rodam mais rápido) do que as interpretadas. Entretanto, o custo de tempo para desenvolver uma solução em

uma linguagem de baixo nível é muito maior.

Enquanto uma solução usando Python, por exemplo, pode ser finalizada em poucos dias, a mesma solução em C/C++ pode levar muito mais tempo. Outro ponto relevante que sempre precisamos ter em mente ao desenvolver um projeto de software é o chamado **Princípio de Pareto**.

Vilfredo Pareto, ao estudar a distribuição de rendas na Itália (em 1897), descobriu que cerca de 80% da riqueza estava distribuída em apenas 20% da população. Assim, a sociedade era distribuída naturalmente entre dois grupos, sendo que o menor dominava a maior parte da riqueza. Ao analisar outras áreas, também percebemos essa distribuição de responsabilidades.

No caso de software, podemos dizer que a parte mais complicada que temos de desenvolver e que vai consumir 80% do tempo de trabalho representa apenas 20% de código escrito. Já em relação ao tempo de execução de um programa por um computador, temos que 20% do código é que vai consumir mais de 80% do tempo de execução. Com isso em mente, uma estratégia tem sido a de escrever apenas a parte do código com maior custo computacional em uma linguagem compilada, deixando a maior parte dele em linguagens de mais alto nível.

A escolha da linguagem a ser usada em um projeto vai depender de características específicas. O uso da linguagem C/C++ é fundamental quando se quer criar sistemas embarcados e drives, ou seja, quando o desenvolvimento está ligado ao hardware. Já a linguagem Java ganhou muito espaço no meio corporativo, enquanto linguagens como Python e Matlab são muito comuns no meio acadêmico. Com isso, vemos que não existe uma solução

única para um problema, sendo que ainda podemos combinar linguagens diferentes para desenvolver um mesmo projeto.

Ao longo deste livro, abordaremos com mais detalhes os elementos da linguagem Python, sempre acompanhados de projetos práticos, para que o leitor possa se sentir seguro sobre o que está aprendendo com esse material. Nos próximos capítulos, vamos conhecer a linguagem Python de forma mais profunda.

Todos os códigos deste livro estão disponíveis no GitHub do autor: https://github.com/duducosmos/livro_python.

A LINGUAGEM PYTHON

Neste capítulo, vamos fazer uma breve introdução à linguagem Python e aos principais elementos que vamos utilizar ao longo dos projetos desenvolvidos neste livro. O objetivo principal aqui é ser um ponto de partida para os iniciantes, ou para aqueles que já trabalham em outras linguagens e têm interesse em conhecer Python.

A linguagem Python foi desenvolvida inicialmente por Guido Van Rossum no Instituto Nacional de Pesquisas para Matemática e Ciência da Computação nos países baixos, sendo sucessora da linguagem ABC. O principal objetivo era ser uma linguagem mais acessível ao meio acadêmico e alternativa a linguagens como Matlab e IDL.

Em 1991, saiu a versão 0.9 e, em 2001, saiu a 2.1 com licença renomeada para Python Foundation License, com o objetivo de se ter uma linguagem verdadeiramente *open source*. Hoje, temos a versão 2.x e a 3.x, sendo que algumas mudanças significativas ocorreram nessa transição. Nem todos os projetos foram completamente portados para a nova versão, então ainda existem ambas as versões da linguagem paralelamente no mercado.

A versão 3 traz muitas novidades e evoluções importantes, por

isso recomenda-se usar a versão mais recente. Apesar de existirem duas, a essência da linguagem é a mesma, não havendo diferenças muito drásticas de uma versão em relação a outra. Ou seja, se você aprender Python 3.x, mas estiver atuando em um projeto que usa a versão 2.x, não terá nenhum problema – exceto que algumas facilidades novas não estarão disponíveis nas versões mais antigas da linguagem. Na data em que este livro foi escrito, as mais recentes eram a **3.6.1** e **2.7.13**.

Uma das coisas mais aclamadas pela comunidade está no fato de que Python já vem com **baterias incluídas**. Isso, na verdade, significa que o interpretador traz uma série de módulos e funções embutidas na linguagem, conhecidas também por *built-in*. Para ter acesso a informações mais detalhadas de como usar cada uma, é recomendado acessar a documentação oficial mantida pela **Python Software Foundation**, disponível em <https://docs.python.org/3/>.

Ao navegar nessa página, você verá logo abaixo do link sobre a referência das bibliotecas o texto *keep this under your pillow*, que indica que você deve manter isso embaixo do seu travesseiro. Isso quer dizer que ele deve ser um texto de referência que deve ser consultado de forma constante.

2.1 CONFIGURANDO O AMBIENTE DE TRABALHO

Nos sistemas tipo Linux e Unix (como Ubuntu e BSD) e no OSX da Apple, o Python já vem pré-instalado. Já para sistemas Windows, é preciso baixar e instalar o interpretador em <https://www.python.org/downloads/>.

Para usuários do Windows, após baixar a versão adequada para o seu sistema, basta seguir normalmente pelo procedimento de instalação. Depois, é interessante adicionar o interpretador Python ao PATH do Windows, que em geral consiste nos seguintes passos (podendo variar um pouco de uma versão para outra):

1. Clicar no Menu do Windows;
2. Clicar com o botão direito na opção Computador e selecionar a opção Propriedades ;
3. Clicar na opção Configurações avançadas do Sistema no painel esquerdo;
4. Clicar no botão Variáveis de ambiente na parte inferior da caixa de diálogo;
5. Editar o conteúdo da variável PATH nas variáveis de sistema;
6. Adicionar o diretório escolhido para a instalação do Python, por exemplo, C:\Python2.7 , usando ; como separador.

Nas versões mais atuais do instalador (Python 3.6), no processo final de instalação, já é dada a opção de adicionar o Python à variável de ambiente do Windows de forma automática.

Para o Windows, o Python possui um *Ambiente integrado de Desenvolvimento e Aprendizado* (em inglês *Integrated Development and Learning Environment*, ou IDLE) que permite rodar o Python no modo terminal. Isso facilita muito a vida do desenvolvedor, pois permite testes rápidos de trechos de códigos, além de facilitar o processo de aprendizagem da linguagem. Na figura a seguir, vemos um exemplo do terminal Python (Python Shell) para o Windows.

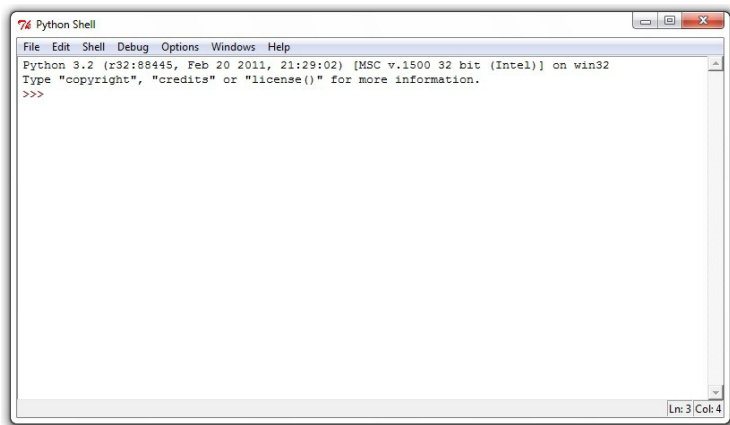


Figura 2.1: Terminal do interpretador Python 3.2 rodando no Windows

Os exemplos deste capítulo serão apresentados como se estivessem sendo executados em um terminal desse tipo. Para sistemas OSX e Linux, basta abrir o terminal e digitar `Python`, que um terminal similar a esse será executado.

Ambiente virtual

Quando trabalhamos em vários projetos diferentes, vemos que cada um vai usar um grupo particular de bibliotecas e pacotes. Em alguns casos, pode ocorrer algum grau de incompatibilidade entre eles, ou um projeto acabar dependendo de versões diferentes de um mesmo pacote. Além disso, a instalação direta de um monte de bibliotecas diretamente no seu sistema pode gerar um ambiente "poluído", com um monte de coisas que serão necessárias apenas em um determinado momento, mas não serão mais usadas mais tarde.

Uma boa prática é criar ambientes isolados para cada projeto.

A grande vantagem é que, ao final, você será capaz de mapear todas as dependências que são exclusivas do seu projeto.

Para o Python2.X, é preciso ter instalado o `setuptools` para, em seguida, instalarmos o `virtualenv`. Por exemplo, para usuários do sistema do tipo Debian:

```
sudo apt-get install python-setuptools
sudo easy_install virtualenv
```

Já para a versão Python3.x, é recomendado o uso do `venv`. Para usuários do sistema tipo Debian:

```
sudo apt-get install python3-venv
```

Para criar um ambiente isolado usando o `virtualenv`, usamos o comando:

```
virtualenv --no-site-packages teste
```

A pasta de teste será criada com as seguintes subpastas:

```
bin include lib share
```

Já para criar o ambiente virtual usando `venv`, usamos o comando:

```
python3 -m venv teste03
```

Nesse caso, será criada a pasta `teste03`.

Para ativar o ambiente virtual (tanto para o `virtualenv` quanto para o `venv`), usamos o comando:

```
source teste03/bin/activate
```

No caso do Windows, abrimos o `cmd.exe`, mudamos para o diretório raiz em que se encontra a pasta do ambiente virtual e, em

seguida, usamos o comando:

```
C:\> <env>\Scripts\activate.bat
```

Uma vez que o ambiente virtual esteja ativado, vamos instalar o `flask` :

```
pip install flask
```

Todas as bibliotecas e suas dependências serão instaladas.

2.2 COMENTÁRIOS E VARIÁVEIS

Um programa em Python nada mais é que um arquivo de texto com extensão `.py` . Lembre-se de que os comentários podem ser feitos usando sustenido (`#` , o atualmente famoso *hashtag*), aspas simples e aspas duplas, sendo que comentários com mais de uma linha abrem e fecham com três aspas simples, ou três aspas duplas

```
'Isso é um comentário em uma linha'
"Isso é um comentário em uma linha"
#Isso é um comentário em uma linha

"""
Isso é um comentário
em mais de uma linha
"""

'''
Isso é um comentário
em mais de uma linha
'''

3.0 # Isso é um comentário
```

Como dito anteriormente, Python é uma linguagem dinâmica não tipada. Isso significa que o tipo de uma variável é automaticamente reconhecido pelo interpretador em tempo de

execução. Exemplos de variáveis em Python são:

```
>>> b = 2 # b é um inteiro
>>> print(b)
2
>>> b = b * 2.0 # Agora b é um primitivo do tipo float
>>> print(b)
4.0
>>> b = "Oi Mundo" # Agora b é uma string
>>> print(b)
"Oi Mundo"
```

2.3 PALAVRAS RESERVADAS E TIPOS PRIMITIVOS

Em toda linguagem de programação, existem palavras especiais, usadas pelo compilador ou interpretador. São as chamadas palavras reservadas, o que significa que não se pode definir funções ou variáveis com elas, que, para a linguagem Python, são:

```
and, assert, break,
class, continue, del, def,
if, except, exec, finally,
for, from, global, lambda, not,
or, pass, print, raise,
return, try, while, elif,
else, import, in, is, True, False, print, nonlocal
```

Além das palavras reservadas, existem estruturas de dados fundamentais, chamados de *tipos pré-definidos de dados*, ou *tipos primitivos*. No caso do Python, eles podem ser simples ou compostos. Alguns dos tipos simples da linguagem são os inteiros, inteiros longos, ponto flutuante, complexos e cadeias de caracteres (Strings).

Exemplo de tipos de dados simples:

```
>>> x = 1 # Inteiro de precisão fixa
>>> x = 100L # Inteiro de precisão arbitrária
>>> x = 1.234154 # Ponto flutuante
>>> x = 1 + 2j # Número complexo. j raiz de -1
>>> x = "Oi Mundo" # Cadeia de caracteres
```

Para converter esses tipos simples em outros – método também conhecido como *casting* –, usamos as funções embutidas `int`, `float`, `str`, `long`, `complex`, `bool`.

```
>>> x = 1
>>> print(float(x))
1.0
>>> y = "0.0"
>>> print(int(y))
0
>>> z = 3.4243
>>> print(str(z))
"3.4243"
```

Por exemplo, no primeiro caso, definimos `x` como sendo de valor inteiro; já na segunda linha, convertemos essa variável em ponto flutuante com a função `float`. É comum o uso do termo em inglês **built-in** para se referir a tudo que já vem embutido no interpretador Python, como módulos e funções internas.

Já alguns tipos primitivos compostos usados com mais frequência são as listas, as tuplas e os dicionários.

Exemplo de tipos de dados compostos:

```
x = [1, 2, 3, 4] #lista
y = ['a', 'b', 'c'] #lista
x = ('a', 'b', 'c') #tupla
y = (25, 73, 44) #tupla
myDic = {1:"um", 2:"dois", 3:"tres"} #Dicionário
cesta = ['maca', 'pera', 'uva', 'pera'] #lista
```

Os tipos primitivos simples são objetos mais próximos do que

estamos acostumados; já os primitivos compostos são fundamentais na construção da estrutura de dados, como é o caso dos dicionários, em que se cria a estrutura chave/valor. Por fim, as listas podem ser pensadas como vetores, que podem ter tamanhos variáveis.

As listas e as tuplas são muito parecidas, em geral. Elas são como vetores ou *arrays* que existem em outras linguagens. A grande diferença entre elas é que as listas são objetos mutáveis, nos quais posso adicionar ou remover elementos, e fazer modificações após sua criação. As tuplas são imutáveis; uma vez criadas, a única forma de redefini-las é sobrescrevendo a variável que a representava.

Por fim, os dicionários, como o próprio nome sugere, são formados por pares chave/valor, em que uma chave permite retornar um dado valor. Devido à importância dessas duas estruturas de dados, mais adiante abordaremos mais detalhes das listas e tuplas.

Quando usamos o terminal Python, podemos fazer a chamada da função `help`, que permite visualizar informações importantes sobre funções, módulos e até mesmo dos tipos primitivos. A seguir, veja um trecho do que aparece ao buscarmos mais informações sobre strings usando o `help`:

```
>>> help('string')
Help on module string:
```

NAME

string - A collection of string constants.

MODULE REFERENCE

<https://docs.python.org/3.5/library/string>

The following documentation **is** automatically generated **from** the Python source files. It may be incomplete, incorrect **or** include features that are considered implementation detail **and** may vary between Python implementations. When **in** doubt, consult the module reference at the location listed above.

DESCRIPTION

Public module variables:

```
whitespace -- a string containing all ASCII whitespace
ascii_lowercase -- a string containing all ASCII lowercase letters
ascii_uppercase -- a string containing all ASCII uppercase letters
ascii_letters -- a string containing all ASCII letters
digits -- a string containing all ASCII decimal digits
hexdigits -- a string containing all ASCII hexadecimal digits
:
```

Para continuar lendo o texto que aparece na ajuda, basta pressionar **Enter** ; já para sair da ajuda, basta pressionar a tecla **Q** . A seguir, veremos com mais detalhes alguns tipos primitivos.

Strings ou cadeias de caracteres

As cadeias de caracteres, ou *strings*, podem ser pensadas como um vetor de caracteres, mas imutáveis, sendo representadas por aspas simples (`'` , também conhecida como apóstrofo) ou aspas duplas (`"`). Quando se quer escrever strings com mais linhas, podemos usar três aspas duplas (`"""`) ou três apóstrofes (`'''`). Dessa forma, é possível acessar partes de uma string usando `:` (dois pontos),

```
>>> a = "O rato roeu a roupa do rei"
>>> print(a[2:6])
```

```
"rato"
```

No código anterior, definimos a variável `a` como a string `"O rato roeu a roupa do rei"`. Em seguida, pedimos para ser exibido o trecho que começa com índice `2` e que termina ao ter um total de 6 caracteres desde o início da string.

Já no código a seguir, ao tentar modificar o caractere `c` por `d`, representado pelo índice `2`, o interpretador levanta um erro, indicando que esse tipo de objeto não suporta uma troca de elemento:

```
>>> a = "Faca"
>>> a[2] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Dessa forma, uma vez que uma string é criada, não pode mais ser modificada. Com isso, quando se concatena (junta) duas strings, o que se faz é criar um novo objeto.

Veja alguns exemplos:

```
>>> a = "Oi"
>>> b = "Mundo"
>>> c = a + " " + b
>>> print(c)
Oi Mundo
>>> print(c[0:3])
Oi "
```

Outro ponto importante é o processo de formatação de uma string. Usando o método `format` (ou o operador `%`), podemos inserir valores em posições específicas em uma string. Como exemplo, considere que uma pessoa necessita calcular o imposto de renda sobre os lucros de uma operação. Nesse caso, a pessoa

entra com o valor do lucro, e queremos exibir o valor do imposto.

Assim, consideramos que o imposto será 26% do lucro. Nesse momento, vamos usar a função embutida `input()`, que captura um valor digitado pelo usuário.

```
>>> lucro = float(input("Digite o valor do lucro: "))
Digite o valor do lucro: 3000
>>> imposto = lucro * 0.26
>>> resultado = "Sobre lucro de R$ {0} você deverá pagar R$ {1} d
e imposto".format(lucro, imposto)
>>> print(resultado)
Sobre lucro de R$ 3000.0 você deverá pagar R$ 780.0 de imposto
```

Note que, na primeira linha, pedimos ao usuário para digitar o valor do lucro. Em seguida, calculamos o imposto como sendo 26% do lucro. Por fim, criamos a string `resultado`.

Na posição em que queremos exibir o valor do lucro, usamos o comando `{0}` e, para posicionar o valor do imposto calculado, usamos o comando `{1}` dentro da string. No final da string, usamos o método `format`, no qual colocamos o lucro na primeira posição, e o imposto logo em seguida.

Dessa forma, `{0}` indica que é para pegar o primeiro elemento que aparece entre parênteses no `format`. A outra alternativa é com o operador `%`:

```
>>> lucro = float(input("Digite o valor do lucro: "))
Digite o valor do lucro: 3000
>>> imposto = lucro * 0.26
>>> resultado = "Sobre lucro de R$ %s você deverá pagar R$ %s de
imposto" % (lucro, imposto)
>>> print(resultado)
Sobre lucro de R$ 3000.0 você deverá pagar R$ 780.0 de imposto
```

A seguir, temos mais um exemplo do uso de formatadores de string:

```
>>> a = 23.45
>>> b = "0 numero %s foi adicionado à cadeia de caracteres " % 2
3.45
>>> print(b)
0 numero 23.45 foi adicionado à cadeia de caracteres
>>> print("Os valores são, a = {0}, b = {1}, c = {2}.".format(42
, 23.5, 64))
Os valores são, a = 42, b = 23.5, c = 64
```

Vale ressaltar que o número que aparece entre as chaves está ligado à sequência de parâmetros que é passada ao `format` .

```
>>> print("Os valores são, a = {2}, b = {0}, c = {1}.".format(42,
23.5, 64))
Os valores são, a = 64, b = 42, c = 23.5
```

Listas

As listas são tipos compostos primitivos mutáveis. Isso significa que é possível modificar um dado elemento de uma lista. São objetos que lembram um array, em que um trecho da lista pode ser capturado usando `:` (dois pontos).

Assim como as strings, o índice de uma lista começa com `0` , sendo que, para obter o último elemento, usamos `-1` . Dessa forma, índices inteiros negativos indicam que estamos buscando elementos de forma reversa na lista. Veja a seguir alguns exemplos de operações sobre listas:

```
>>> a = [1, 2, 3, 4]
>>> print(a[0])
1
>>> print(a[-1])
4
>>> print(a[1:3])
[2, 3]
>>> a[0] = 42
>>> print(a)
[42, 2, 3, 4]
```


Inicialmente, criamos a lista `a` e, no primeiro `print`, pedimos para exibir o primeiro elemento da lista, demarcado pelo índice `0`. Em seguida, usamos o `-1` para indicar que queremos imprimir o último valor da lista. Assim, usando valores negativos, podemos pegar elementos da lista em sentido contrário.

Para o caso de `print(a[1:3])`, é apresentado na tela o elemento que começa com o índice `1`, seguindo por até o terceiro elemento da lista (índice `3 - 1 = 2`). No caso da lista, podemos modificar um valor localizado por um determinado índice, tal como em `a[0]=42`. Desta forma, quando voltamos a exibir a lista, é possível ver que o primeiro valor passou a ser `42`.

Alguns métodos muito usados das listas são:

- `append` – Adiciona um novo elemento ao final da lista;
- `insert` – Insere um elemento a uma dada posição.

Para obter o número de elementos de uma lista, usamos a função interna `len`:

```
>>> a = [1, 2, 3, 4, 5]
>>> a.append(6) #Adicionando o 6 ao final da lista
>>> print(a)
[1, 2, 3, 4, 5, 6]
>>> a.insert(0, 0.0) #adicionando 0.0 na posicao 0
>>> print(a)
[0.0, 1, 2, 3, 4, 5, 6]
>>> print(len(a)) # mostrar o num. de elementos da lista
7
>>> a[2:4] = [5.0, 5.0] #Modificar trecho da lista
>>> print(a)
[0.0, 1, 5.0, 5.0, 4, 5, 6]
```

Um ponto muito importante para o caso de objetos mutáveis, tal como as listas, está no fato de que, ao se declarar `b = a`, com

a sendo a lista, isso não resultará em um novo objeto. Porém, b passará a ser uma referência para a . Dessa forma, qualquer modificação em b vai resultar em uma modificação em a .

Para gerar uma cópia, usamos a declaração do tipo `c=a[:]` .
Veja:

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(b)
[1, 2, 3]
>>> b[0] = 42
>>> print(a)
[42, 2, 3]
>>> c = a[:]
>>> print(c)
[42, 2, 3]
>>> c[0] = 0
>>> print(c)
[0, 2, 3]
>>> print(a)
[42, 2, 3]
```

Dicionários

Dicionários são elementos mutáveis formados por pares de chave e valor. Eles são representados por chaves da forma `{chave: valor}` , sendo que, a partir de uma chave, retorna-se o valor correspondente. Por ser mutável, podemos criar um dicionário vazio e ir adicionado elementos após sua criação.

```
>>> diasDaSemana = {1:"domingo", 2:"segunda",
                    3:"terça", 4:"quarta",
                    5:"quinta", 6:"sexta"}
>>> print(diasDaSemana)
{1: 'domingo', 2: 'segunda', 3: 'terca', 4: 'quarta',
 5: 'quinta', 6: 'sexta'}
>>> diasDaSemana[7] = "sabado" #Adiciona elemento ao Dicionario
>>> print(diasDaSemana)
```

```
{1: 'domingo', 2: 'segunda', 3: 'terca', 4: 'quarta',
 5: 'quinta', 6: 'sexta', 7: 'sábado'}
>>> print(diasDaSemana[3]) #Retorna o valor da chave 3
"terça"
```

No código anterior, criamos o dicionário `diasDaSemana`, e fizemos com que os números inteiros (de 1 a 6) representem os dias de domingo a sexta-feira. O inteiro representa a chave, e as strings com os nomes equivalentes dos dias da semana representam os valores.

Usando o código `diasDaSemana[7] = "sábado"`, acabamos por adicionar a chave de valor `7`, e indicamos que o valor associado a ela será a string `"sábado"`. Note que, ao pedirmos para imprimir na tela a chave `3`, com o comando `print(diasDaSemana[3])`, é exibida a string `"terça"`.

Para saber os valores, chaves ou itens que compõem os dicionários, usamos os métodos `keys`, `values` e `items`:

```
>>> diasDaSemana = {1:"domingo", 2:"segunda",
                    3:"terca", 4:"quarta",
                    5:"quinta", 6:"sexta", 7: 'sábado'}
>>> print(list(diasDaSemana.keys()))
[1, 2, 3, 4, 5, 6, 7]
>>> print(list(diasDaSemana.values()))
['domingo', 'segunda', 'terca', 'quarta', 'quinta', 'sexta', 'sab
ado']
>>> print(list(diasDaSemana.items()))
[(1, 'domingo'), (2, 'segunda'), (3, 'terca'),
 (4, 'quarta'), (5, 'quinta'), (6, 'sexta'), (7, 'sabado')]
```

No Python 3, esses métodos retornam iteradores correspondentes ao seu significado. Por exemplo, o método `keys` retorna um objeto do tipo `dict_keys`. Por ser iterável, podemos usar esses objetos diretamente em laços com o `for in`, por exemplo:

```
>>> diasDaSemana = {1:"domingo", 2:"segunda",
                    3:"terca", 4:"quarta",
                    5:"quinta", 6:"sexta", 7: 'sábado'}
>>> for key in diasDaSemana.keys():
...     print(key)
1
2
3
4
5
6
7
```

Tuplas

Assim como as strings, as tuplas são imutáveis, ou seja, não podemos modificá-las após sua criação, embora se assemelhem muito com as listas. Porém, como as tuplas aceitam qualquer objeto, podemos fazer modificações dos componentes individuais. Matematicamente, uma tupla é definida como uma sequência finita de objetos e é representada por `()` (parênteses).

```
>>> a = ("carro", 1, [2,3,4])
>>> print(a[0])
carro
>>> a[2].append(5)
>>> print(a)
('carro', 1, [2, 3, 4, 5])
>>> a[1] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

No código anterior, criamos a tupla `a` e, em seguida, pedimos para exibir na tela o elemento de índice `0`. Como o item localizado pelo índice `2` era uma lista, foi adicionado a essa lista o número `5`. Porém, ao tentar trocar o valor do inteiro `1` (localizado pelo índice `1`) por `42`, o interpretador levantou um

erro, indicando que esse valor não pode ser modificado.

No caso da lista, o objeto em si não foi modificado, apenas adicionamos mais um elemento. A tupla tem apenas os métodos `count` e `index`, que retornam, respectivamente, o número total de elementos e o índice da primeira ocorrência de um elemento:

```
>>> a = ("carro", 1, [2,3,4], "carro")
>>> a.count("carro")
2
>>> a.index("carro")
0
```

2.4 OPERADORES MATEMÁTICOS E LÓGICOS

As operações aritméticas comuns são suportadas pelo Python, sendo que algumas delas, como soma e multiplicação, também são definidas para strings. Também temos as operações de comparação. Ambas são apresentadas na tabela a seguir.

| Operador | Símbolo |
|------------------|---------|
| Soma | + |
| Subtração | - |
| Divisão | / |
| Exponenciação | ** |
| Resto da divisão | % |
| Menor que | < |
| Maior que | > |
| Menor igual a | <= |
| Maior igual a | >= |
| Igual | == |
| | |

Um exemplo de operação de soma e multiplicação não intuitivo está no tratamento de listas e strings, apresentado a seguir:

```
>>> a = "oi "  
>>> print(3 * a)  
"oi oi oi "  
>>> b = [1, 2, 3]  
>>> print(3 * b)  
[1, 2, 3, 1, 2, 3, 1, 2, 3]  
>>> print(b + [4, 5])  
[1, 2, 3, 4, 5]
```

2.5 ESTRUTURAS CONDICIONAIS E LOOPS

As estruturas condicionais em Python são `if`, `elif` e `else`; já as estruturas de laço, ou loop, são `while/else` e `for in/else`, sendo que as declarações são definidas por blocos de indentação. Nas linguagens tipo C, como Java, JavaScript e C#, os blocos de código são demarcados por chaves (`{}`).

Em Python, para obrigar essa organização visual do código, a definição de blocos é feita através de espaços em branco. Em geral, recomenda-se o uso de quatro espaços em branco para definir um novo bloco de código.

É possível também usar tabulação, porém, devido ao fato de que sistemas do tipo Linux/Unix interpretam a tabulação de forma diferente dos sistemas Windows, não se recomenda misturar as duas formas de criar blocos. A documentação oficial da linguagem recomenda o uso de quatro espaços em branco.

Para o condicional `if/elif/else`, temos o seguinte código:

```

a = int(input("Digite um número: "))
if(a < 10):
    print("O valor de a é menor que 10")
elif(a == 10):
    print("O valor de a é igual a 10")
else:
    print("O valor de a é maior que 10")

```

Na primeira linha, pedimos para o usuário digitar um número. Com a função `int`, o valor digitado será convertido em um inteiro. Em seguida, comparamos o valor digitado. Se o número for menor que 10, será exibida a mensagem "o valor de a é menor que 10". O `elif` será alcançado somente se for digitado o número 10.

Na verdade, se digitarmos qualquer número maior ou igual a 10 e menor que 11, como 10.9, a função `int` vai convertê-lo para 10. Dessa forma, o `else` será alcançado quando nenhuma das duas opções anteriores for verdadeira, ou seja, quando o valor de `a` for maior que 10.

Note que, ao final de cada declaração, usamos o símbolo `:`, indicando que na linha de baixo será iniciado um novo bloco de código demarcado por indentação. Já para operações em laço, ou loop, nas quais queremos que um bloco de código se repita até que uma condição seja alcançada, usamos o `while`:

```

a = 0
while(a < 10):
    print(a)
    a += 1
else:
    a = 0

```

No código anterior fizemos com que o valor de `a` fosse inicialmente igual a 0. Em seguida, escrevemos `while(a < 10):`,

que significa: faça enquanto `a` for menor que 10. Dentro do bloco de código, pedimos para que o valor corrente de `a` seja impresso no monitor e que, depois, o valor de `a` seja acrescido de 1.

O comando `else`, logo depois do `while`, fará com que o valor de `a` volte a 0 somente após a condição avaliada pelo `while` for verdadeira. Ou seja, primeiro `a` terá valor igual a 11, interrompendo o laço e executando o que está dentro do `else`. Esse processo seria muito similar a ter um loop infinito no qual a condição de parada é feita por uma combinação de `if` e `break`:

```
a = 0
while(True):
    if(a < 10):
        print(a)
        a += 1
    else:
        a = 0
        break
```

Note que a palavra `break` é usada para indicar a interrupção do laço. Outra estrutura de laço é o `for in`, que atua sobre cada elemento de um objeto iterável, tal como as listas.

```
a = [1, 2, 3, 4]
for i in a:
    print(i)
```

O trecho de código anterior terá como saída:

```
1
2
3
4
```

Logo, o que estamos fazendo é atribuir a `i` cada um dos elementos da lista, um de cada vez. Em seguida, fazemos uma operação; no nosso caso, exibimos o valor de `i` na tela. Esse

processo vai se repetir até que todos os elementos da lista sejam varridos.

2.6 FUNÇÕES

Uma função é um bloco de código com um nome que recebe um conjunto de argumentos e retorna valores. A sintaxe da função é:

```
def nome_da_funcao(arg0, arg1, ..., argn):  
    #Bloco de código  
    return valor_retornado
```

O uso da palavra `return`, ao final da função, não é obrigatório, porém, se ela não for adicionada, o Python automaticamente adicionará um `return None`, ou seja, a função retornará de forma implícita o `None`. Isso é algo equivalente ao vazio (ou `void`, em linguagens tipo C), por exemplo.

Além da criação de funções via bloco de código, também é possível fazê-lo em uma única linha usando a função **lambda**, que também é exemplificada a seguir:

```
>>> def c(a, b):  
...     d = a + b  
...     return d  
...  
>>> print(c(2, 3))  
5  
>>> func = lambda x, y: x * y  
>>> z = func(3, 3)  
>>> print(z)  
9
```

Com relação ao escopo das variáveis, toda variável dentro da função armazena o valor na tabela de símbolos local. Referências às

variáveis são buscadas primeiro na tabela local, depois na tabela de símbolos globais e, finalmente, na tabela de símbolos internos (*built-in*) – que, nesse último caso, representam as funções, os módulos e as variáveis padrões da linguagem.

No exemplo seguinte, vemos que o valor de `v` fora da função é zero e, dentro da função, é 1. O `v` declarado dentro da função tem escopo local, isto é, qualquer modificação em `v` dentro da função não modificará o valor de `v` fora dela.

```
>>> v = 0
>>> def teste(t):
...     v = 1
...     print("O valor de v é %i, o valor de t é %i " %(v, t))
>>> teste(3)
"O valor de v é 1, o valor de t é 3"
>>> print(v)
0
```

Usando o termo `global`, podemos fazer com que uma variável possa ser acessada de forma global:

```
>>> v = 0
>>> def teste(t):
...     global v
...     v = 42
...     print("O valor de v é %i, o valor de t é %i " %(v, t))
>>> teste(3)
"O valor de v é 42, o valor de t é 3"
>>> print(v)
42
```

Se quisermos acessar uma variável definida externamente, sem ter de alterar seu valor dentro da função, podemos fazer o seguinte:

```
>>> v = 0
>>> def teste(t):
...     saida = "O valor de v é {0}, o valor de t é {1} ".format(
v, t)
...     print(saida)
```

```
>>> teste(3)
"0 valor de v é 0, o valor de t é 3"
>>> print(v)
0
```

Aqui, primeiramente declaramos `v` com valor inicial igual a 0. Em seguida, criamos a função `teste` que recebe como parâmetro a variável `t`. Note que não declaramos `v` dentro da função, porém, ao formatar a string que será armazenada na variável `saida`, usamos `v` como primeiro elemento passado ao método `format(v, t)`. Isso indica que `v` tem escopo global, em que apenas capturamos o seu valor dentro da função.

2.7 OS ARGUMENTOS ARGS E KWARGS

Em alguns casos, torna-se interessante criar funções com número de parâmetros variáveis, ou seja, criar funções que recebem parâmetros de forma dinâmica. Em Python, usamos os parâmetros `*args` e `**kwargs` para isso.

O `*` (asterisco) é um operador que transforma listas em argumentos de uma função; já o `**` transforma um dicionário em chaves de argumentos de função. Vamos criar a função que soma dois números e testar o conceito do operador `*`:

```
>>> def soma(a, b):
...     return a + b
...
>>> valores = [1, 2]
>>> b = soma(valores)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: soma() takes exactly 2 arguments (1 given)
>>> b = soma(*valores)
3
```

No exemplo anterior, criamos a função `soma`, que recebe dois argumentos e, em seguida, criamos a lista `valores` com dois elementos. Ao tentar passar `valores` para a função `soma`, o interpretador levanta um erro, indicando que a função precisa receber dois argumentos, mas a lista é apenas um elemento.

Agora, com o comando `*valores`, a lista será transformada em sequência de argumentos para a função, ou seja, o valor de `a` será o primeiro valor da lista e o valor de `b` será o segundo valor. Porém, se a nossa lista tiver mais elementos que o número de argumentos que a função deve receber, também será levantado um erro:

```
>>> valores = [1, 2, 3]
>>> b = soma(*valores)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: soma() takes exactly 2 arguments (3 given)
```

Para o caso do operador `**`, transformamos um dicionário em argumentos:

```
>>> valores = {'a': 2, 'b': 3}
>>> b = soma(**valores)
```

No código anterior, criamos um dicionário em que as chaves têm o mesmo nome que os argumentos da função `soma`. Dessa forma, o operador `**` vai atribuir o valor 2 ao argumento `a`, e o valor 3 ao argumento `b`.

Podemos também passar o `*` e o `**` diretamente na definição da função, indicando que passaremos uma lista de argumentos arbitrária para a função ou um conjunto arbitrário de argumentos:

```
>>> def soma(*args):
```

```

...     tmp = 0
...     for arg in args:
...         tmp += arg
...     return tmp
>>> b = soma(1, 2)
>>> print(b)
3
>>> c = soma(1, 2, 3, 4, 5)
>>> print(c)
15

```

Na definição da função, passamos o argumento `*args`. Dessa forma, estamos dizendo que nossa função poderá receber um número arbitrário de argumentos. Dentro da função, criamos uma variável auxiliar inicializada com 0.

Como não sabemos *a priori* quantos números teremos de somar, criamos um laço para avaliar cada um dos argumentos que serão recebidos. Em seguida, somamos um a um e retornamos o resultado ao final. Veja que, para `b`, passamos dois argumentos; já para `c`, passamos cinco. Ou seja, o operador `*`, na definição de argumentos da função, converte o que for passado posteriormente em uma lista com esses elementos. Por esse motivo, podemos usar a operação `for arg in args:`.

No caso do operador `**`, na definição da função, tudo o que for passado na forma de par de argumento e valor será convertido em dicionários:

```

>>> def funcao(**kwargs):
...     if(kwargs is not None):
...         for chave, valor in kwargs.items():
...             print("Chave: {0}; Valor: {1}".format(chave, valor))
>>> funcao(a=1, b=2)
Chave: a; Valor: 1
Chave: b; Valor: 2

```

Veja que passamos `**kwargs` como parâmetro de `funcao` . Depois, verificamos se algum argumento foi passado, verificando se `kwargs` não é vazio (tipo `None`). Então, pegamos os pares chave e valor do dicionário `kwargs` . O par chave/valor no argumento da função foi passado ao colocarmos `a=1, b=2` .

Esses dois métodos de definição de argumentos são interessantes quando queremos passar quantidades arbitrárias de parâmetros para uma função.

2.8 DECORADORES

Um decorador (ou *decorator*) é algo que nos permite modificar ou adicionar um novo comportamento a uma função. Nesse caso, estamos de certa forma modificando o aspecto da função, o que lembra muito o modelo de Programação Orientada a Aspectos, presente, por exemplo, em Java.

Para começar a entender esse conceito em Python, vale relembrar que podemos ter variáveis globais que podem ser acessadas por outras funções. Além disso, podemos definir funções dentro de outras funções. De forma prática, temos o seguinte exemplo:

```
>>> def primeira_funcao(valor):  
...     quadrado = valor * valor  
...     def segunda_funcao():  
...         return "O quadrado é {}".format(quadrado)  
...     return segunda_funcao()
```

Aqui, criamos a função `primeira_funcao` , que recebe com parâmetro a variável `valor` . Note que a função `segunda_funcao` , definida internamente, não recebe nenhum

parâmetro, mas consegue "ver" o que está contido na variável `quadrado` .

Assim, apesar de a variável `quadrado` estar definida somente no escopo da função `primeira_funcao` , suas variáveis podem ser acessadas, como se fossem globais, por funções definidas internamente. Agora, um exemplo de mudança de comportamento pode ser visto a seguir:

```
>>> def info(nome):
...     return "Bem-vindo {0} ao nosso sistema".format(nome)
...
>>> def decorador_p(func):
...     def funcao_encapsulada(nome):
...         return "<p>{0}</p>".format(func(nome))
...     return funcao_encapsulada
...
>>> info_em_p = decorador_p(info)
>>> print(info_em_p("Eduardo"))
<p>Bem-vindo Eduardo ao nosso sistema</p>
```

Primeiramente, criamos a função `info` , que recebe um nome como parâmetro e retornará uma string com uma mensagem padrão de boas-vindas com o nome que foi passado. Porém, queríamos ter o poder de pegar funções que retornam strings e adicionar essa informação entre a tag HTML de parágrafo `<p>` `</p>` . Como essa última necessidade era de proposta geral, criamos uma função, a `decorador_p` , que recebe outra função `func` .

Internamente, criamos uma função encapsuladora, que vai receber um nome, pegar o resultado da função `func` e adicioná-lo entre as tags `<p>``</p>` . Note que a função principal está retornando somente o nome da função interna. Isso significa que uma função pode retornar uma outra função.

Em seguida, criamos o seguinte comando `info_em_p = decorador_p(info)` . Assim, passamos a função `info` para o `decorador_p` , ou seja, `inf_em_p` nada mais é que a função `info` com comportamento modificado, para que o texto de boas-vindas fique internamente na tag `<p></p>` . Dessa forma, o que fizemos foi decorar a função `info` para que ela tivesse um comportamento adicional.

Podemos então criar encapsulamentos de comportamentos para, por exemplo, criar funções que só podem ser acessadas depois que o usuário digitar uma senha. Mas o nosso decorador ainda não está completo. Antes de seguir, precisamos conhecer outras formas de passar argumentos para funções.

Com o objetivo de deixar o decorador com um caráter mais geral, podemos usar os operadores `*args` e `**kwargs` , de tal forma que a nossa função interna possa manipular funções arbitrárias:

```
>>> def info(nome):
...     return "Bem-vindo {0} ao nosso sistema".format(nome)
...
>>> def decorador_p(func):
...     def funcao_encapsulada(*args, **kwargs):
...         return "<p>{0}</p>".format(func(*args, **kwargs))
...         return funcao_encapsulada
...
>>> info_em_p = decorador_p(info)
>>> print(info_em_p("Eduardo"))
'<p>Bem-vindo Eduardo ao nosso sistema</p>'
```

No exemplo anterior, ao definirmos a função `funcao_encapsulada` , passamos os argumentos `*args` e `**kwargs` . Isso significa que os argumentos que essa função receberá, ou serão transformados em uma lista, ou em um

dicionário.

Já no caso da chamada da função `func`, ela indica que, se a lista `args` não for do tipo `None`, ela será decomposta em uma lista de argumentos. No caso em que o `kwargs` não for do tipo `None`, ele será decomposto em pares de argumento na forma par-valor.

Passar uma função como parâmetro para outra função, para depois referenciar a tal função a uma outra variável, pode ser tornar algo bastante confuso. Para esse tipo de tarefa – ou seja, criação de decoradores –, a linguagem Python oferece uma notação específica. Para isso, usamos o símbolo `@` seguido do nome da função decoradora.

No exemplo a seguir, criaremos a função decoradora chamada `decorador_p`. Em seguida, queremos que a função `info` adquira também o comportamento definido pela nossa função decoradora:

```
>>> def decorador_p(func):
...     def funcao_encapsulada(*args, **kwargs):
...         return "<p>{0}</p>".format(func(*args, **kwargs))
...     return funcao_encapsulada
...
>>> @decorador_p
>>> def info(nome):
...     return "Bem-vindo {0} ao nosso sistema".format(nome)
...
>>> print(info("Eduardo"))
'<p>Bem-vindo Eduardo ao nosso sistema</p>'
```

Usamos o `@decorador_p` antes da definição da função `info` para informar que queremos que essa função, além da propriedade original, retorne o texto de boas-vindas. Também queremos que esse texto esteja dentro da tag `<p></p>`, que é um aspecto da

função decorador_p .

Podemos aplicar mais de um decorador a uma mesma função:

```
>>> def decorador_p(func):
...     def funcao_encapsulada(*args, **kwargs):
...         return "<p>{0}</p>".format(func(*args, **kwargs))
...     return funcao_encapsulada
...
>>> def decorador_div(func):
...     def funcao_encapsulada(*args, **kwargs):
...         return "<div>{0}</div>".format(func(*args, **kwargs))
...     return funcao_encapsulada
...
>>> @decorador_div
... @decorador_p
... def info(nome):
...     return "Bem-vindo {0} ao nosso sistema".format(nome)
...
>>> print(info("Eduardo"))
'<div><p>Bem-vindo Eduardo ao nosso sistema</p></div>'
```

Aqui vale ressaltar que a ordem em que aplicamos os decoradores importa. Se tivéssemos mudado a ordem para @decorador_p , e depois aplicado @decorador_div ao rodar o comando print(info("Eduardo")) , a saída seria: '<p><div>Bem-vindo Eduardo ao nosso sistema</div></p>' .

Também podemos passar argumentos para decoradores:

```
>>> def tags(nome_da_tag):
...     def decorador_tag(func):
...         def funcao_encapsulada(*args, **kwargs):
...             return "<{0}>{1}</{0}>".format(nome_da_tag, func(
... args, **kwargs))
...         return funcao_encapsulada
...     return decorador_tag
...
>>> @tags("p")
... def info(nome):
...     return "Bem-vindo {0} ao nosso sistema".format(nome)
...
...
```

```
>>> print(info("Eduardo"))
'<p>Bem-vindo Eduardo ao nosso sistema</p>'
```

Nesse caso, tivemos de colocar mais um nível em nossa camada. Criamos a função `tags`, que recebe `nome_da_tag`, e, a parte mais interna, mudamos a função `funcao_encapsulada`, que vai retornar uma nova string, que terá o nome da tag que foi passada ao decorador dentro do sinal `<>` e `</>`.

Por fim, decoramos a função `info` com o `@tag("p")`. Indicamos que queremos decorar nossa mensagem de boas-vindas com a tag de parágrafo do HTML. Usaremos bastante decoradores ao trabalhar com Flask.

2.9 TRATAMENTO DE EXCEÇÕES

Muitas vezes, precisamos tratar possíveis erros de execução em um programa. Por exemplo, imagine que estamos criando uma calculadora. Se o usuário fizer uma divisão por zero, nosso código vai levantar um erro e parar de executar. Para evitar que o programa simplesmente pare de funcionar, podemos fazer um tratamento dessa exceção. Para isso, usamos os comandos `try/except/finally`. Como exemplo, vamos tomar o caso de divisão por zero:

```
try:
    print( 1 / 0)
except:
    print("Não é possível fazer divisão por zero")
```

No caso do código anterior, ao tentar imprimir a divisão de um por zero, um erro será levantado e o bloco logo abaixo do `try` será interrompido, passando a execução para o bloco pertencente ao `except`. Assim, será exibida a mensagem "Não é possível

fazer divisão por zero" .

Podemos ter exceções aninhadas. Por exemplo, podemos criar uma função que fará a leitura de dados em um arquivo.

```
def leitura_de_arquivo(nome_arquivo):
    nlines = 0
    try:
        arq = open(nome_arquivo)
        try:
            l1 = arq.readline()
            nlines += 1
            l2 = arq.readline()
            nlines += 1
        finally:
            print("Erro na leitura da linha {} do arquivo".format
(nlines + 1))
            arq.close()
    except Exception, e:
        print("Erro ao abrir o arquivo.")
        print("O erro foi:")
        print(str(e))
```

No código anterior, criamos a função `leitura_de_arquivo` que recebe o nome do arquivo (`nome_arquivo`). Em seguida, criamos o contador `nlines` para marcar as linhas do arquivo que já foram lidas e adicionamos um tratamento de exceções. Caso algum erro ocorra ao tentar ler o arquivo, o bloco `except` será executado, sendo que o tipo de erro será armazenado na variável `e` . Nesse caso, as mensagens "Erro ao abrir o arquivo." e "O erro foi:" serão exibidas, seguidas de uma mensagem com o tipo de erro encontrado.

Observe que dentro do primeiro bloco, marcado pelo `try` , logo abaixo de `nlines` , temos um segundo tratamento de exceções aninhado. Nesse segundo bloco, se algum erro ocorrer, ao ler uma linha do arquivo, o bloco `finally` será executado,

informando em qual linha ocorreu o erro, através da mensagem "Erro na leitura da linha {} do arquivo".format(nlines + 1) . O último comando do bloco `finally` faz o fechamento do arquivo (`arq.close()`).

2.10 CONCLUSÃO

Neste capítulo, foi apresentada uma breve introdução à linguagem Python e aos seus principais elementos. Para realmente aprender a desenvolver em uma linguagem de programação, é preciso trabalhar com projetos.

Um grupo de projetos interessante e que ajuda bastante a aprimorar os conceitos de lógica de programação é o desenvolvimento de jogos. Tendo isso em mente, no próximo capítulo, será apresentado o passo a passo do desenvolvimento de um jogo simples, o Jogo da Velha.

O objetivo principal do próximo capítulo será o de apresentar, de forma prática, os conceitos que foram apresentados ao longo deste capítulo. Assim, o leitor poderá começar efetivamente a escrever seus próprios programas.

APRENDENDO NA PRÁTICA: CRIANDO JOGOS

Um programa em Python nada mais é que um arquivo de texto contendo código. Para indicar que o arquivo é um programa, usamos a extensão `.py`. Os arquivos `.py` podem conter classes, funções e constantes. O código-fonte dos nossos programas ficaram nesses arquivos, os quais são chamados de módulos.

Além dos módulos que podemos criar, o Python vem com uma série de outros módulos embutidos, que facilitam muito a vida do desenvolvedor. Para consultar a lista completa desses módulos e como usá-los é recomendado sempre acessar a documentação da biblioteca padrão, disponível em: <https://docs.python.org/3/library/index.html>.

Uma prática comum, ao se escrever um programa em Python, é adicionar na primeira linha do arquivo um comando específico, que indica aos sistemas do tipo Linux e Unix, que o nosso arquivo é um script que pode ser executado diretamente no terminal:

```
#!/usr/bin/env python3.6
```

Nessa linha, que é um comentário em Python, e que começa com `#!`, o sistema vai olhar esse comentário e ver se estamos indicando um interpretador válido para o nosso script. Nesse caso, o que estamos fazendo é indicar que o sistema deve executar o comando `env`, que está na pasta `/usr/bin`, passando para esse comando `python3.6` como argumento. Esse comando vai executar a máquina virtual Python que está registrada na variável de ambiente do sistema.

Já na segunda linha do arquivo, é comum adicionar a informação sobre qual codificação de texto estamos usando em nosso programa, para evitar problemas de portabilidade, principalmente em versões do Python 2.X.

Por exemplo, ao se escrever um código em um editor que usa a codificação ISO-8859-15, no Windows, e em seguida tentar executar esse mesmo código em uma máquina rodando Linux, o interpretador poderá não reconhecer acentuações, gerando erro. Para manter essa portabilidade, deixamos identificado na segunda linha do nosso programa qual a codificação usada, tal como no exemplo a seguir:

```
# -*- coding: iso-8859-15 -*-
```

Uma outra boa prática é adicionar na terceira linha do arquivo um comentário, em mais de uma linha, informando sobre do que se trata o nosso programa, além de incluirmos nesse momento informações sobre licença e dados do desenvolvedor. Esse texto é chamado de **docstring**.

```
"""
Jogo da Velha.
Programa sob licença GNU V.3
"""
```

Logo após o **docstring** é que fazemos as importações dos módulos externos, que serão utilizados pelo nosso programa, usando os comando `import` e `from import`.

Módulos são como bibliotecas, que contêm trechos de códigos que podem ser utilizados por outros programas. Cada programa em Python é um módulo por si só, ou seja, podemos criar um arquivo `.py`, adicionar funções e constantes nesse arquivo para serem usados em outros lugares.

Um dos grandes pontos do Python é que ele possui uma série de módulos embutidos no próprio interpretador: os chamados módulos nativos. Por exemplo, suponha que queremos adicionar ao nosso jogo ações que são aleatórias, como jogar dados. Imagine que queremos simular o lançamento de um dado, que deve retornar um número entre 1 e 6 de forma aleatória. Nesse caso, podemos usar a função `randint` do módulo nativo `random`:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Jogo de dados
Programa sob licença GNU V.3
Desenvolvido por: E. S. Pereira
Versão 0.0.1
"""

from random import randint

print("Jogo de Dados")
print("Teste sua sorte")

while(True):
    numero = int(input("Escolha um número: "))
    dado = randint(1, 6)
    if dado == numero:
        print("Parabéns, saiu o seu número no dado.")
    else:
```



```

print("Não foi dessa vez.")
print("O número sorteado foi: ")
print(dado)
print("Deseja tentar a sorte novamente?")
continuar = input("Digite S para continuar ou N para sair: ")

if continuar == 'N' or continuar == 'n':
    break

```

Primeiro, vamos analisar, de forma breve, o que faz o código anterior, porém, para o jogo que vamos desenvolver, vamos apresentar todas as suas funcionalidades de forma detalhada e passo a passo.

No exemplo anterior, logo abaixo do **docstring**, importamos a função `randint` do módulo `random`. Em seguida usamos a função `print` para apresentar na tela informações sobre o nosso programa para o usuário.

Ao desenvolver jogos, veremos que um ponto central é representado pelo condicional `while` com o parâmetro `True`, que representa um laço, ou loop, infinito. Esse laço será executado de forma repetida, criando um programa que só será interrompido por uma ação do usuário.

O próximo passo foi o de pedir que o usuário digitasse um número. Para isso, usamos a função `input`, que recebe como parâmetro o texto a ser exibido na tela. Essa função pega o valor digitado pelo usuário e converte em uma string.

Como queremos um número, precisamos usar a função `int`, que converte a string digitada em um número inteiro. Em seguida, usamos a função `randint` para gerar um número aleatório entre 1 e 6, sendo que seu valor é armazenado variável `dado`.

O próximo passo foi o de utilizar o condicional `if/else` para comparar o valor digitado pelo usuário com o valor gerado pelo nosso 'dado'. Se os dois forem iguais, aparece na tela a informação de que o número escolhido pelo usuário foi o mesmo que o sistema gerou; do contrário, informamos que o jogador perdeu e mostramos qual foi o número sorteado.

Na última parte do programa, perguntamos se o usuário deseja jogar novamente. Se ele digitar `n` ou `N` o condicional final `if` será acionado e o operador `break` será ativado, interrompendo o loop infinito.

Pronto! Com esta base de conhecimentos, já podemos seguir para o nosso jogo. Na próxima seção vamos iniciar a criação de um jogo da velha, no modo terminal.

3.1 JOGO DA VELHA

Nesse jogo, o jogador escolherá entre `x` e `o` e em seguida tentará alinhar as três strings que escolheu em qualquer direção. Vamos criar um arquivo chamado `jogo_da_velha.py` e escrever nosso programa.

No seção anterior vimos como usar as funções `print` e `input` para criar interação com o usuário. Porém, para se criar um jogo, no modo terminal, elas não são as mais indicadas, pois não temos muito controle sobre a posição em que queremos 'desenhar' no terminal, tal como definir posição exata de um texto. Em um jogo, cada ação do jogador deve permitir uma construção dinâmica da tela e, dessa forma, mesmo que seja apenas um jogo em terminal, precisamos de uma ferramenta que nos dê um maior

poder para isso.

Para esse projeto vamos usar o módulo `curse`, que fornece ferramentas para a criação de terminais, independente de plataforma. Nosso jogo será no estilo dos primórdios da computação e isso ajudará a entender os conceitos fundamentais de programação.

Tecnicamente, o jogo da velha consiste basicamente de uma matriz, no qual um jogador marca um **x** e o outro jogador marca um **o**. O objetivo do jogo é conseguir marcar uma sequência de três **x** (ou **o**) nas diagonais, verticais ou horizontais. Vamos representar o espaço em que o jogo ocorre em uma lista formada por três outras listas, cada uma com três elementos. Para não ficar repetindo código e para poder organizar o nosso programa, serão criadas várias funções.

O resultado final será algo parecido com a seguinte tela:

```
Bem-vindo ao Jogo da Velha.
Pressione q para sair e h para obter ajuda.

O x venceu...
Pressione Y para jogar novamente ou Q para sair.

      | |X
      -----
      o|x|o
      -----
      x| |

Desenvolvido por: E. S. Pereira.
Licença Nova BSD.
```

Figura 3.1: Jogo da velha em modo terminal.

Para usuários do sistema Windows

Para esse jogo, vamos usar a biblioteca *curse*, que fornece opções de construção de aplicações, de modo terminal, com funcionalidades de desenhar em tela, exibir textos e manipulação de entradas de teclado, independentemente de sistema operacional. Com ela, podemos criar aplicações de terminal extremamente portáteis. Para saber mais veja o site com a documentação dessa biblioteca: <https://docs.python.org/3/howto/curses.html>.

O *curse* já vem pré-instalado nos sistemas Unix e Linux, porém não é um módulo nativo no Windows. Para rodar o jogo da velha no Windows é preciso instalá-lo, o que vai depender da versão do Python instalado na sua máquina, 32 ou 64 bits.

Os arquivos para instalar o *curse* no Windows estão disponíveis em: <http://www.lfd.uci.edu/~gohlke/pythonlibs/#curses>

Para instalar a versão para o Python 32 bits use o comando

```
pip install curses-2.2-cp36-cp36m-win32.whl
```

Para versões 64 bit

```
pip install curses-2.2cp36-cp36m-win_amd64.whl
```

Alternativamente pode-se tentar o comando:

```
python -m pip install curses-2.2cp36-cp36m-win_amd64.whl
```

Realizada instalação do módulo extra podemos seguir com o projeto.

Criando o programa

Vamos criar um arquivo chamado `jogo_da_velha.py` , sendo que as primeiras linhas do nosso arquivo serão informações padrões, incluindo a *docstring* do nosso programa. Também vamos importar as funções necessárias para criar o nosso programa:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Um jogo da velha simples.

Licença:

Copyright 2017 E. S. Pereira
"""

from curses import initscr, wrapper
from random import randint
```

Nesse caso, estamos importando as funções `initscr` e `wrapper` do módulo `curses` . A função `initscr` será usada apenas para inicializar o nosso terminal. Já a função `wrapper` será usada para encapsular a função principal do jogo, para que possamos ter acesso às facilidades desse módulo, como permitir imprimir um determinado texto na tela, em uma posição específica, tal como ocorre nas telas de jogos.

Vamos criar a função principal, `main` . Essa função receberá como entrada um objeto que representa a tela principal do nosso programa. Vamos aproveitar e já escrever nosso programa de modo a poder reaproveitar nossas funções em outros projetos. Nesse caso, a primeira versão do nosso programa terá:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Um jogo da velha simples.
"""
```

```
from curses import initscr, wrapper
from random import randint
```

```
def main(stdscr):
    pass
```

```
if __name__ == "__main__":
    initscr()
    wrapper(main)
```

Criamos a função `main` que recebe com parâmetro a variável `stdscr`, o qual representa a nossa tela padrão (*standard screen* - `stdscr`). Em seguida criamos um bloco de código que só deverá ser executado se o nosso arquivo for o programa principal, representado pelo bloco definido pelo condicional `if __name__ == "__main__":`.

A declaração `initscr()` faz a inicialização da nossa tela padrão, ou seja, ela é usada para inicializar o terminal do nosso jogo. Já a função `wrapper(main)` está recebendo como argumento a função principal do jogo, `main`. Essa última chamada está indicando para o módulo `curses` que quem vai fazer a manipulação do terminal do jogo é a função `main`. Isto é, o módulo `curses` vai gerar a tela padrão, representada pela variável `stdscr`, e deixará a cargo da função `main` a manipulação dessa nova tela.

3.2 DESENHANDO A TELA DO JOGO

Vamos começar desenhando nossa tela. Vamos criar uma borda para a nossa tela e criar duas funções para exibir uma mensagem de boas-vindas e os créditos:

```
def boas_vindas(stdscr):
    stdscr.addstr(1, 1, "Bem-vindo ao Jogo da Velha.")
    stdscr.addstr(2, 1, "Pressione q para sair ou h para obter aj
uda.")
    stdscr.addstr(16, 1, "Desenvolvido por: E. S. Pereira.")
    stdscr.addstr(17, 1, "Licença Nova BSD.")

def main(stdscr):
    stdscr.clear()
    stdscr.border()
    boas_vindas(stdscr)
    stdscr.refresh()

    while True:
        pass
```

Na função `main`, o primeiro comando foi usado para limpar a tela, através do comando `stdscr.clear()`. Mais adiante veremos que essa notação é típica de acesso a métodos de objetos. Por hora é importante saber que estamos usando o `.` para poder acessar funções que fazem parte do terminal representado pela variável `stdscr`. O comando `stdscr.border()` cria um retângulo, ou borda, em torno da nossa tela. Note que a função `boas_vindas` também recebe o terminal como entrada. Finalmente, usamos o comando `stdscr.refresh()` para indicar que queremos que todas as operações que fizemos no terminal possam ser exibidas na tela do usuário. Outro ponto importante na função `main` é que nela vamos adicionar um dos elementos mais importantes dos jogos. O coração de todo jogo é o laço infinito que só será interrompido quando o jogador quiser encerrar o jogo. Adicionamos esse laço dentro da função `main` com o comando `while(True)`.

Na função `boas_vindas`, usamos a função `addstr` da tela que estamos trabalhando. O próprio nome sugere que essa função

é usada para adicionar (`add`) uma string (`str`) na tela. O primeiro parâmetro que passamos é a posição `y` em que queremos exibir o início do texto; o segundo parâmetro é a posição `x` , e em seguida passamos o texto em si. O topo da tela é a posição `y = 0` . A regra é: à medida que descemos na tela, o valor de `y` cresce. As distâncias `x` e `y` são medidas em termos de caracteres e não em dimensões espaciais.

O resultado dessa primeira versão será a seguinte tela:

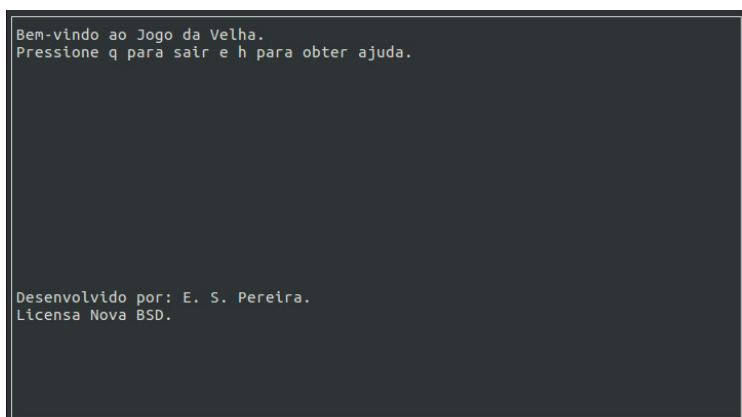


Figura 3.2: Jogo da velha em modo terminal.

Note a posição em que o texto se encontra. Veja que a mensagem de crédito ficou mais próxima da base da nossa janela, pois usamos um maior valor para a posição y .

3.3 INTERAÇÃO COM O USUÁRIO

Vamos agora mapear as teclas que o usuário está digitando, usando o comando `getkey`. Também vamos criar uma lista para identificar que algo deverá ser realizado somente se uma das teclas

de interesse for pressionada. Esse mapeamento deverá ocorrer a cada instante. Assim vamos começar a preencher o conteúdo do bloco do laço dentro de `main` :

```
def ajuda(stdscr):
    stdscr.clear()
    stdscr.border()
    stdscr.addstr(1, 1, "Pressione Q para sair ou H para exibir e
ssa ajuda.")
    stdscr.addstr(2, 1, "Para mudar a posição, use as teclas: A,
D, S, W")
    stdscr.addstr(3, 1, "Para definir uma posição do jogo, digite
: L")
    stdscr.addstr(4, 1, "Para reiniciar a partida, digite: Y")
    stdscr.addstr(5, 1, "Pressione espaço para sair dessa tela.")
    stdscr.refresh()

def reiniciar_tela(stdscr):
    stdscr.clear()
    stdscr.border()
    boas_vindas(stdscr)
    stdscr.refresh()

def main(stdscr):

    reiniciar_tela(stdscr)

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break
        if entrada == 'h':
            ajuda(stdscr)
        else:
            boas_vindas(stdscr)
```

Nesse ponto, foi criada uma função que exibe informações de ajuda. Nessa função, toda a informação atual da tela será apagada e somente as informações da ajuda serão exibidas. Outra coisa que foi modificada é que criamos a função `reiniciar_tela` . Com ela, vamos reiniciar a nossa tela para o modo de entrada sempre

que for necessário.

Dentro do laço, na função `main`, adicionamos a variável `entrada`, que conterá a informação da tecla que for digitada pelo usuário, através da declaração `stdscr.getkey()`. Dentro do bloco `while(True)` será definido o escopo de execução do jogo. Assim, precisamos já definir algumas ações básicas, como a função `ajuda` e a opção de finalizar o jogo. A variável `entrada` armazena o valor da tecla que o usuário digitar. Nesse caso, se o usuário digitar a tecla `q` o operador `break` será executado e o loop principal do jogo será interrompido. Isso fará com que o jogo termine. Agora, se o usuário digitar a tecla `h`, a função de ajuda será executada, apresentando na tela informações sobre o jogo. Agora, se o usuário digitar qualquer outra tecla, a função `boas_vindas` será executada.

Dessa forma, estamos definindo as interações básicas do usuário com o nosso jogo.

```
Pressione Q para sair ou H para exibir essa ajuda.  
Para mudar a posição, use as teclas: A, D, S, W  
Para definir uma posição do jogo, digite: L  
Para reiniciar a partida, digite: Y  
Pressione espaço para sair dessa tela.█
```

Figura 3.3: Tela de ajuda do jogo.

3.4 O TABULEIRO

Em um jogo, a todo o instante a tela é redesenhada, ou renderizada em tempo real. A velocidade, ou taxa, de renderização da tela interfere na experiência do jogo. No nosso caso, cada vez que o loop principal é executado, nossa tela será redesenhada. Porém, se o usuário digitar `h`, para ajuda, e em seguida voltar à tela principal do jogo, vamos precisar armazenar as informações atuais de alguma forma. Ou seja, precisamos armazenar o estado corrente da tela. Para isso, criaremos a região do tabuleiro e modificaremos a função `reiniciar_tela`. Assim poderemos armazenar o estado atual da tela, além de poder desenhar novos textos sobre o texto de boas-vindas.

```
def reiniciar_tela(stdscr, limpar=True):
    if limpar is True:
        stdscr.clear()
        stdscr.border()
        boas_vindas(stdscr)
        stdscr.refresh()

def tabuleiro(stdscr, posicoes, x_center):
    stdscr.clear()
    reiniciar_tela(stdscr, limpar=False)

    stdscr.addstr(10, x_center - 3, "-----")
    stdscr.addstr(12, x_center - 3, "-----")
    i = 9
    for linha in posicoes:
        tela = "%s|%s|%s " % tuple(linha)
        stdscr.addstr(i, x_center - 3, tela)
        i += 2
```

Como `True` é uma variável booleana, uma forma mais clara de comparar se uma variável é verdadeira ou não é através da expressão `is`. No caso da função `reiniciar_tela` usamos a declaração `if limpar is True:` para verificar se a variável

`limpar` é `True` ou `False` . Observe também que na declaração da função já indicamos que o valor padrão de `limpar` deve ser `True` . Isso significa que, ao chamar essa função, não precisamos passar todos os parâmetros, e sim somente aqueles que não tiverem valores predefinidos.

A função `tabuleiro` recebe o terminal, a lista `posicoes` , que vai armazenar informações da jogada atual, e o `x_center` , que representará o centro do terminal. Dentro dessa função, a primeira coisa a ser executada é a limpeza da tela. Em seguida, fizemos a chamada da função `reiniciar_tela` , passando o parâmetro `limpar` como `False` . Estamos 'desenhando' linhas horizontais nas posições `y = 10` e `y = 12` . Ambas começarão na posição central menos três caracteres.

Nosso tabuleiro vai começar efetivamente na posição `y = 9` , definido pela variável `i=9` . Em seguida foi criado um laço sobre os elementos da lista `posicoes` . Dentro desse laço criamos a variável `tela` , que consiste de uma string contendo as informações de cada elemento da sublista que faz parte da lista `posicoes` . Como desenhamos linhas horizontais, fizemos com que o conteúdo da jogada fosse desenhado de dois em dois na posição `y` , usando para isso o `i += 2` .

Agora modificaremos a função `main` para incluir as funcionalidades criadas anteriormente:

```
def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
```

```

while True:
    entrada = stdscr.getkey()
    if entrada == 'q':
        break

    if entrada == 'h':
        ajuda(stdscr)
    else:
        tabuleiro(stdscr, posicoes, x_center)

```

Aqui foi utilizada a função `stdscr.getmaxyx()` para obter a altura e largura do terminal. Essa função retorna uma tupla. Como estamos interessados apenas na largura da tela, usamos o `[1]` ao final do comando para guardar apenas a informação necessária. Em seguida, calculamos o centro como sendo a divisão inteira, por dois, da largura menos um, feito usando o operador de divisão inteira `//`. Em seguida criamos a lista `posicoes` contendo apenas strings que representam espaços em branco. Ao final modificamos o condicional de ajuda para que, quando o usuário digitar qualquer outra tecla, o nosso tabuleiro seja exibido na tela. O resultado dessa versão do código será:

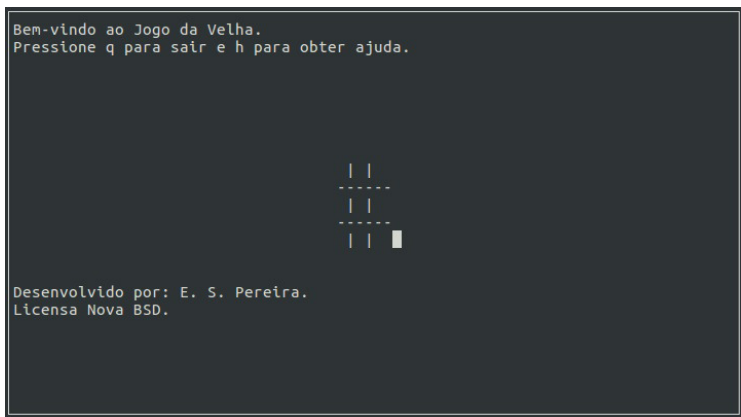


Figura 3.4: Desenho do tabuleiro do jogo.

3.5 MOVENDO AS PEÇAS DO JOGO

Agora precisamos fazer nosso cursor mover dentro do tabuleiro. No nosso programa vamos usar as teclas: `w` e `s` para mover o cursor para cima e para baixo; `a` e `d` para mover o cursor para a esquerda e para direita; a tecla `Enter` do teclado principal para indicar que queremos marcar um `x` no local.

Antes de mais nada, vamos criar duas funções para mapear o espaço do jogo em si:

```
def limites(pos_x, pos_y):
    if pos_x > 2:
        pos_x = 0
    if pos_x < 0:
        pos_x = 2

    if pos_y > 2:
        pos_y = 0

    if pos_y < 0:
        pos_y = 2

    return pos_x, pos_y

def espaco_do_tabuleiro(pos_x, pos_y, entrada):
    if entrada == 'a':
        pos_x, pos_y = limites(pos_x - 1, pos_y)
    elif entrada == 'd':
        pos_x, pos_y = limites(pos_x + 1, pos_y)
    elif entrada == 's':
        pos_x, pos_y = limites(pos_x, pos_y + 1)
    elif entrada == 'w':
        pos_x, pos_y = limites(pos_x, pos_y - 1)
    else:
        pass

    return pos_x, pos_y
```

A primeira função, `limites`, está apenas verificando se o

jogador está se movendo dentro do espaço do tabuleiro, definido por uma matriz 3x3, ou seja, cada vez que ele chegar nos limites, a posição deverá levada para o lado oposto. Já a função `espaco_do_tabuleiro` vai receber a posição corrente e a tecla digitada pelo usuário.

Note que usamos um conjunto de `if` s para atualizar os valores de `pos_x` e `pos_y` de acordo com a direção que cada tecla representa. Por exemplo, a tecla `a` representa mover para esquerda, dessa forma foi subtraído o valor 1 da posição `x`, ao mesmo tempo que estamos verificando se essa subtração não vai resultar em um valor menor do que o espaço definido do nosso jogo.

Como já temos as informações de como devemos atualizar as posições relativas de `pos_x` e `pos_y`, podemos agora criar uma função que usa essa informação para mover o cursor no terminal para a posição equivalente na tela.

```
def cursor(stdscr, pos_x, pos_y, x_center):

    cursor_y = 9
    cursor_x = x_center - 3
    if pos_y == 1:
        cursor_y += 2

    if pos_y == 2:
        cursor_y += 4

    if pos_x == 1:
        cursor_x += 2

    if pos_x == 2:
        cursor_x += 4

    stdscr.move(cursor_y, cursor_x)
```

A atualização do cursor na tela foi feita usando o comando `stdscr.move(cursor_y, cursor_x)`.

Vamos modificar novamente a função `main`:

```
def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if(entrada in ['a', 's', 'w', 'd']):
            pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y, entrada)

        if entrada == 'h':
            ajuda(stdscr)
        else:
            tabuleiro(stdscr, posicoes, x_center)
            cursor(stdscr, pos_x, pos_y, x_center)
```

As modificações realizadas foram: adição das variáveis `pos_x` e `pos_y`, inicializadas com valor 0; adição do condicional que mapeará as teclas digitadas, dentro do laço infinito; e atualização das variáveis de posição de acordo com a tecla digitada. Ao final, logo abaixo da chamada da função `tabuleiro`, fizemos a chamada da função `cursor`. É extremamente importante ressaltar que a ordem dessa chamada vai alterar o resultado do programa. Isso devido ao fato de que nossa tela está sendo desenhada, ou renderizada, a cada passo do laço. O cursor sempre

fica no último elemento desenhado. Assim, para que a nossa função `cursor` funcione, ela sempre deve ser a última a ser chamada, logo após o desenho de todo o resto da tela.

Na figura a seguir temos o curso piscando no centro do tabuleiro:

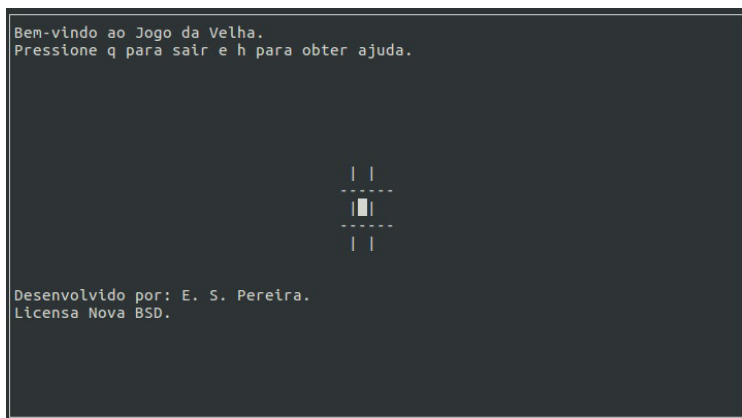


Figura 3.5: Cursor movendo-se para o centro do tabuleiro após adicionar a função `cursor`.

3.6 MARCANDO O X DA JOGADA

O próximo passo será o de definir a ação do jogador, ou seja, quando ele pressionar a tecla `Enter` num espaço vazio do tabuleiro, lá deverá ser marcado com um `x` :

```
def jogador(pos_x, pos_y, posicoes):  
    if posicoes[pos_y][pos_x] == " "  
        posicoes[pos_y][pos_x] = "x"  
    return posicoes
```

O que essa função fará é verificar se na lista `posicoes` tem um espaço em branco e, se for verdade, vai substituir esse espaço por um `x` .

A função principal ficará assim:

```
def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ']]
]
pos_x, pos_y = 0, 0

while True:
    entrada = stdscr.getkey()
    if entrada == 'q':
        break

    if entrada in ['a', 's', 'w', 'd']:
        pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y, entrada)

    if entrada == "\n":
        posicoes = jogador(pos_x, pos_y, posicoes)

    if entrada == 'h':
        ajuda(stdscr)
    else:
        tabuleiro(stdscr, posicoes, x_center)
        cursor(stdscr, pos_x, pos_y, x_center)
```

A tecla `Enter` do teclado principal é mapeada pelo gerador de novas linhas `\n`. Logo, toda vez que o usuário pressionar `Enter`, a função `jogador` será executada.

```
Bem-vindo ao Jogo da Velha.  
Pressione q para sair e h para obter ajuda.  
  
  x|x|x  
  ----  
  x|x|x  
  ----  
  x|x|x  
  
Desenvolvido por: E. S. Pereira.  
Licença Nova BSD.
```

Figura 3.6: Marcando x no tabuleiro usando a tecla Enter.

3.7 CRIANDO UMA INTELIGÊNCIA ARTIFICIAL

Nesse exemplo vamos usar a função `randint` para criar um jogador robô. Aqui faremos com que o robô seja executado após a jogada do jogador. Fica para o leitor experimentar novas possibilidades e incluir a opção de mudar quem deve jogar primeiro, fazendo um *cara ou coroa* antes de cada partida, usando, por exemplo, a ideia do jogo de jogar dados, apresentado no início desse capítulo.

Nossa inteligência artificial será dada pela seguinte função:

```
def robo(posicoes):  
  
    vazias = []  
    for i in range(0, 3):  
        for j in range(0, 3):  
            if posicoes[j][i] == " ":  
                vazias.append([j, i])  
  
    n_escolhas = len(vazias)
```

```

if n_escolhas != 0:
    j, i = vazias[randint(0, n_escolhas - 1)]
    posicoes[j][i] = "o"

return posicoes

```

A função `robo` vai receber a lista contendo as posições da jogada corrente. Em seguida, criaremos uma nova lista contendo as posições que ainda estão vazias. Para isso usamos dois laços aninhados. Podemos pensar na lista `posicoes` como uma matriz. Nosso laço vai correr sobre cada linha e em seguida sobre cada coluna. Quando a linha e coluna dessa matriz estiver com o espaço em branco, o vetor `vazias` guardará a posição da linha e coluna. Em seguida, criamos a variável `n_escolhas`, contendo o número total de espaços vazios para fazer novas jogadas. Se ainda sobrar espaço, usamos o `randint` para fazer uma escolha aleatória dentro de todas essas possibilidades. Por fim, atualizamos a posição escolhida na matriz `posicoes` com o valor da string `o`.

```

def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if entrada in ['a', 's', 'w', 'd']:
            pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y, entrada)

        if entrada == "\n":
            posicoes = jogador(pos_x, pos_y, posicoes)
            posicoes = robo(posicoes)

```

```

if entrada == 'h':
    ajuda(stdscr)
else:
    tabuleiro(stdscr, posicoes, x_center)
    cursor(stdscr, pos_x, pos_y, x_center)

```

```

Bem-vindo ao Jogo da Velha.
Pressione q para sair e h para obter ajuda.

      x|o|o
      ----
      x|x|o
      ----
      x|o|x

Desenvolvido por: E. S. Pereira.
Licença Nova BSD.

```

Figura 3.7: Adicionando máquina versus jogador.

Na última modificação do jogo, cada vez que o jogador pressionar `Enter` o robô realizará uma jogada. Porém, se o jogador digitar `Enter` em um espaço que já está marcado, sua jogada não será computada, mas o robô imediatamente tentará uma outra jogada. Para resolver esse problema vamos modificar a função `jogador` para que ela retorne também `True` quando uma jogada for efetivamente realizada.

```

def jogador(pos_x, pos_y, posicoes):
    if posicoes[pos_y][pos_x] == " ":
        posicoes[pos_y][pos_x] = "x"
        return True, posicoes
    return False, posicoes

```

Note que a função só retornará `True` quando for encontrado um espaço vazio, o qual é possível substituir por um `x`. Na função

main , vamos adicionar um condicional que fará com que a função do robô só seja chamada quando a função jogador retornar verdadeiro. Observe que, em Python, uma função pode retornar mais de um valor ao mesmo tempo. Nesse caso, a saída da função será uma tupla. Atualizando a função main temos:

```
def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if entrada in ['a', 's', 'w', 'd']:
            pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y, entrada)

            if entrada == "\n":
                jogou, posicoes = jogador(pos_x, pos_y, posicoes)
                if jogou is True:
                    posicoes = robo(posicoes)

            if entrada == 'h':
                ajuda(stdscr)
            else:
                tabuleiro(stdscr, posicoes, x_center)
                cursor(stdscr, pos_x, pos_y, x_center)
```

Aqui criamos a variável jogou . Observe que usamos duas variáveis antes do igual, para a saída da função jogador . Agora o robô só poderá jogar quando o jogador efetivamente tiver feito uma nova marcação no tabuleiro.

3.8 DETERMINANDO O GANHADOR

O próximo passo é definir quem ganhou uma partida. No jogo da velha, temos as opções de ganhar ou empatar, o famoso "deu velha". O primeiro que completar sua marcação na horizontal, na vertical ou na diagonal ganha. Vamos criar duas listas para validar as diagonais. Também vamos tratar nossa lista de posições como uma matriz. Nesse caso, se a soma de uma das linhas contendo o mesmo elemento for igual a três, isso indica que alguém ganhou. Para facilitar a operação, vamos criar outra matriz que será a transposta da matriz de posições, ou seja vamos trocar linhas por colunas. A primeira parte do código ganhador será:

```
def ganhador(posicoes):
    diagonal1 = [posicoes[0][0], posicoes[1][1], posicoes[2][2]]
    diagonal2 = [posicoes[0][2], posicoes[1][1], posicoes[2][0]]

    transposta = [[], [], []]
    for i in range(3):
        for j in range(3):
            transposta[i].append(posicoes[j][i])
```

No caso da transposta, fizemos um laço aninhado no qual, para cada linha `i` da matriz `transposta`, indicamos que queremos adicionar um elemento `[j][i]` da matriz `posicoes`.

Vamos criar uma função auxiliar que retornará 'x' ou 'o' ou `None`. Vamos passar para ela uma lista que representará uma das linhas de nossa matriz. Essa função fará a contagem de quantos elementos do mesmo tipo tem na lista e retornará o símbolo que ocorre três vezes. Caso nenhum atenda a essa condição, a função retornará `None`:

```
def total_alinhado(linha):
    num_x = linha.count("x")
    num_o = linha.count("o")
```

```

if num_x == 3:
    return "x"
if num_o == 3:
    return "o"

return None

```

Com essa função auxiliar podemos continuar a função ganhador :

```

def ganhador(posicoes):
    diagonal1 = [posicoes[0][0], posicoes[1][1], posicoes[2][2]]
    diagonal2 = [posicoes[0][2], posicoes[1][1], posicoes[2][0]]

    transposta = [[], [], []]
    for i in range(3):
        for j in range(3):
            transposta[i].append(posicoes[j][i])

    gan = total_alinhado(diagonal1)
    if gan is not None:
        return gan

    gan = total_alinhado(diagonal2)

    if gan is not None:
        return gan

    velha = 9
    for i in range(3):

        gan = total_alinhado(posicoes[i])
        if gan is not None:
            return gan

        gan = total_alinhado(transposta[i])
        if gan is not None:
            return gan

    velha -= posicoes[i].count("x")
    velha -= posicoes[i].count("o")

```



```

if velha == 0:
    return "velha"

return None

```

Inicialmente verificamos se alguém completou uma das diagonais, passando os vetores com os conteúdos das diagonais para a função `total_alinhado`. Como o tabuleiro tem nove espaços de jogada, a estratégia para verificar se ninguém ganhou foi a de ir subtraindo a ocorrência de `x` e `o`. Se ao final não sobrou nenhum espaço em branco, mas ao mesmo tempo ninguém ganhou, a função retornará a string `"velha"`. Se nenhuma das condições anteriores for alcançada, a função retornará `None` e o jogo poderá continuar.

Na função `main`, vamos criar a variável `fim_de_partida` no qual guardaremos a informação que será retornada pela função `ganhador`.

```

def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    fim_de_partida = None

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if fim_de_partida is None:

            if entrada in ['a', 's', 'w', 'd']:
                pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y,

```

```
entrada)
```

```
    if entrada == "\n":
        jogou, posicoes = jogador(pos_x, pos_y, posicoes)
        fim_de_partida = ganhador(posicoes)
        if jogou is True and fim_de_partida is None:
            posicoes = robo(posicoes)

    if entrada == 'h':
        ajuda(stdscr)
    else:
        tabuleiro(stdscr, posicoes, x_center)
        cursor(stdscr, pos_x, pos_y, x_center)
```

Note que adicionamos um novo bloco dentro do laço principal do jogo. Nele colocamos o mapeador das entradas e as efetivações das jogadas. Esse novo bloco é definido pelo condicional `if fim_de_partida is None: .` Também adicionamos a chamada `fim_de_partida = ganhador(posicoes)` , que vai interromper o jogo caso alguém ganhe.

3.9 REINICIANDO A PARTIDA

Anteriormente, definimos um mecanismo para indicar o fim da partida, porém falta desenhar na tela um texto indicando o ganhador. Também precisamos de um mecanismo que permita começar uma nova partida sem ter que fechar e abrir o programa.

Vamos começar criando a função que deverá indicar um ganhador:

```
def fim_de_jogo(stdscr, vencedor):
    stdscr.addstr(6, 1, "O %s venceu..." % vencedor)
    stdscr.addstr(7, 1, "Pressione Y para jogar novamente ou Q pa"
ra sair.")
    stdscr.refresh()
```

Essa é uma função que recebe a tela do jogo e o nome do

vencedor. Ela vai desenhar na tela nas linhas 6 e 7 as informações que queremos. Veja que indicamos que, se o jogador quiser começar uma nova partida, bastará pressionar a tecla `y`. Então vamos colocar um condicional para reiniciar o jogo caso o usuário digite `y`. Também vamos verificar quem ganhou e exibir essas informações na tela. Com essas últimas adições, temos um jogo da velha completo. A versão final da função `main` será:

```
def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    fim_de_partida = None

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if fim_de_partida is None:

            if entrada in ['a', 's', 'w', 'd']:
                pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y,
entrada)

            if entrada == "\n":
                jogou, posicoes = jogador(pos_x, pos_y, posicoes)
                fim_de_partida = ganhador(posicoes)
                if jogou is True and fim_de_partida is None:
                    posicoes = robo(posicoes)

            if entrada == "y":
                fim_de_partida = None
                posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
                pos_x = 0
```

```

pos_y = 0

if entrada == 'h':
    ajuda(stdscr)
else:
    tabuleiro(stdscr, posicoes, x_center)
    if fim_de_partida is not None:
        fim_de_jogo(stdscr, fim_de_partida)
    cursor(stdscr, pos_x, pos_y, x_center)

```

No código anterior, adicionamos o bloco marcado por `if entrada == "y":`, com que voltamos as variáveis principais do jogo para seu estado inicial. Nas últimas linhas da função anterior, verificamos se a partida chegou ao fim; sendo isso verdade, exibimos o nome do ganhador na tela. Como a função `tabuleiro` sempre limpa a tela a cada vez que é chamada, é importante colocar o que vamos desenhar sobre a tela principal logo após a chamada dessa função, tal como fizemos para a função `cursor`.

Segue o código completo do jogo, lembrando que todos os códigos deste livro estão disponíveis no GitHub do autor: https://github.com/duducosmos/livro_python.

```

#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Um jogo da velha simples.
"""

from curses import initscr, wrapper
from random import randint

def boas_vindas(stdscr):
    stdscr.addstr(1, 1, "Bem-vindo ao Jogo da Velha.")
    stdscr.addstr(2, 1, "Pressione q para sair e h para obter ajuda.")
    stdscr.addstr(16, 1, "Desenvolvido por: E. S. Pereira.")
    stdscr.addstr(17, 1, "Licença Nova BSD.")

```

```

def ajuda(stdscr):
    stdscr.clear()
    stdscr.border()
    stdscr.addstr(1, 1, "Pressione Q para sair ou H para exibir e
ssa ajuda.")
    stdscr.addstr(2, 1, "Para mudar a posição, use as teclas: A,
D, S, W")
    stdscr.addstr(3, 1, "Para definir uma posição do jogo, digite
: L")
    stdscr.addstr(4, 1, "Para reiniciar a partida, digite: Y")
    stdscr.addstr(5, 1, "Pressione espaço para sair dessa tela.")
    stdscr.refresh()

def reiniciar_tela(stdscr, limpar=True):
    if limpar is True:
        stdscr.clear()
    stdscr.border()
    boas_vindas(stdscr)
    stdscr.refresh()

def tabuleiro(stdscr, posicoes, x_center):
    stdscr.clear()
    reiniciar_tela(stdscr, limpar=False)

    stdscr.addstr(10, x_center - 3, "-----")
    stdscr.addstr(12, x_center - 3, "-----")
    i = 9
    for linha in posicoes:
        tela = "%s|%s|%s " % tuple(linha)
        stdscr.addstr(i, x_center - 3, tela)
        i += 2

def limites(pos_x, pos_y):
    if pos_x > 2:
        pos_x = 2
    if pos_x < 0:
        pos_x = 0

    if pos_y > 2:
        pos_y = 2
    if pos_y < 0:
        pos_y = 0

```

```

    return pos_x, pos_y

def espaco_do_tabuleiro(pos_x, pos_y, entrada):
    if entrada == 'a':
        pos_x, pos_y = limites(pos_x - 1, pos_y)
    elif entrada == 'd':
        pos_x, pos_y = limites(pos_x + 1, pos_y)
    elif entrada == 's':
        pos_x, pos_y = limites(pos_x, pos_y + 1)
    elif entrada == 'w':
        pos_x, pos_y = limites(pos_x, pos_y - 1)
    else:
        pass

    return pos_x, pos_y

def cursor(stdscr, pos_x, pos_y, x_center):

    cursor_y = 9
    cursor_x = x_center - 3
    if pos_y == 1:
        cursor_y += 2

    if pos_y == 2:
        cursor_y += 4

    if pos_x == 1:
        cursor_x += 2

    if pos_x == 2:
        cursor_x += 4

    stdscr.move(cursor_y, cursor_x)

def jogador(pos_x, pos_y, posicoes):
    if posicoes[pos_y][pos_x] == " ":
        posicoes[pos_y][pos_x] = "x"
        return True, posicoes
    return False, posicoes

def robo(posicoes):

    vazias = []
    for i in range(0, 3):
        for j in range(0, 3):

```

```

        if posicoes[j][i] == " ":
            vazias.append([j, i])

n_escolhas = len(vazias)
if n_escolhas != 0:
    j, i = vazias[randint(0, n_escolhas - 1)]
    posicoes[j][i] = "o"

return posicoes

def ganhador(posicoes):
    diagonal1 = [posicoes[0][0], posicoes[1][1], posicoes[2][2]]
    diagonal2 = [posicoes[0][2], posicoes[1][1], posicoes[2][0]]

    transposta = [[], [], []]
    for i in range(3):
        for j in range(3):
            transposta[i].append(posicoes[j][i])

    gan = total_alinhado(diagonal1)
    if gan is not None:
        return gan

    gan = total_alinhado(diagonal2)

    if gan is not None:
        return gan

    velha = 9
    for i in range(3):

        gan = total_alinhado(posicoes[i])
        if gan is not None:
            return gan

        gan = total_alinhado(transposta[i])
        if gan is not None:
            return gan

        velha -= posicoes[i].count("x")
        velha -= posicoes[i].count("o")

    if velha == 0:
        return "velha"

```

```

    return None

def total_alinhado(linha):
    num_x = linha.count("x")
    num_o = linha.count("o")

    if num_x == 3:
        return "x"
    if num_o == 3:
        return "o"

    return None

def fim_de_jogo(stdscr, vencedor):
    stdscr.addstr(6, 1, "O %s venceu..." % vencedor)
    stdscr.addstr(7, 1, "Pressione Y para jogar novamente ou Q pa
ra sair.")
    stdscr.refresh()

def main(stdscr):
    reiniciar_tela(stdscr)
    width = stdscr.getmaxyx()[1]
    x_center = (width - 1) // 2
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
]
    pos_x, pos_y = 0, 0

    fim_de_partida = None

    while True:
        entrada = stdscr.getkey()
        if entrada == 'q':
            break

        if fim_de_partida is None:

            if entrada in ['a', 's', 'w', 'd']:
                pos_x, pos_y = espaco_do_tabuleiro(pos_x, pos_y,
entrada)

            if entrada == "\n":
                jogou, posicoes = jogador(pos_x, pos_y, posicoes)
                fim_de_partida = ganhador(posicoes)
                if jogou is True and fim_de_partida is None:

```



```

posicoes = robo(posicoes)

if entrada == "y":
    fim_de_partida = None
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
    pos_x = 0
    pos_y = 0

if entrada == 'h':
    ajuda(stdscr)
else:
    tabuleiro(stdscr, posicoes, x_center)
    if fim_de_partida is not None:
        fim_de_jogo(stdscr, fim_de_partida)
    cursor(stdscr, pos_x, pos_y, x_center)

if __name__ == "__main__":
    initscr()
    wrapper(main)

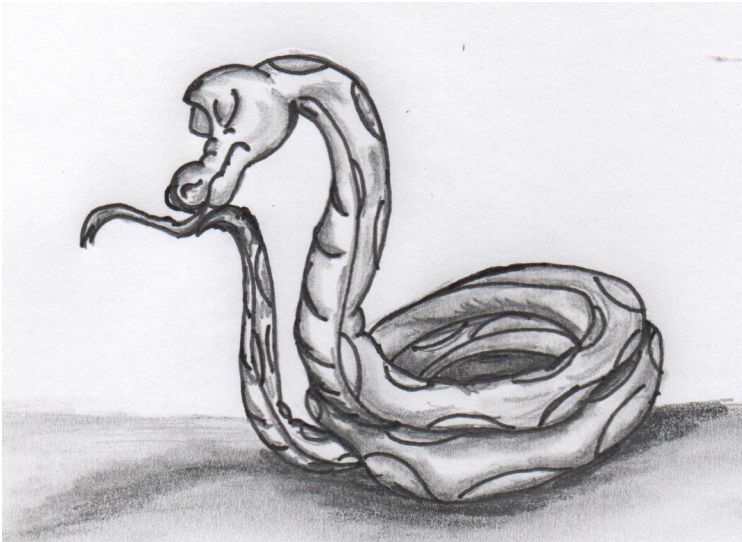
```

3.10 CONCLUSÃO

O objetivo principal deste capítulo foi o de mostrar que, usando apenas o conhecimento básico da linguagem e entendendo bem as estruturas condicionais e de laço, já é possível criar jogos eletrônicos. O nosso jogo remete à era inicial da computação, quando não se tinha muitos recursos gráficos. A partir desse ponto, o leitor já poderá se aventurar um pouco mais no universo da programação com Python. Minhas recomendações são de modificar o código anterior para deixá-lo com mais opções, chamar um amigo para jogar (ele pode usar as setas do teclado e o segundo `Enter` para fazer as jogadas). Outra sugestão é adicionar um placar, indicando quantas vezes cada jogador ganhou até o momento. Agora é usar a imaginação para criar coisas novas.

Paradigmas de programação

Na segunda parte do livro veremos que não existe uma forma universal de lógica de programação. Esses diferentes princípios, que usam conceitos lógicos distintos, são chamados de paradigmas de programação, tal como Programação Funcional, Orientação a Objetos e Programação Estruturada. Cada uma delas se mostrará mais adequada, de acordo como o problema que se tenta resolver. Dentro desse princípio, vamos ver alguns dos principais paradigmas que podem ser usados através da linguagem Python.



UMA LINGUAGEM, MUITOS PARADIGMAS

Quando se inicia no mundo da programação, acabamos escutando que devemos começar a estudar lógica de programação, existindo até alguns cursos com esse nome. O grande problema com isso é que fica parecendo que existe uma forma universal para se pensar, quando se trata de resolver problemas usando um computador. Mas a grande verdade é que isso não passa de um mito. Em geral, o que esses cursos mostram é, na verdade, a chamada Programação Estruturada. No capítulo anterior, ao apresentar o Jogo da Velha, nosso programa foi escrito usando o paradigma de Programação Estruturada. Em computação, um paradigma se refere a um modelo ou padrão, que deve ser adotado para escrever um programa de computador. Na Programação Estruturada, usamos os condicionais (`for` , `while` , `if/elif/else`), juntamente com funções, para criar estruturas de código que fazem uma determinada tarefa, obedecendo uma certa sequência, porém, essa não é a única forma de se programar. Desse modo, não seria correto falar em termos de uma lógica de programação, mas sim, acerca de qual paradigma estamos usando.

4.1 PARADIGMAS DE PROGRAMAÇÃO

Um paradigma de programação reflete um pensamento filosófico sobre como resolver problemas usando um computador. Cada um terá vantagens e desvantagens, ou seja, um determinado paradigma poderá ser mais adequado que outro, dependendo do tipo de problema que se quer resolver. Alguns exemplos de paradigma de programação são: lógico, funcional, imperativo, orientado a objetos. Algumas linguagens de programação dão um enfoque maior a um desses, como a linguagem Prolog, que é focada em Programação Lógica, e a Smalltalk, focada em Orientação a Objetos. Já linguagens como Python são chamadas multiparadigma, pois permitem escrever códigos usando paradigmas distintos. Python permite escrever códigos tanto na forma imperativa, como vimos até o momento, quanto usando Orientação a Objetos ou Programação Funcional. Para se tornar um bom programador é importante conhecer as diversas formas de se resolver problemas usando programação. Logo, é importante conhecer outros paradigmas de programação.

Programação Lógica

No final da década de 1970 e início da década de 1980, o governo japonês iniciou um projeto chamado Quinta Geração de Computadores, com foco em Inteligência Artificial. A intenção era criar um computador que tivesse como base a Programação Lógica, criando um computador *prologuiano*. A Programação Lógica usa como base a lógica formal, na qual regras são escritas na forma de cláusulas.

Como exemplo de regra em lógica formal temos: i) Todo humano é mortal; ii) Sócrates é humano; iii) Sócrates é mortal. As regras anteriores podem ser escritas usando a programação lógica,

na linguagem chamada Prolog, como:

```
humano(sócrates).  
mortal(X) :- humano(X).
```

Na primeira linha do código anterior, estamos afirmando que sócrates é humano . Na segunda linha, estamos escrevendo que um humano x qualquer será mortal. Observe que essa linguagem tem características muito distintas das linguagens estruturadas que são comumente conhecidas. Existem bibliotecas, como PyDatalog (<https://sites.google.com/site/pydatalog/>), que são usadas como extensão para permitir escrever programas lógicos em Python.

Programação Funcional

Em 1958, John MacCarthy apresenta um artigo no qual mostra que é possível usar exclusivamente funções matemáticas como estrutura de dados. Pensar em resolver problemas computacionais usando apenas funções matemáticas é chamado de Programação Funcional. Uma linguagem com esse caráter é a Lisp, que foi muito popular nas décadas de 1970 e 1980, com aplicações em inteligência artificial. A seguir, temos um exemplo de código em Lisp que faz a soma de 1 e 5:

```
((lambda (arg) (+ arg 1)) 5)
```

Em Python, podemos criar programas funcionais da seguinte forma:

```
number = [1,2,3,4]  
print(sum(number))
```

No código anterior, criamos uma lista de números, em seguida pedimos para exibir a soma desses números.

Programação Imperativa e Estruturada

A Programação Imperativa foi um dos primeiros paradigmas de programação. Seu conceito básico está na ideia de criar declarações que representam ordens a serem executadas pelo computador. Um exemplo de linguagem puramente imperativa é a linguagem Assembly.

Quando se desenvolve um programa em Assembly os recursos de programação presentes são sequências de instruções e saltos condicionais e incondicionais. É muito fácil escrever códigos que são pouco legíveis e cuja estrutura lógica é difícil de acompanhar. Para resolver esses problemas da Programação Imperativa é que surge a Programação Estruturada.

O paradigma de Programação Estruturada procura formalizar a ideia de dividir o programa em blocos. Ela é derivada direta da Programação Imperativa e foi apresentada no início dos anos de 1970 pelo suíço Niklaus Wirth. Assim, continuamos com o conceito de dar ordens ao computador. A vantagem nesse caso é que o programador precisa saber exatamente o estado do programa antes e depois de cada bloco, ou seja, o programador passa a saber o que esperar da execução de um dado bloco.

A Programação Estruturada tem como base três estruturas de controle para a construção de um programa:

1. Sequência: Define que as instruções do programa são executadas sequencialmente, de cima para baixo, linha a linha, de forma sequencial.
2. Seleção: Permite que o fluxo de execução das instruções seja executado de acordo com uma condição lógica que é

avaliada e, caso seja verdadeira, permite que uma instrução ou um grupo de instruções seja executado.

3. Repetição: Permite que uma instrução ou um grupo de instruções seja executado repetidamente, de acordo com uma condição lógica.

A Programação Estruturada usa um número pequeno de estruturas de controle, que são:

1. Sequencial - Um comando abaixo do outro;
2. Decisão - Os condicionais do tipo `if/elif/else`
3. Seleção múltipla - Os condicionais de caso, presentes em linguagens tipo C `case/select` ;
4. Iteração do tipo enquanto-faça - Condicional do tipo `while`
5. Iteração do tipo repita-enquanto - Condicional do tipo `do/while`

A partir desse conjunto de estruturas de controle, temos o chamado princípio da Programação Estruturada, que afirma que qualquer algoritmo pode ser escrito combinando-se blocos formados pelas estruturas de controle.

É dessa possibilidade de escrever qualquer algoritmo, usando Programação Estruturada, que surge erroneamente o termo "Lógica de Programação", no sentido de existir uma lógica universal, para resolver problemas usando computadores. Assim, um curso de "Lógica de Programação" normalmente é na verdade um curso de Programação Estruturada.

Aqui vale ressaltar que as principais linguagens modernas, como Python, C/C++, Java, JavaScript e diversas outras, têm como

base a Programação Estruturada, por isso, para quem estiver iniciando é fundamental dominar os princípios desse paradigma.

Programação Orientada a Objetos

Na Programação Estruturada, o que se tem, em geral, são trechos de códigos, que fazem parte de sub-rotinas ou funções, que se combinam para realizar uma tarefa. O grande problema desse paradigma é que, à medida que o sistema cresce e mais funções e ordens são organizadas em conjunto, mais difícil fica a manutenção do sistema. Pensar somente em ordens a serem executadas acaba por impedir o desenvolvimento de sistemas mais complexos. Além de que tentar trocar um trecho de código por algo mais eficiente pode ser uma tarefa quase impossível.

Dessa forma, notou-se que era preciso repensar a forma de se desenvolver softwares. Ou seja, que era importante ter códigos mais independentes, mas que ao mesmo tempo pudessem ser altamente reaproveitáveis. Um paradigma com tais características é o da Programação Orientada a Objetos (POO). Apesar de parecer um conceito novo, o termo POO foi criado por Alan Kay, autor da linguagem Smalltalk, desenvolvida no fim da década em 1960 e início de 1970. Contudo, a primeira linguagem que realmente usava o conceito de Orientação a Objetos foi a Simula 67, desenvolvida por Ole-Johan Dahl e Kristen Nygaard na década de 1960.

Apesar de antigo, somente na década de 1990 é que realmente esse paradigma começou a se tornar popular e passou a ser adotado por grandes empresas. Segundo Allan Kay (Essa informação foi disponibilizada a partir de uma conversa pessoal

entre Allan Kay e Stefan Ram, o qual pode ser acessada em http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en), o termo POO foi cunhado ao final de 1966 inspirado por linguagens como o Simula e também pelo fato de que ele possuía formação tanto em matemática quanto em biologia. O conceito principal de POO surge da forma como células e computadores individuais em uma rede conseguem comunicar entre si somente através de mensagens.

Pensando em células que comunicam entre si, podemos ver que cada célula não precisa saber o que existe dentro da outra. O mais importante é que uma consiga compreender a mensagem que está recebendo. Nesse caso, temos que toda a informação necessária para que a célula funcione está dentro do seu involucro, ou simplesmente encapsulada na própria célula. Podemos estender essa mesma visão para um determinado indivíduo ou um animal. Não precisamos conhecer como o corpo humano funciona para que exista a reprodução. Além disso, para criar novos indivíduos que carreguem características do pais, basta herdar os genes do seu progenitor. Essa é a essência original da POO, encapsulamento e herança de informações.

Pensando novamente nas células, temos dois componentes importantes: um é a célula em si, o outro é a receita de criação da célula, que no caso da biologia é o DNA. Em POO, o equivalente ao DNA é a classe, já a célula em si é o objeto. Classe é a receita e o objeto é o elemento de código que está em execução e que ocupa espaço na memória RAM do computador.

4.2 CONCLUSÃO

Neste capítulo, vimos que não existe uma lógica universal de programação, mas que na verdade o que temos são diversos paradigmas. Cada paradigma representa uma forma distinta usada para se tratar problemas usando computação. Ao longo deste livro, será dado destaque aos paradigmas de Programação Estruturada, Orientada a Objetos e Funcional. No próximo capítulo, entraremos em mais detalhes sobre Programação Orientada a Objetos.

PROGRAMAÇÃO ORIENTADA A OBJETOS

No capítulo anterior, vimos que, ao se tratar de programação, não existe uma lógica universal, mas sim, que temos formas diferentes de pensar em soluções de problemas usando computadores. A essas diferentes formas de pensar chamamos de paradigmas de programação. Vimos também que Python é uma linguagem multiparadigma. Ao se começar com uma linguagem de programação, temos que pensar sobre como resolver um problema e qual a melhor forma de fazê-lo. Mas qual paradigma adotar? O que será melhor em cada caso? Respostas a essas questões são bem difíceis, mas a grande verdade é que só com experiência e buscando entender as nuances de determinados problemas é que aprendemos a encontrar um melhor caminho. Assim, saber como resolver problemas usando diferentes paradigmas é fundamental para se tornar um programador melhor.

A grande vantagem de Programação Orientada a Objetos, POO, é que podemos decompor mais facilmente um problema, separando-o em partes que comunicam entre si, sem que necessariamente cada parte saiba como a outra foi implementada. Anteriormente, foi apresentado um projeto do Jogo da Velha formado apenas de funções e condicionais, usando o Paradigma

Estruturado. Neste capítulo, vamos remodelar nosso programa baseado em POO.

Apesar de Python ser uma linguagem multiparadigma, cada componente que construímos na linguagem pode ser tratado como um objeto. Sendo assim, será importante entrar em mais detalhes sobre o que é esse paradigma. Neste capítulo, será apresentada uma breve introdução a Programação Orientada a Objetos em Python.

5.1 CLASSES, OBJETOS E INSTÂNCIAS

De forma mais formal, temos que uma classe é uma descrição de um conjunto de objetos com propriedades, comportamentos, relacionamentos e semântica comuns. Nas classes, definimos as variáveis como sendo atributos, enquanto os métodos são funções que executam uma ação. Um atributo vai definir as características de um objeto. Podemos pensar em `cor` como sendo um atributo, já o `'amarelo'` é a característica de `cor` que um objeto terá. Vamos criar a classe `Carro` em Python.

```
class Carro(object):
    def __init__(self, cor, potencia, modelo, marca):
        self.cor = cor
        self.potencia = potencia
        self.modelo = modelo
        self.marca = marca

    def acelerar(self):
        print("Vrummm")

    def freiar(self):
        print("parando")
```

Aqui foi usada a palavra reservada `class` para indicar que estamos começando um bloco de código que representa a classe

`Carro` . Observe que a palavra `Carro` logo após `class` começa com letra maiúscula. Apesar de não ser obrigatório, uma boa prática é colocar a primeira letra do nome de uma classe em maiúscula, enquanto o nome do objeto deverá ficar em minúscula. Após escrever a palavra `Carro` colocamos entre parênteses a palavra `object` . Isso indica que a classe `Carro` está herdando características de uma classe fundamental chamada `object` . Nas versões mais antigas de Python (versão 2.x), o uso de `object` não é obrigatório, porém para as versões mais recentes é necessário indicar que a classe mãe herda de `object` . Logo em seguida, usamos a palavra de definição de função `def` e criamos um método especial com o nome `__init__` . Esse método é conhecido como construtor. Isso indica que sempre que um objeto é criado, ou instanciado, o primeiro método que será executado de forma automática será o `__init__` . Nesse método fazemos as inicializações dos atributos e de tudo o que for necessário inicializar para o correto funcionamento do objeto.

Podemos pensar no construtor como tudo que deve acontecer quando viramos a chave do motor do carro. Ao girar a chave o sistema elétrico é iniciado e o motor de arranque começa a girar o motor, em seguida a bomba de combustível começa a levar combustível do tanque para o motor, que inicia o ciclo de queima. Assim, para que um objeto funcione em muitos casos será necessário criar passos de inicialização. No caso de POO, podemos adicionar essa sequência inicial no construtor da classe.

Note que a primeira palavra que colocamos nos métodos foi a `self` . Essa palavra é usada para indicar que o método pertence à própria classe, ou que o método vai se referir ao próprio objeto. Assim, todo método deverá conter ao menos como parâmetro a

palavra `self` . Ainda no construtor, passamos como parâmetros a seguinte sequência: `cor` , `potencia` , `modelo` , `marca` . Em seguida, dentro do construtor usamos a palavra `self` seguida de ponto e reutilizamos o nome que foi passado como parâmetro do construtor, por exemplo, definimos `self.cor = cor` . Nesse caso estamos criando o atributo `cor (self.cor)` no qual é inicializado com o valor do parâmetro `cor` . Assim, toda vez que quisermos usar um atributo dentro de outros métodos, usamos o termo `self.nome_do_atributo` .

Uma vez criada a classe `carro`, podemos instanciar um objeto da seguinte forma:

```
carro = Carro(cor="azul", potencia="2.0", modelo="Sandero", marca:
"Renault")
carro.acelerar()
```

No código anterior, criamos o objeto `carro` a partir da classe `Carro` . Nosso `carro` tem como atributos a `cor azul` , a `potência 2.0` , o `modelo é Sandero` e a `marca Renault` . Para usar um dos métodos do objeto, usamos o nome do objeto, seguido de ponto e finalizando com o nome do método que queremos usar. No caso do exemplo foi `carro.acelerar()` .

5.2 HERANÇA

Outro ponto importante em POO é o conceito de herança. Assim como na biologia, herança se trata de passar características a descendentes. Em programação, podemos criar uma classe mãe. Em seguida, podemos criar outras classes derivadas, que terão métodos e atributos da classe mãe, mas que implementam novos métodos e têm atributos próprios. Nesse caso, o que estamos

fazendo é estender a classe mãe. Considere que criamos a classe `veiculo`, que deve ter apenas o método `motor`. Essa será a classe mãe. Em seguida, criaremos a classe `carro`, que herda o método `motor` da classe mãe. Vamos criar um módulo chamado `veiculo.py` e escrever o código a seguir:

```
class veiculo(object):
    def motor(self):
        print("motor do veiculo")

class carro(veiculo):
    def __init__(self, cor, potencia, modelo, marca):
        self.cor = cor
        self.potencia = potencia
        self.modelo = modelo
        self.marca = marca

    def acelerar(self):
        print("Vrummm")

    def freiar(self):
        print("parando")
```

Em seguida, no modo terminal, vamos chamar a classe `carro`:

```
>> carro = Carro(cor="azul", potencia="2.0", modelo="Sandero", ma
rca="Renault")
>> carro.motor()
motor do veículo
```

Note que foi possível usar o método `motor`, pertencente à classe mãe, sem ter escrito o mesmo na classe `carro`. A herança foi explicitada ao passarmos `veiculo` como parâmetro na declaração da classes `carro`, através da declaração `class carro(veiculo):`.

5.3 PÚBLICO, PRIVADO E PROTEGIDO

Em Python, não existe uma forma de criar métodos e atributos que são do tipo privado ou protegido, à risca, mas existe uma convenção, na qual indicamos que um determinado método ou atributo não deve ser usado externamente à classe.

Quando criamos um método ou atributo público, o que estamos fazendo é criando a chamada Interface de Programação de Aplicação (em inglês *Application Programming Interface* - API). A API é uma indicação de como devemos usar um determinado objeto. Por exemplo, criamos o método `acelerar`, isso indica para o usuário de nossa classe que, se ele quiser acelerar o carro, ele deverá usar esse método.

Por outro lado, às vezes para criar um método será necessário escrever muitas linhas de código. Porém, não é uma coisa boa escrever métodos muito grandes, isso gera algo conhecido como *bad smell*, ou cheiro ruim. Não é um bug, nem um problema em si, mas poderá dificultar a manutenção e aprimoramento da sua classe.

Para evitar métodos muito grandes é interessante quebrar um determinado métodos em outros. Porém, quando fazemos essa quebra em submétodos, alguns deles só serão úteis para organizar o código. Para um outro desenvolvedor que estiver usando nossa classe, usar diretamente esses submétodos poderá gerar problemas, pois ele estará usando trechos de código que estão fora de contexto. Nesses casos, é conveniente criar métodos privados, que não deverão ser usados fora do contexto da própria classe e que não devem pertencer à nossa API. Uma convenção para isso é iniciar o método com um *underline* (`_`), para indicar aos outros

programadores que esses métodos não deverão ser usados de forma alguma, devendo ser tratados com privados. Já quando queremos dificultar ainda mais o acesso a um método ou atributo usamos dois *underlines* (`__`), no início do nome. Por exemplo:

```
class Carro(object):
    def __init__(self, cor, potencia):
        self.cor = cor
        self.potencia = potencia
        self._velocidade = 0

    def __atualiza_velocidade(self, valor):
        self._velocidade = valor

    def acelerar(self):
        self.__atualiza_velocidade(valor=10)
        print("Vrummm")

    def freiar(self):
        self.__atualiza_velocidade(valor=0)
        print("Parando")
```

No caso anterior, estamos dizendo que a velocidade do carro não deve ser modificada fora do contexto da classe, devendo ser evitado o seu uso direto por outros desenvolvedores. Isso se deu pelo fato de que no construtor da classe foi criado o atributo `self._velocidade`, em que o nome `velocidade` começa com um *underline*. Note que criamos um método com dois *underlines*. Estamos dificultando ainda mais o uso desses métodos: mesmo via herança seu uso não será direto. Nesse caso, podemos dizer que criamos um método protegido. Aqui estamos informando para os desenvolvedores que, mesmo que ele crie outras classes filhas de `Carro`, o método `__atualiza_velocidade` não deverá ser sobrescrito ou usado de forma indiscriminada.

5.4 DESCOBRINDO MÉTODOS E ATRIBUTOS

O interessante em Python é que tudo é objeto, mesmo os tipos primitivos como inteiros e strings são objetos. Para saber quais métodos e atributos estão associados a um dado objeto usamos a função `dir`. Vamos criar uma variável contendo um primitivo simples do tipo ponto flutuante e ver quais métodos ele possui:

```
>>> a = 1.2
>>> dir(a)
['_abs_', '__add__', '__class__', '__coerce__',
 '__delattr__', '__div__', '__divmod__', '__doc__', '__eq__',
 '__float__', '__floordiv__', '__format__', '__ge__',
 '__getattr__', '__getnewargs__', '__gt__', '__hash__',
 '__init__', '__int__', '__le__', '__long__',
 '__lt__', '__mod__', '__mul__', '__ne__',
 '__neg__', '__new__', '__nonzero__', '__pos__', '__pow__',
 '__radd__', '__rdiv__', '__rdivmod__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__',
 '__rmul__', '__rpow__', '__rsub__', '__rtruediv__',
 '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__truediv__', 'conjugate', 'imag',
 'real']
```

Vemos que existe uma lista de opções. Alguns desses métodos são usados pelo próprio interpretador para se realizarem comparações entre dois números, por exemplo, ou permitir a conversão de um tipo para outro, quando possível.

No caso do objeto `carro` temos:

```
print(dir(carro))
['_class_', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'acelerar', 'cor', 'freiar', 'marca', 'modelo', 'potencia']
```

No caso da nossa classe, nós não definimos todos esses

métodos que foram exibidos pelo comando `dir` . Eles fazem parte da nossa classe devido ao fato de que são métodos de `object` , ou seja, nossa classe herdou esses métodos de `object` .

5.5 MÓDULOS COMO OBJETOS

No capítulo de introdução à linguagem Python, vimos que módulos são arquivos com extensão `.py` contendo o nosso programa. Agora que criamos uma classe, será interessante definir um padrão para guardar as classes, de forma que possamos usá-las mais facilmente em outros projetos. Uma boa prática é escrever uma classe por arquivo. O arquivo deverá ter o mesmo nome da classe nele contida. Porém, esse arquivo deverá começar com letra minúscula. Por exemplo, para a classe `Carro` , vamos criar um arquivo chamado `carro.py` .

O módulo `carro.py` terá então a seguinte estrutura:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Modulo contendo a classe construtora de carros.
"""

class Carro(object):
    def __init__(self, cor, potencia):
        self.cor = cor
        self.potencia = potencia
        self._velocidade = 0

    def __atualiza_velocidade(self, valor):
        self._velocidade = valor

    def acelerar(self):
        self.__atualiza_velocidade(valor=10)
        print("Vrummm")
```

```
def freiar(self):
    self.__atualiza_velocidade(valor=0)
    print("Parando")
```

Agora basta abrir o terminal, ou o `cmd` do Windows, e fazer mudança de diretório até o local em que se encontra o arquivo `carro.py`. Em seguida, executamos o interpretador Python (com o comando `python` ou `python.exe`, dependendo do sistema). Se quisermos usar a classe, basta apenas usar o comando de importação:

```
>>> import carro
>>> print(carro.__doc__)
```

Modulo contendo a classe construtora de carros.

```
>>> meu_carro = carro.Carro('preto', '2.0')
>>> meu_carro.acelerar()
Vrummm
>>>
```

O código anterior está sendo executado via terminal. Na primeira linha, usamos o comando `import carro`, no qual importamos o módulo principal e passamos a ter acesso a todo o conteúdo disponível dentro do módulo. Em seguida, pedimos para exibir na tela a variável `__doc__`, que contém informações de documentação do módulo. Note que o texto exibido é justamente o comentário em mais de uma linha que colocamos no módulo, logo abaixo da informação sobre codificação do texto do módulo. Para usar a classe `carro`, fizemos uso do nome do módulo seguido por ponto e acrescentamos o nome da classe. Dessa forma, vemos que um módulo também é um objeto. Em vez de fazer a importação do módulo de forma geral, podemos importar elementos específicos. Se quisermos importar apenas a classe `carro`, para usá-la diretamente sem ter que ficar referenciando o nome do módulo

usamos:

```
from carro import Carro
```

Dessa forma, usando o comando `from`, indicamos de qual módulo queremos importar. Já com o `import` indicamos exatamente o que queremos usar.

5.6 PACOTES

Quando trabalhamos em um projeto, na maioria das vezes vamos criar mais de uma classe, ou teremos muitas funções que deverão ser agrupadas de acordo com algum critério. Nesses casos, acabaremos tendo uma série de módulos soltos. O ideal é ir agrupando os módulos em pastas, separando-os em grupos de afinidade. Tal como costumamos fazer ao criar páginas Web, em que colocamos imagens na pasta `img` e código JavaScript na pasta `js`, por exemplo, o mesmo podemos fazer com os módulos.

Quando criamos pastas para organizar nosso código Python chamamos isso de pacote. Ao criar um pacote, podemos importar módulos dentro de pacotes tal como fazíamos com os módulos. Para criar um pacote basta fazer uma nova pasta e dentro dela adicionar um arquivo com o nome `__init__.py`. Assim, o interpretador Python poderá verificar se uma pasta é ou não um pacote contendo módulos.

Vamos criar uma pasta chamada `fabrica` e adicionar o módulo `carro.py` dentro. Nossa pasta terá a estrutura:

```
fabrica
|  __init__.py
|  carro.py
```

Vamos abrir o terminal e mudar para o diretório logo acima da pasta `fabrica` . Em seguida vamos executar o terminal Python. Agora podemos acessar a classe `Carro` da seguinte forma:

```
>>> from fabrica.carro import Carro
>>> meu_carro = Carro('preto', '2.0')
>>> meu_carro.acelerar()
Vrummm
>>>
```

O nome da chamada foi `from nome_do_pacote.nome_do_modulo import classe, funcao, constante` , ou seja, usamos o operador ponto para selecionar um modulo que estava dentro do pacote e em seguida importamos algo específico. Podemos criar subpacotes dentro de pacotes, bastando criar novos diretórios e lá dentro adicionar os arquivos `__init__.py` e todos os outros módulos:

```
fabrica
|  __init__.py
|  carro.py
|-- motor
    |  __init__.py
    |  mercedes.py
    |  toyota.py
```

O arquivo `__init__.py` pode ser um arquivo vazio, mas também pode conter trechos de códigos para fazer uma inicialização de componentes do pacote. Outra coisa importante que pode ter nesse arquivo é a lista com o nome `__all__` . Essa lista deverá conter os nomes dos módulos que serão importados caso o usuário use a expressão `from meu_pacote import *` . O `*` no final do comando está sendo usado para importar todos os módulos que existirem dentro do pacote. Com a lista `__all__` dentro do arquivo `__init__.py` , informamos ao interpretador quais deverão ser os módulos que o usuário pode importar se ele

tentar buscar tudo o que estiver contido no pacote. Além disso, o arquivo `__init__.py` é o local em que escrevemos o *docstring* com a documentação geral sobre o pacote que estamos criando.

5.7 CONCLUSÃO

Neste capítulo, foram abordados alguns conceitos fundamentais de Programação Orientada a Objetos. Também vimos que tudo em Python são objetos, inclusive que os módulos e pacotes são tratados como objetos. Em seguida, foi tratada a questão de que as classes em Python não possuem estritamente métodos e atributos privados e protegidos. Porém, foi mostrado que existe uma convenção para indicar que alguns métodos ou atributos não devem ser utilizados fora do contexto da codificação da classe em si. Também vimos que módulos são arquivos contendo o texto do nosso código, enquanto os pacotes que são ficheiros nos quais adicionamos conjuntos de módulos.

ORIENTAÇÃO A OBJETOS NA PRÁTICA

Quando criamos o Jogo da Velha, usando Programação Estruturada, foram criadas funções, mas não tínhamos organizado o código de acordo com um significado mais concreto para cada elemento. Para mostrar o potencial da Programação Orientada a Objetos, vamos reescrever o jogo. Assim, vamos colocar em prática toda a teoria que vimos no capítulo anterior.

Vamos criar três classes, uma representando a tela do jogo, outra para o controle e uma para os jogadores. Também vamos criar um módulo contendo a função principal do jogo. Dessa forma, teremos uma maior clareza sobre os componentes que fazem parte do nosso sistema, facilitando a manutenção e a troca de componentes. Nosso sistema ficará mais consistente, modular e com menos chances de falhas.

Controle do jogo

Para começar, vamos criar a classe que representa o controle do jogo. Vamos começar os exemplos apresentando primeiro as classes baseadas nas funções que desenvolvemos na primeira versão do jogo. Em seguida, vamos adicionando os novos

elementos que precisamos para efetivamente construir o jogo.

Iniciemos o módulo responsável pelo controle, começando pelo construtor da classe:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Módulo responsável por controlar as ações do jogo.
"""
import sys

class Controle(object):

    def __init__(self, stdscr):
        self.stdscr = stdscr
        width = stdscr.getmaxyx()[1]
        self.x_center = (width - 1) // 2
        self.pos_x = 0
        self.pos_y = 0
        self.entrada = None
```

Após a declaração inicial, importamos o módulo padrão `sys`. Esse módulo é usado para ter acesso a algumas variáveis usadas ou mantidas pelo interpretador, além de fornecer acesso a funções com maior poder de interação com o interpretador. No caso dessa classe vamos usar a função `exit`, que quando chamada encerra imediatamente um programa.

O construtor da classe `controle` recebe como parâmetro a tela principal do jogo. Essa tela é um objeto `stdscr`, que já foi apresentado anteriormente. No construtor da classe criamos o atributo `x_center`, que guardará as informações do centro horizontal da tela; os atributos `pos_x` e `pos_y`, contendo a posição relativa do movimento corrente do jogador principal; por último, criamos o atributo `entrada`, que vai conter os dados da tecla que o jogador pressionou.

Vamos precisar de um método para verificar a posição relativa da jogada. Criaremos um método protegido, lembrando que em Python usamos um `_` para antes do nome do método para indicar tal fato. Nas mais diversas linguagens de Programação Orientada a Objetos, o uso de métodos ou instâncias protegidos ou privados é algo comum. Em geral, seu uso serve para indicar para um futuro desenvolvedor que aquele método ou atributo é de uso exclusivo de funcionalidades internas da classe, não devendo ser acessado fora dela. No caso do nosso projeto, estamos indicando que o método `_limites` será criado como ferramenta de legibilidade do código, em que em vez de criar um método muito grande, acabamos por particioná-lo em métodos menores. Porém, alguns desses métodos menores só terão significado dentro da própria classe. Em linguagens como Java e C++, a diferença entre privado e protegido é que um método/atributo privado não será acessível nem mesmo às classes filhas. Já os métodos protegidos podem ser usados por classes derivadas da classe mãe, que contém tal método.

```
def _limites(self):  
  
    if self.pos_x > 2:  
        self.pos_x = 0  
    if self.pos_x < 0:  
        self.pos_x = 2  
  
    if self.pos_y > 2:  
        self.pos_y = 0  
  
    if self.pos_y < 0:  
        self.pos_y = 2
```

A função `_limites` foi modificada, mas continua fazendo o papel de verificar se a posição relativa da jogada ainda está dentro dos limites do tabuleiro. Note que agora ela não recebe nem retorna nenhum valor. Porém, dentro dessa função fazemos a

modificação dos atributos `pos_x` e `pos_y`. Por exemplo, se o atributo `pos_x` for maior que dois, seu valor será modificado para 0. Nesse caso, vemos que um atributo não precisa ser algo estático, mas que reflete o momento em que um dado objeto se encontra. Outro exemplo seria a altura de uma pessoa, um atributo que varia ao longo da vida de um indivíduo. A velocidade atual de um carro também é um atributo que varia a todo o momento. Podemos pensar em atributos como sendo variáveis globais, mas que só existem dentro do contexto de cada objeto em si. Essa é a propriedade de encapsulamento. Cada objeto poderá ter variáveis que são globalmente acessadas dentro do seu escopo.

Precisamos também de um método para gerenciar as entradas do jogador:

```
def espaco_do_tabuleiro(self):

    self.entrada = self.stdscr.getkey()

    if self.entrada == 'q':
        sys.exit(0)

    if self.entrada == 'a':
        self.pos_x -= 1
    elif self.entrada == 'd':
        self.pos_x += 1
    elif self.entrada == 's':
        self.pos_y += 1
    elif self.entrada == 'w':
        self.pos_y -= 1
    else:
        pass

    self._limites()
```

No método `espaco_do_tabuleiro` passamos a gerenciar a entrada digitada pelo jogador, sendo que essa informação fica

armazenada no atributo `entrada`. Veja que, ao digitar a tecla `q`, o método `exit` será chamado, encerrando o jogo. O valor `0` passado à função `exit` fará com que ao final o nosso programa retorne o valor `0` para o terminal. Isso serve de indicação de que o programa terminou de forma correta. Se passarmos o valor `1` para essa função, estamos indicando que algum problema interno ocorreu. Isso é muito útil ao trabalhar com sistemas Unix, no qual podemos tomar ações de acordo com o fato de um programa ter sido corretamente executado ou não.

As teclas `a` e `d` representam movimentos para esquerda e direita, enquanto as teclas `s` e `w` representam movimento para baixo e para cima. Logo após uma ação do jogador, o método `_limites` será chamado. Assim verificamos se o movimento ultrapassou ou não os limites do tabuleiro, atualizando as posições relativas de forma correta.

Tendo as posições relativas, precisamos passar essa informação para o gerenciador da tela do jogo. Isso é feito por meio do método `cursor`:

```
def cursor(self):  
    cursor_y = 9  
    cursor_x = self.x_center - 3  
    if self.pos_y == 1:  
        cursor_y += 2  
  
    if self.pos_y == 2:  
        cursor_y += 4  
  
    if self.pos_x == 1:  
        cursor_x += 2  
  
    if self.pos_x == 2:  
        cursor_x += 4
```

```
self.stdscr.move(cursor_y, cursor_x)
```

No método anterior, `cursor_x` e `cursor_y` são as posições absolutas, com relação ao centro da tela do jogo. Trabalhar com movimentos relativos, para depois converter para movimento absoluto permite uma maior liberdade na hora de desenhar a tela do jogo, não tendo que ficar modificando todas as classes, quando precisamos modificar o tamanho da tela.

Após converter os movimentos relativos para posições absolutas, fazemos a chamada do método `move` do objeto `stdscr`, que fará o posicionamento do curso no local correto do tabuleiro.

Com isso criamos um ente que carrega a responsabilidade de gerenciar as informações de entrada. A grande vantagem aqui é que passamos a ter clareza sobre o papel de cada código, além de o termos organizado de acordo com um significado bem específico. Isso facilita muito a manutenção e aperfeiçoamento do código.

O módulo `controle.py`, que contém a classe `Controle`, ficou da seguinte forma:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Módulo responsável por controlar as ações do jogo.
"""
import sys

class Controle(object):

    def __init__(self, stdscr):
        self.stdscr = stdscr
        width = stdscr.getmaxyx()[1]
        self.x_center = (width - 1) // 2
        self.pos_x = 0
```

```

self.pos_y = 0
self.entrada = None

def _limites(self):

    if self.pos_x > 2:
        self.pos_x = 0
    if self.pos_x < 0:
        self.pos_x = 2

    if self.pos_y > 2:
        self.pos_y = 0

    if self.pos_y < 0:
        self.pos_y = 2

def espaco_do_tabuleiro(self):

    self.entrada = self.stdscr.getkey()

    if self.entrada == 'q':
        sys.exit(0)

    if self.entrada == 'a':
        self.pos_x -= 1
    elif self.entrada == 'd':
        self.pos_x += 1
    elif self.entrada == 's':
        self.pos_y += 1
    elif self.entrada == 'w':
        self.pos_y -= 1
    else:
        pass

    self._limites()

def cursor(self):

    cursor_y = 9
    cursor_x = self.x_center - 3
    if self.pos_y == 1:
        cursor_y += 2

    if self.pos_y == 2:
        cursor_y += 4

```

```

if self.pos_x == 1:
    cursor_x += 2

if self.pos_x == 2:
    cursor_x += 4

self.stdscr.move(cursor_y, cursor_x)

```

6.1 A TELA DO JOGO

Vamos agora desenhar a tela do nosso jogo. Nessa classe, vamos colocar as funções que desenharam as informações do jogo. Lembre-se de que na classe controle foi adicionado o atributo que determina o centro da tela principal. Adicionamos essa informação lá porque precisávamos disso para poder posicionar o cursor corretamente na tela. Em vez de duplicar essa informação na nova classe, vamos criar uma relação de dependência entre a classe `tela` com a classe `cursor`. Com isso, estamos querendo informar que, apesar de a tela ser um ente com características próprias, algumas informações que ele deve tratar vão depender de outros objetos. Isso é algo natural, se pensarmos que queremos coletar informações da entrada e transferir isso para a tela do jogo.

O construtor da classe `Tela` será o seguinte:

```

class Tela(object):
    """
    Classe para desenhar a tela do Jogo
    Recebe como atributo a tela gerada pelo módulo curses
    """

    def __init__(self, stdscr, posicoes):
        self.stdscr = stdscr
        self.posicoes = posicoes

```

O objeto `stdscr` representa a tela em si, enquanto o atributo

posicoes é a matriz contendo os movimentos das jogadas (uma matriz representando os x 's e o 's das jogadas).

O primeiro método será o que desenha na tela a mensagem de boas-vindas:

```
def boas_vindas(self):
    self.stdscr.addstr(1, 1, "Bem-vindo ao Jogo da Velha.")
    self.stdscr.addstr(2, 1, "Pressione Q para sair ou H para obter ajuda.")
    self.stdscr.addstr(3, 1, "Para jogar, digite umas das teclas: A, S, W, D.")
    self.stdscr.addstr(16, 1, "Desenvolvido por : E. S. Pereira.")
)
    self.stdscr.addstr(17, 1, "Licença Nova BSD.")
```

Aqui estamos adicionando os textos na tela, por meio do método `addstr`, lembrando que o primeiro valor do método representa a linha em que o texto deve começar a ser escrito, enquanto o segundo representa a coluna em que o texto deve iniciar. Seguindo a mesma ideia, vamos criar o método responsável por desenhar a tela com as informações de ajuda:

```
def ajuda(self):
    self.stdscr.clear()
    self.stdscr.border()
    self.stdscr.addstr(1, 1, "Pressione Q para sair ou H para exibir essa ajuda.")
    self.stdscr.addstr(2, 1, "Para mudar a posição, use as teclas: A, S, W, D")
    self.stdscr.addstr(3, 1, "Para definir uma posição do jogo, aperte: Enter")
    self.stdscr.addstr(4, 1, "Para reiniciar a partida, digite: Y")
)
    self.stdscr.addstr(5, 1, "Pressione espaço para sair dessa tela.")
    self.stdscr.refresh()
```

Nesse método, primeiramente, limpamos a tela anterior. Em seguida, chamamos o método que desenha bordas ao redor da tela.

Acrescentamos o texto com as informações de como jogar. Finalmente, fazemos a atualização efetiva da tela, com o método `refresh`.

Ao final de cada partida será necessário reiniciar a tela. Para isso, criaremos o seguinte método:

```
def reiniciar_tela(self, limpar=True):
    if limpar is True:
        self.stdscr.clear()
    self.stdscr.border()
    self.boas_vindas()
    self.stdscr.refresh()
```

Precisamos também de métodos para criar a tela com informações do final da partida:

```
def fim_de_jogo(self, vencedor):
    self.stdscr.addstr(6, 1, "O %s venceu..." % vencedor)
    self.stdscr.addstr(7, 1, "Pressione y para jogar novamente ou q para sair.")
    self.stdscr.refresh()
```

E um método para atualizar o placar ao fim de cada jogada:

```
def placar(self, jogador1, jogador2):
    self.stdscr.addstr(4, 1, "Jogador: {0} | Máquina {1}.".format(
        jogador1, jogador2))
```

Por fim, será necessário conectar o controlador com a tela do jogo. Faremos isso ao criar o método `tabuleiro`:

```
def tabuleiro(self, controle):

    self.stdscr.clear()
    self.reiniciar_tela(limpar=False)

    self.stdscr.addstr(10, controle.x_center - 3, "-----")
    self.stdscr.addstr(12, controle.x_center - 3, "-----")
    i = 9
    for linha in self.posicoes:
```

```
tela = "%s|%s|%s " % tuple(linha)
self.stdscr.addstr(i, controle.x_center - 3, tela)
i += 2
```

Note que o método `tabuleiro` recebe um objeto do tipo `controle`. Nosso tabuleiro é então desenhado levando em conta o atributo `x_center`, ou seja, o tabuleiro deverá ser centralizado de acordo com um atributo da classe `controle` do jogo. Em seguida, desenhamos na tela cada uma das linhas da matriz de jogada, armazenada no atributo `self.posicoes`

O módulo `tela.py`, contendo a classe `Tela`, será:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Módulo contendo informações de exibição da Tela do jogo.
"""

class Tela(object):
    """
    Classe para desenhar a tela do Jogo
    Recebe como atributo a tela gerada pelo módulo curses
    """

    def __init__(self, stdscr, posicoes):
        self.stdscr = stdscr
        self.posicoes = posicoes

    def boas_vindas(self):
        self.stdscr.addstr(1, 1, "Bem-vindo ao Jogo da Velha.")
        self.stdscr.addstr(2, 1, "Pressione Q para sair ou H para obter ajuda.")
        self.stdscr.addstr(3, 1, "Para jogar, digite umas das teclas: A, S, W, D.")
        self.stdscr.addstr(16, 1, "Desenvolvido por: E. S. Pereira.")
        self.stdscr.addstr(17, 1, "Licença Nova BSD.")

    def ajuda(self):
        self.stdscr.clear()
        self.stdscr.border()
```

```

        self.stdscr.addstr(1, 1, "Pressione Q para sair ou H para
exibir essa ajuda.")
        self.stdscr.addstr(2, 1, "Para mudar a posição, use as te
clas: A, S, W, D")
        self.stdscr.addstr(3, 1, "Para definir uma posição do jog
o, aperte: Enter")
        self.stdscr.addstr(4, 1, "Para reiniciar a partida, digit
e: Y")
        self.stdscr.addstr(5, 1, "Pressione espaço para sair dess
a tela.")
        self.stdscr.refresh()

    def fim_de_jogo(self, vencedor):
        self.stdscr.addstr(6, 1, "0 %s venceu..." % vencedor)
        self.stdscr.addstr(7, 1, "Pressione Y para jogar novament
e ou Q para sair.")
        self.stdscr.refresh()

    def reiniciar_tela(self, limpar=True):
        if limpar is True:
            self.stdscr.clear()
            self.stdscr.border()
            self.boas_vindas()
            self.stdscr.refresh()

    def tabuleiro(self, controle):

        self.stdscr.clear()
        self.reiniciar_tela(limpar=False)

        self.stdscr.addstr(10, controle.x_center - 3, "-----")
        self.stdscr.addstr(12, controle.x_center - 3, "-----")
        i = 9
        for linha in self.posicoes:
            tela = "%s|%s|%s " % tuple(linha)
            self.stdscr.addstr(i, controle.x_center - 3, tela)
            i += 2

    def placar(self, jogador1, jogador2):
        self.stdscr.addstr(4, 1, "Jogador: {0} |Máquina {1}.".for
mat(
            jogador1, jogador2))

```

6.2 OS JOGADORES

Nessa classe, vamos organizar as funções ligadas aos jogadores, bem como as que informam quem ganhou e a que efetivamente realiza uma jogada.

Algo considerado como boa prática em Orientação a Objetos é que não se recomenda o acesso direto aos atributos de um objeto, sendo que isso deveria ser feito por meio de métodos específicos. Isso porque quem deve ser capaz de fazer mudanças em um atributo deve ser o próprio objeto e não um agente externo. Assim, em linguagens como o Java é muito comum criar métodos do tipo `get` para obter um atributo e `set` para mudar o valor de um atributo. Em Python, não é comum fazer o uso desses métodos de forma tão clara. Porém, podemos usar algo chamado **propriedade** que executará os `getters` e `setters` de forma implícita. Vamos testar esse conceito para o atributo que vai retornar o nome do vencedor.

O construtor da classe `Jogadores` será o seguinte:

```
class Jogadores(object):  
  
    def __init__(self, controle, posicoes):  
        self.controle = controle  
        self.posicoes = posicoes  
        self.fim_de_partida = False  
        self._vencedor = None
```

Note que a classe terá os atributos: i) `controle`, que é um objeto construído a partir da classe `Controle`; ii) `posicoes`, que é a matriz contendo as jogadas correntes; iii) `fim_de_partida`, que armazena um booleano indicativo do fim da partida; iv) o atributo privado `_vencedor`, que vai armazenar as informações

do vencedor. Vamos usar esse atributo para ilustrar o uso de `property`.

O uso de `getter` e `setter` é comum em diversas linguagem orientadas a objeto. Seu principal motivo é o de garantir o encapsulamento do código, fazendo com que atributos que devem ser modificados apenas internamente pela classe não sejam modificados de forma direta através de chamadas externas. No caso do atributo `_vencedor`, queremos que somente os métodos da classe `Jogadores` seja capaz de alterar esse atributo de forma explícita. Um exemplo de se fazer isso usando um `setter`, ao estilo do que é usado em Java é apresentado a seguir:

```
import copy
class Jogadores(object):

    def __init__(self, controle, posicoes):
        self.controle = controle
        self.posicoes = posicoes
        self.fim_de_partida = False
        self._vencedor = None

    def set_vencedor(self, vencedor):
        self._vencedor = vencedor

    def get_vencedor(self):
        return copy.copy(self._vencedor)
```

Mas, em Python, tudo é passado por referência, e o uso do método `copy` para criar uma cópia de objetos pode ser mais complicado do que parece. Nesse caso, acabamos não tendo o efeito esperado de encapsulamento. Usando `property` conseguimos construir `getters` e `setters` de forma mais "pythônica", de forma a garantir o encapsulamento na hora de modificar ou obter um atributo da classe.

Para usar o `property` para obter e atualizar o atributo privado `_vencedor`, criamos os métodos privados `_get_vencedor` e `_set_vencedor`. Em seguida, fora de qualquer método adicionamos o comando `vencedor = property(_get_vencedor, _set_vencedor)`. Com isso, criamos um novo atributo, chamado `vencedor` que nos permite acessar, via `getter` e `setter`, o atributo privado `_vencedor`. A codificação dessa parte será:

```
def _get_vencedor(self):
    return self._vencedor

def _set_vencedor(self, vencedor):
    self._vencedor = vencedor

vencedor = property(_get_vencedor, _set_vencedor)
```

Agora veja que, no fim da classe, fora de qualquer método, criamos a variável `vencedor`, que recebe as propriedades de obter e atribuir valor ao atributo `_vencedor`. Note também que nesse último caso não usamos em momento algum o operador `self.`. Assim, se definirmos o nome de uma variável fora de um método, automaticamente ela será tratada como atributo.

Uma vez definido o atributo `_vencedor`, precisamos de um método que verifique quais são as posições vazias no tabuleiro. Isso será feito através do método `jogador`. Caso uma posição esteja disponível, será preenchida com a string `x`. Caso essa operação seja bem-sucedida, o método retorna o booleano `True`, do contrário retornará `False`:

```
def jogador(self):
    if self.posicoes[self.controle.pos_y][self.controle.pos_x] ==
" ":
        self.posicoes[self.controle.pos_y][self.controle.pos_x] =
"x"
```

```

        return True
    return False

```

O método `robo` representará a máquina. Nesse caso, o robô vai vasculhar todas as posições ainda não preenchidas na matriz de posições. Criamos então uma lista contendo os índices `i`, `j` da posição da matriz, que está vazia. Nosso robô apenas escolherá uma posição vazia de forma aleatória. A variável `n_escolhas` é usada para verificar se ainda existe alguma posição que o robô possa jogar. Caso ela exista, é escolhido aleatoriamente um elemento da lista `vazias`. Em seguida, a preenchemos a posição `i`, `j` com a string `o`, através do comando `self.posicoes[j][i] = "o"`. O método `robo` é dado por:

```

def robo(self):
    vazias = []
    for i in range(0, 3):
        for j in range(0, 3):
            if self.posicoes[j][i] == " ":
                vazias.append([j, i])

    n_escolhas = len(vazias)
    if n_escolhas != 0:
        j, i = vazias[randint(0, n_escolhas - 1)]
        self.posicoes[j][i] = "o"

```

Vamos criar um método protegido para verificar se em uma dada linha temos 3 strings `x` ou `o` alinhadas, lembrando que, no Jogo da Velha, quem conseguir alinhar o seu marcador 3 vezes consecutivas ganha a partida:

```

def __total_alinhado(self, linha):
    num_x = linha.count("x")
    num_o = linha.count("o")

    if num_x == 3:
        return "x"
    if num_o == 3:
        return "o"

```

```
return None
```

O método `__total_alinhado` recebe a lista de três elementos, que representa a linha da matriz `posicoes`. Em seguida, contamos quantos elementos repetidos do tipo `x` e o há na linha. Para isso, usamos o método `count` dos objetos do tipo lista. Se um deles tiver um total de três repetições, o método retornará a string representando o vencedor, do contrário, o método retorna `None`.

Com esse método, podemos começar a escrever o método que determina efetivamente o ganhador. No caso do Jogo da Velha, um jogador também pode ganhar se ele colocar a sua marca 3 vezes na diagonal. Nesse caso, criaremos duas listas, de três elementos, contendo apenas as diagonais da matriz `posicoes`:

```
def ganhador(self):
    diagonal1 = [self.posicoes[0][0],
                 self.posicoes[1][1],
                 self.posicoes[2][2]
                ]

    diagonal2 = [self.posicoes[0][2],
                 self.posicoes[1][1],
                 self.posicoes[2][0]
                ]

    gan = self.__total_alinhado(diagonal1)
    if gan is not None:
        self._vencedor = gan
        return True

    gan = self.__total_alinhado(diagonal2)

    if gan is not None:
        self._vencedor = gan
        return True
```

Veja que passamos as diagonais para o método privado

`__total_alinhado` . Se para alguma das diagonais esse método não for do tipo `None` , o atributo `_vencedor` recebe a marcação do campeão e o método `ganhador` retorna `True` .

Além da diagonal, precisamos verificar se um dos jogadores conseguiu fazer as 3 marcações na vertical. Para fazer isso, vamos gerar uma matriz transposta da matriz `posicoes` , ou seja, vamos converter linha em coluna e vice-versa.

```
transposta = [[], [], []]
for i in range(3):
    for j in range(3):
        transposta[i].append(self.posicoes[j][i])
```

Para finalizar, verificamos tanto na horizontal, quanto na vertical, se algum dos jogadores conseguiu colocar três marcações em linha. Como no caso da diagonal, se em algum caso um dos jogadores ganhar, o método atribuirá a string representando o ganhador ao atributo `_vencedor` e retornar `True` , indicando que alguém ganhou. Ao mesmo tempo, criamos a variável `velha=9` . Ela é usada para indicar empate. No Jogo da Velha, temos 9 posições a serem preenchidas. Se todas essas posições forem marcadas, sem que ninguém consiga fazer três posições consecutivas, passa-se a considerar jogo empatado. Nesse caso, costuma-se dizer que o jogo "deu velha" e ninguém ganhou.

```
velha = 9
for i in range(3):

    gan = self.__total_alinhado(self.posicoes[i])
    if gan is not None:
        self._vencedor = gan
        return True

    gan = self.__total_alinhado(transposta[i])
```

```

    if gan is not None:
        self._vencedor = gan
        return True

    velha -= self.posicoes[i].count("x")
    velha -= self.posicoes[i].count("o")

    if velha == 0:
        self._vencedor = "velha"
        return True

    return False

```

O `velha = 9` indica que existem 9 posições que podem ser preenchidas se ao final do laço de repetição, não sobrou nenhuma posição, ou seja `velha == 0`, o atributo `_vencedor` recebe a string `"velha"` e o método retorna `True`. Se o jogo não deu velha e ninguém ainda ganhou, o método retornará `False`.

A seguir está o método `ganhador` completo:

```

def ganhador(self):
    diagonal1 = [self.posicoes[0][0],
                 self.posicoes[1][1],
                 self.posicoes[2][2]
                ]

    diagonal2 = [self.posicoes[0][2],
                 self.posicoes[1][1],
                 self.posicoes[2][0]
                ]

    transposta = [[], [], []]
    for i in range(3):
        for j in range(3):
            transposta[i].append(self.posicoes[j][i])

    gan = self.__total_alinhado(diagonal1)
    if gan is not None:
        self._vencedor = gan
        return True

```

```

gan = self.__total_alinhado(diagonal2)

if gan is not None:
    self._vencedor = gan
    return True

velha = 9
for i in range(3):

    gan = self.__total_alinhado(self.posicoes[i])
    if gan is not None:
        self._vencedor = gan
        return True

    gan = self.__total_alinhado(transposta[i])
    if gan is not None:
        self._vencedor = gan
        return True

    velha -= self.posicoes[i].count("x")
    velha -= self.posicoes[i].count("o")

if velha == 0:
    self._vencedor = "velha"
    return True

return False

```

O último método da classe `Jogadores` será o `jogar`. Esse método verifica se o jogador fez uma jogada. Em seguida, verifica se a partida chegou ao fim, ou não. Esse último ponto ocorrerá, se o método `ganhador` retornar `True`. Finalmente, se o jogador retornou `True` e se o atributo `fim_de_partida` for `False`, chamaremos o método `robo`, que fará a jogada da máquina. Após o robô jogar, devemos verificar se a partida chegou ao fim. O método `jogar` é apresentado a seguir:

```

def jogar(self):
    jogou = self.jogador()
    self.fim_de_partida = self.ganhador()

```

```

if jogou is True and self.fim_de_partida is False:
    self. robo()
    self.fim_de_partida = self.ganhador()

```

Finalmente, o módulo `jogadores.py` terá o seguinte código:

```

#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Módulo responsável por gerenciar os Jogadores.
"""

from random import randint

class Jogadores(object):

    def __init__(self, controle, posicoes):
        self.controle = controle
        self.posicoes = posicoes
        self.fim_de_partida = False
        self._vencedor = None

    def jogador(self):
        if self.posicoes[self.controle.pos_y][self.controle.pos_x
] == " ":
            self.posicoes[self.controle.pos_y][self.controle.pos_
x] = "x"
            return True
        return False

    def robo(self):
        vazias = []
        for i in range(0, 3):
            for j in range(0, 3):
                if self.posicoes[j][i] == " ":
                    vazias.append([j, i])

        n_escolhas = len(vazias)
        if n_escolhas != 0:
            j, i = vazias[randint(0, n_escolhas - 1)]
            self.posicoes[j][i] = "o"

    def __total_alinhado(self, linha):

```

```

num_x = linha.count("x")
num_o = linha.count("o")

if num_x == 3:
    return "x"
if num_o == 3:
    return "o"

return None

def ganhador(self):
    diagonal1 = [self.posicoes[0][0],
                  self.posicoes[1][1],
                  self.posicoes[2][2]
                 ]

    diagonal2 = [self.posicoes[0][2],
                  self.posicoes[1][1],
                  self.posicoes[2][0]
                 ]

    transposta = [[], [], []]
    for i in range(3):
        for j in range(3):
            transposta[i].append(self.posicoes[j][i])

    gan = self.__total_alinhado(diagonal1)
    if gan is not None:
        self._vencedor = gan
        return True

    gan = self.__total_alinhado(diagonal2)

    if gan is not None:
        self._vencedor = gan
        return True

    velha = 9
    for i in range(3):

        gan = self.__total_alinhado(self.posicoes[i])
        if gan is not None:
            self._vencedor = gan
            return True

```

```

        gan = self.__total_alinhado(transposta[i])
        if gan is not None:
            self._vencedor = gan
            return True

        velha -= self.posicoes[i].count("x")
        velha -= self.posicoes[i].count("o")

    if velha == 0:
        self._vencedor = "velha"
        return True

    return False

def jogar(self):
    jogou = self.jogador()
    self.fim_de_partida = self.ganhador()

    if jogou is True and self.fim_de_partida is False:
        self. robo()
        self.fim_de_partida = self.ganhador()

def _get_vencedor(self):
    return self._vencedor

def _set_vencedor(self, vencedor):
    self._vencedor = vencedor

vencedor = property(_get_vencedor, _set_vencedor)

```

6.3 FUNÇÃO PRINCIPAL

Agora que temos os componentes do jogo separados por classe é hora de colocar todos juntos para gerar o nosso jogo. Para isso, vamos criar um módulo que representará o programa principal. Nosso módulo terá o conveniente nome `main`. Primeiramente, devemos importar todas as classes e módulos necessários para o jogo:

```

from curses import initscr, wrapper

```

```

from tela import Tela
from controle import Controle
from jogadores import Jogadores

```

Dentro da função `main`, a primeira coisa a se fazer é instanciar os objetos que representam os elementos do jogo:

```

def main(stdscr):
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

    controle = Controle(stdscr=stdscr)
    tela = Tela(stdscr=stdscr, posicoes=posicoes)
    jogadores = Jogadores(controle=controle, posicoes=posicoes)
    tela.reiniciar_tela()

    jogador_x = 0
    jogador_o = 0

```

No código anterior, criamos a lista `posicoes`, que representa a matriz em que guardaremos as informações da jogada corrente. Também criamos os objetos `controle`, `tela` e `jogadores`. As variáveis `jogador_x` e `jogador_o` vão armazenar o placar do jogo.

O coração do jogo é o laço de repetição, que continuará enquanto o jogador não encerrar efetivamente a partida:

```

while True:
    controle.espaco_do_tabuleiro()
    if jogadores.fim_de_partida is False:
        if controle.entrada == "\n":
            jogadores.jogar()

        if jogadores.fim_de_partida is True:
            ganhador = jogadores.vencedor
            if ganhador == "x":
                jogador_x += 1
            if ganhador == "o":
                jogador_o += 1

```

```

if controle.entrada == "y":
    posicoes = [[' ', ' ', ' ', ' '], [' ', ' ', ' ', ' '], [' ', ' ', ' ',
' ']]
    controle.pos_y = 0
    controle.pos_x = 0
    jogadores.vencedor = None
    jogadores.fim_de_partida = False
    tela.reiniciar_tela()

if controle.entrada == 'h':
    tela.ajuda()
else:
    tela.tabuleiro(controle)
    tela.placar(jogador_x, jogador_o)
    if jogadores.fim_de_partida is True:
        tela.fim_de_jogo(jogadores.vencedor)
    controle.cursor()

```

Dentro do `while` começamos chamando o método `espaço_do_tabuleiro`, que é responsável por mapear as teclas do jogo. Em seguida, verificamos se o jogo chegou ao fim, usando o atributo `fim_de_partida`. Depois, vemos se o jogador apertou a tecla `Enter`. Ainda dentro desse sub-bloco verificamos se ao final de cada partida existirá um novo ganhador e adicionamos a pontuação para o vencedor.

Um outro ponto importante está na necessidade de verificar se o jogador deseja reiniciar uma partida. Nesse caso, se ele digitar a tecla `y`, devemos fazer com que a variável `posicoes` seja reiniciada, fazendo também com que as variáveis `pos_x` e `pos_y` fiquem com valor `0`. Em seguida, colocamos `vencedor` como vazio (`None`) e definimos `fim_de_partida` como `False`. Também adicionamos a opção de exibir a tela de ajuda ao clicar na tecla `h`. Finalmente, fazemos com que a tela seja redesenhada a cada vez que uma nova tecla for digitada. Se uma partida tiver

terminado, o método `fim_de_jogo` deverá ser chamado.

O código completo do módulo `main.py` será o seguinte:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Módulo principal do Jogo
"""

from curses import initscr, wrapper

from tela import Tela
from controle import Controle
from jogadores import Jogadores

def main(stdscr):
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

    controle = Controle(stdscr=stdscr)
    tela = Tela(stdscr=stdscr, posicoes=posicoes)
    jogadores = Jogadores(controle=controle, posicoes=posicoes)
    tela.reiniciar_tela()

    jogador_x = 0
    jogador_o = 0

    while True:
        controle.espaco_do_tabuleiro()
        if jogadores.fim_de_partida is False:
            if controle.entrada == "\n":
                jogadores.jogar()

            if jogadores.fim_de_partida is True:
                ganhador = jogadores.vencedor
                if ganhador == "x":
                    jogador_x += 1
                if ganhador == "o":
                    jogador_o += 1

            if controle.entrada == "y":
                posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
```

```

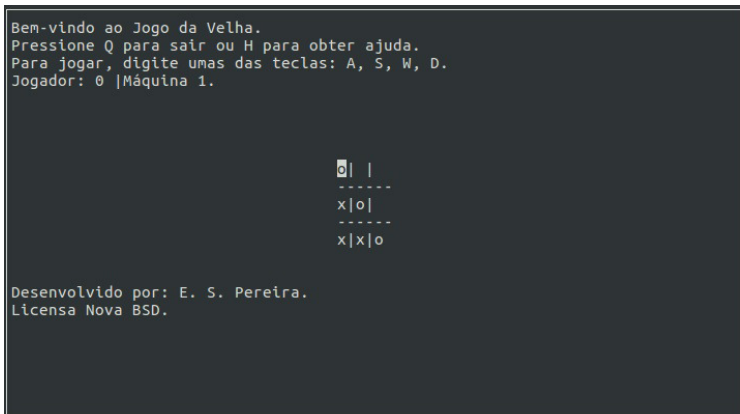
        controle.pos_y = 0
        controle.pos_x = 0
        jogadores.vencedor = None
        jogadores.fim_de_partida = False
        tela.reiniciar_tela()

    if controle.entrada == 'h':
        tela.ajuda()
    else:
        tela.tabuleiro(controle)
        tela.placar(jogador_x, jogador_o)
        if jogadores.fim_de_partida is True:
            tela.fim_de_jogo(jogadores.vencedor)
            controle.cursor()

if __name__ == "__main__":
    initscr()
    wrapper(main)

```

Vamos testar o jogo. Vamos rodar o programa, e tentar usar a tecla `y` para reiniciar a partida. Algo deu errado. Ao pressionar `y` o tabuleiro não ficou limpo, como mostra a figura a seguir:



```

Bem-vindo ao Jogo da Velha.
Pressione Q para sair ou H para obter ajuda.
Para jogar, digite umas das teclas: A, S, W, D.
Jogador: 0 | Máquina 1.

          0 | |
          -----
          x|o|
          -----
          x|x|o

Desenvolvido por: E. S. Pereira.
Licença Nova BSD.

```

Figura 6.1: Área do tabuleiro não ficou limpa após pressionar a tecla "y".

Algo errado aconteceu dentro do bloco que verifica se o jogador quer reiniciar o jogo. Esse erro não é trivial. Ele surge da forma como um objeto é passado para outra variável. No início da função `main` criamos a variável chamada `posicoes`, que contém um objeto do tipo `lista`. O que a variável faz é apontar para a referência a essa lista, ou seja, ela aponta para o local na memória RAM em que a lista existe. Quando passamos a variável `posicoes` para o construtor de `tela` e `jogadores` o que estamos fazendo é passando o valor da referência de um objeto. Dessa forma, estamos indicando para os objetos em que posição da memória eles devem procurar a lista, que nesse momento está sendo referenciada pela variável `posicoes`.

Dentro do bloco, demarcado pela verificação da tecla `y`, quando tentamos reiniciar a variável `posicoes`, o que estamos fazendo na verdade é criando um novo objeto e colocando o endereço desse novo objeto na variável `posicoes`. Só que em lugar algum fizemos a atualização desse novo endereço para os objetos `tela` nem `jogadores`. Isso indica que a primeira versão da lista ainda existe na memória e que alguém ainda deve estar usando esse objeto inicial. Uma alternativa é limpar cada elemento da nossa matriz da seguinte forma:

```
if controle.entrada == "y":

    for i in range(3):
        for j in range(3):
            posicoes[i][j] = " "

    controle.pos_y = 0
    controle.pos_x = 0
    jogadores.vencedor = None
    jogadores.fim_de_partida = False
    tela.reiniciar_tela()
```

Não estamos criando um novo objeto e, sim, trabalhando com cada elemento contido dentro da matriz `posicoes`. Com essa modificação nosso jogo passa a funcionar corretamente. Outra alternativa é a de recriar todos os objetos que definem o jogo:

```
if controle.entrada == "y":
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
    controle = Controle(stdscr=stdscr)
    tela = Tela(stdscr=stdscr, posicoes=posicoes)
    jogadores = Jogadores(controle=controle, posicoes=posicoes)
    tela.reiniciar_tela()
```

Os objetos anteriores ficaram na memória, porém, depois de um tempo, não definido, o interpretador verificará que eles não estão mais sendo usados e, ao final, ele excluirá esses objetos antigos. O problema com esse modelo é que o desenvolvedor não tem controle sobre quando o interpretador vai coletar esse lixo. Em algumas situações isso poderá gerar um uso desnecessário de memória. Dependendo do que você estiver desenvolvendo e de quanta memória o computador tiver, isso poderá gerar algum tipo de travamento. Para o nosso jogo, o ideal é continuar trabalhando com os mesmos objetos enquanto o jogo estiver em execução.

A versão completa da função `main` está a seguir:

```
def main(stdscr):
    posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

    controle = Controle(stdscr=stdscr)
    tela = Tela(stdscr=stdscr, posicoes=posicoes)
    jogadores = Jogadores(controle=controle, posicoes=posicoes)
    tela.reiniciar_tela()

    jogador_x = 0
    jogador_o = 0
```

```

while True:
    controle.espaco_do_tabuleiro()
    if jogadores.fim_de_partida is False:
        if controle.entrada == "\n":
            jogadores.jogar()

        if jogadores.fim_de_partida is True:
            ganhador = jogadores.vencedor
            if ganhador == "x":
                jogador_x += 1
            if ganhador == "o":
                jogador_o += 1

    if controle.entrada == "y":
        """
        # Gera o bug de referência
        posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
        controle.pos_y = 0
        controle.pos_x = 0
        jogadores.vencedor = None
        jogadores.fim_de_partida = False
        tela.reiniciar_tela()
        """

        #Adequado para poupar memória ao
        #longo da execução prolongada do programa
        for i in range(3):
            for j in range(3):
                posicoes[i][j] = " "

        controle.pos_y = 0
        controle.pos_x = 0
        jogadores.vencedor = None
        jogadores.fim_de_partida = False
        tela.reiniciar_tela()

        """
        #Cria novos objetos do jogo a cada reinicialização da
        partida
        #Ao longo do tempo poderá gerar muito lixo na memória
        posicoes = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]
        controle = Controle(stdscr=stdscr)

```

```

        tela = Tela(stdscr=stdscr, posicoes=posicoes)
        jogadores = Jogadores(controle=controle, posicoes=pos
icoes)

        tela.reiniciar_tela()
        """

    if controle.entrada == 'h':
        tela.ajuda()
    else:
        tela.tabuleiro(controle)
        tela.placar(jogador_x, jogador_o)
        if jogadores.fim_de_partida is True:
            tela.fim_de_jogo(jogadores.vencedor)
            controle.cursor()

if __name__ == "__main__":
    initscr()
    wrapper(main)

```

Note que aqui estamos mesclando Programação Estruturada - ao criar a função `main` que gerenciará o loop principal do jogo - com Programação Orientada a Objetos. Ou seja, para um dado problema podemos usar composição de paradigmas de programação distintos.

6.4 CONCLUSÃO

Neste capítulo organizamos o projeto do Jogo da Velha dentro do paradigma de Programação Orientada a Objetos. Vimos também como criar os métodos `getters` e `setters` e operá-los de forma implícita. Também foi apresentada a forma como objetos são repassados para funções e métodos, através de valores de referência do local em que um objeto existe.

No próximo capítulo faremos uma introdução à Programação Funcional em Python.

PROGRAMAÇÃO FUNCIONAL

No capítulo 2, vimos os elementos básicos da linguagem Python. Agora vamos tratar de temas um pouco mais avançados como compreensões e geradores, que são elementos importantes no desenvolvimento de programas baseados no paradigma de Programação Funcional em Python. Com isso, poderemos ter uma visão mais ampla sobre as diversas formas de se resolver problemas usando uma linguagem de programação, tendo mais uma ferramenta poderosa no nosso arcabouço intelectual.

7.1 ELEMENTOS DE PROGRAMAÇÃO FUNCIONAL

De acordo com Mertiz, autor do livro *Functional Programming in Python*, de 2015, a Programação Funcional é caracterizada pelos seguintes elementos:

1. Funções são objetos de primeira classe. Dessa forma, a manipulação de dados fica a cargo de funções, tal que possamos passar funções como argumentos de outras funções.

2. Uso de chamadas recursivas são elementos primários de estrutura de controle, tal que em algumas linguagens não existem outras formas de construir laços de repetição (ausência do comando *while*).
3. O foco maior está no processamento de iteráveis, como listas.
4. Programação Funcional trabalha com avaliações de expressões no lugar de declarações. Em vez de lista de comandos, temos expressões que remetem a expressões matemáticas.
5. O foco da computação em Programação Funcional está no que deve ser computado e não em como será computado.
6. Programação Funcional utiliza funções de ordens superiores, no sentido de que funções operam em funções, podendo esse processo ocorrer de forma sucessiva.
7. Avaliação preguiçosa (*Lazy evaluation*): criar sequências infinitas sem estourar a memória.

Como exemplo de contrapartida de programa estruturado *versus* programa funcional em Python, podemos usar o cálculo fatorial.

Neste primeiro exemplo, estamos utilizando Programação Estruturada pura para o cálculo fatorial.

```
n = int(input("Digite um número inteiro positivo: "))
i = 1
n_fat = 1
while i <= n:
    n_fat = n_fat * i
    i += 1

print("O valor de {0}! é {1}".format(n, n_fat))
```

Na primeira linha do código, atribuímos a `n` um valor inteiro digitado pelo usuário. Em seguida, criamos o contador `i` e a variável `n_fat` que armazenará o valor do cálculo fatorial. Por fim, utilizamos um `while` para operar repetidamente, enquanto `i` for menor ou igual ao valor do número armazenado em `n`.

A versão funcional desse código em Python pode ser escrita com o uso da função `lambda` e chamadas recursivas:

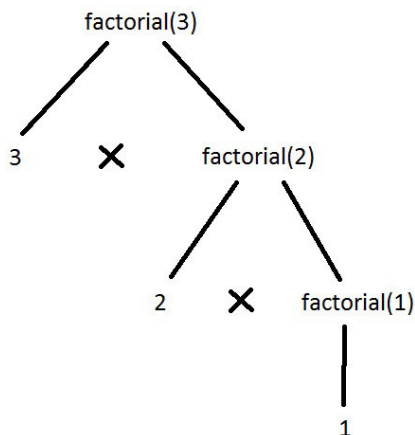
```
n = int(input("Digite um número inteiro positivo: "))
fatorial = lambda x: (x <= 1 and 1) or fatorial(x - 1) * x
print("O valor de {0}! é {1}".format(n, fatorial(n)))
```

Nesse segundo exemplo, estamos criando a função `fatorial` usando o operador `lambda`, que vai receber `x` como argumento. A primeira parte da função `(x <= 1 and 1)` faz o seguinte: quando a função recebe `x`, a expressão verifica se `x` é menor ou igual a 1. Caso isso seja verdade, o interpretador analisa a segunda parte dessa primeira expressão. Como temos apenas o número 1, sem nenhuma outra comparação para ser analisada, a função retornará o número 1. Note que, como a primeira expressão antes do `or` já foi verdadeira, o interpretador vai ignorar o que vem depois do `or`, finalizando a chamada da função.

Caso o valor de `x` seja maior que 1, a primeira parte da expressão já será falsa, nesse caso, o interpretador Python vai avaliar a segunda expressão `fatorial(x - 1) * x`. Aqui estamos fazendo uma chamada recursiva à função `fatorial`, ou seja, estamos chamando a função `fatorial` dentro da própria função `fatorial`. Note que essa chamada só será finalizada quando o valor de `x-1` for igual a 1. Só a partir dessa última chamada é que iremos efetivamente calcular o valor do número fatorial. Observe que aqui temos uma única expressão para realizar o cálculo

fatorial, `fatorial = lambda x: (x <= 1 and 1) or fatorial(x - 1) * x`, em vez de uma sequência de comandos, como foi realizado no primeiro exemplo.

Nesse caso, as chamadas são semelhantes a construção de uma árvore, tal como na figura a seguir:



Observe que a chamada partirá da raiz da árvore, com o `fatorial(3)`, seguindo até a folha mais profunda, com o valor de `x=1`. Ao chegar à última folha, todo o cálculo fatorial poderá ser realizado, de baixo para cima, até chegar à raiz da árvore.

Um outro exemplo de código funcional em Python é o que resolve o seguinte problema: escreva um código que avalia números de 1 até 100. Se o número for múltiplo de 2, o programa deve exibir na tela a palavra *Buzz*; caso o número seja múltiplo de 3, o programa deve exibir a palavra *Fizz*; caso o número seja múltiplo de 2 e de 3 ao mesmo tempo, o programa deve exibir a palavra *FizzBuzz*. Se o número não for múltiplo nem de 2 ou de 3, o programa deve exibir o próprio número.

Usando Programação Funcional em Python, temos a seguinte possível solução:

```
from functools import reduce

buzz_fizz = lambda x: "Fizz" * ( x % 3 == 0 ) + "Buzz" * ( x % 2 == 0 ) or str(x)
string_newline_from_list = lambda x,y: x + "\n" + y
resultado = reduce(string_newline_from_list, map(buzz_fizz, range(101)))
print(resultado)
```

Na primeira linha do código anterior, importamos a função `reduce`. Essa função faz operações sucessivas em uma lista de tal forma que a reduz a um único valor, de acordo com a primeira função passada como parâmetro para a `reduce`. Em seguida, criamos a função `buzz_fizz`. Na primeira parte da expressão, `"Fizz" * (x % 3 == 0)`, estamos avaliando se o resto da divisão de `x` por 3 é igual a 0 e multiplicando esse resultado pela string `"Fizz"`.

Se `x` for múltiplo de 3, o termo entre parênteses será verdadeiro e o resultado de `(x % 3 == 0)` terá valor igual a 1 (equivalente a `True`). Caso `x` não seja múltiplo de 3, o valor da operação entre parênteses terá valor igual a 0 (equivalente a `False`). Assim, multiplicando `"Fizz"` por 0, teremos uma string vazia e `"Fizz"` por 1 teremos a própria String.

Na segunda parte da expressão fazemos algo similar, porém comparando o resto da divisão de `x` por 2, `"Buzz" * (x % 2 == 0)`. Ou seja, se `x` for múltiplo de 2, teremos a string `"Buzz"`, do contrário teremos uma string vazia. Se o número for tanto múltiplo de 2 quanto de 3, o resultado da expressão `"Fizz" * (x % 3 == 0) + "Buzz" * (x % 2 == 0)` será a string

"FizzBuzz" . Caso `x` não seja múltiplo nem de 2 ou de 3, o resultado da primeira expressão será uma string vazia.

Em Python, strings vazias são consideradas como `False` , assim como acontece para os números 0 e -1. A expressão antes do `or` será do tipo `False` e o interpretador passará a analisar o segundo termo, que é o `str(x)` . Nessa última parte da expressão, estamos fazendo a conversão do número inteiro para um elemento do tipo string. Assim, a função `buzz_fizz` retornará o número na forma de string, caso a entrada não seja múltiplo nem de 2 ou de 3. A função `string_newline_from_list` faz simplesmente a concatenação dos caracteres de entrada `x` e `y` com o caractere que representa nova linha, `"\\n"` .

Por fim, fazemos operações de funções sobre funções, com a finalidade de retornar uma string contendo o resultado esperado, `resultado= reduce(string_newline_from_list, map(buzz_fizz, range(101)))` . A função `map(buzz_fizz, range(101))` vai criar uma lista, em que cada elemento será dado pela operação da função `buzz_fizz` sobre os elementos da lista de números inteiros gerados pela função `range(101)` . Em seguida, passamos como argumento a função `string_newline_from_list` e o resultado da `map(buzz_fizz, range(101))` para a função `reduce` .

Nesse caso, a função `reduce` fará com que função `string_newline_from_list` opere sobre os elementos da lista `map(buzz_fizz, range(101))` , fazendo a concatenação de todos os elementos dela, gerando uma única string. Essa string será exibida na tela, usando a função `print` . Como exemplo de resultado, para valores de 2 até 10, teremos a seguinte saída:

```
1
Buzz
Fizz
Buzz
5
FizzBuzz
7
Buzz
Fizz
Buzz
```

Vale ressaltar que Python não é uma linguagem puramente funcional, mas nada impede de mesclar elementos de paradigmas diferentes para criar programas melhores.

Ao longo deste capítulo, vamos abordar em detalhes o que são compreensões e geradores. As compreensões nos permitem focar em "sobre o que é que estamos processando", em vez de em "como estamos processando", que é um elemento importante em Programação Funcional. Veremos que, com compreensões, acabamos por escrever códigos que são muito mais próximos de expressões do que de comandos declarativos. Já com o uso de geradores, poderemos focar não só em "o que estamos processando" como também obter o efeito chamado de *Lazy evaluation*, em que poderemos processar grande volume de dados, sem nos preocuparmos com o estouro de memória.

7.2 COMPREENSÕES

Um termo corriqueiro em Python são os *comprehensions*, ou as compreensões. Em geral, eles se referem a construções alternativas ao tradicional laço usando o `for in`. São estruturas que mais se aproximam da linguagem usada em matemática.

List comprehensions

Os *list comprehensions* formam uma combinação de estrutura condicional e laço, o qual é inspirada na forma como os matemáticos usam para definir listas e conjuntos. Por exemplo, podemos representar o conjunto dos números naturais formados por quadrados perfeitos como:

$$S = \{x^2 \mid x \text{ em } \{0, 1, 2, 3 \dots\}\}$$

Já o conjunto formado por quadrados perfeitos contendo apenas números ímpares pode ser escrito como:

$$M = \{x \mid x \text{ em } S \text{ se } x \text{ é par}\}$$

De forma literal, para S podemos escrever: S é formado por x ao quadrado para x contido nos números naturais, lembrando que a \mid representa o termo "tal que" ou "para". Já para M temos: M é formado por x para valores de x em S , porém somente os números pares.

Antes de seguir, é interessante conhecer a função embutida `range`. No Python 2.x, essa função retorna uma lista com valores dentro de um determinado intervalo, até um valor final menos um. Já no Python 3, essa função retorna um gerador, que permite iterações tal como a lista.

```
>>> print(list(range(4)))
[0, 1, 2, 3]
>>> print(list(range(2,5)))
[2, 3, 4]
>>> print(list(range(1, 10, 2)))
[1, 3, 5, 7, 9]
```

No código anterior, usamos o `range` para criar uma lista com

valores de 0 a 3, no segundo caso criamos uma lista com valores que vão de 2 até 4. Já no último caso, criamos uma lista que vai de 1 até 9, mas note que o terceiro parâmetro do `range` foi o número 2, e nesse caso criamos uma lista com números que pulam de 2 em 2.

A diferença do `range` como gerador e como `range` como lista é que, ao se criar a lista, criamos um espaço em memória para guardar todos os valores dentro do intervalo. Já o gerador calcula sob demanda o valor que será usado. Por exemplo, uma lista com números de 0 a 1000 reservará um espaço na memória para todos esses valores, já o gerador vai ser capaz de criar um número de cada vez, sem a necessidade de alocar memória para todos os valores de uma só vez. Mais à frente entraremos em detalhes sobre geradores e isso ficará mais claro.

Voltando ao *list comprehension*, considerando agora somente os naturais de 0 até 9, podemos converter a primeira notação para a linguagem Python como:

```
>> S = [x**2 for x in range(10)]
>>> print(s)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

O que fizemos foi criar uma lista somente com os números naturais formados pelos quadrados perfeitos. Esse código é equivalente ao seguinte (em Programação Estruturada):

```
>>> S = []
>>> for x in range(10):
...     tmp = x ** 2
...     S.append(tmp)
>>> print(s)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Inicialmente criamos `S` como sendo uma lista vazia. Em

seguida, fizemos uma iteração sobre um intervalo de números que vai de 0 a 9, usando a função `range` para isso. Cada valor gerado pelo `range` é atribuído à variável `x`. Em seguida, criamos a variável auxiliar `tmp` na qual armazenamos o valor de `x` ao quadrado, dado por `x ** 2`. Finalmente, colocamos o valor de `tmp` ao final da lista usando o método `append`. Um ponto importante aqui é que, ao evitar a variável auxiliar e o uso do método `append`, a execução do `list comprehension` ocorre de forma mais rápida, fornecendo uma otimização para o código em tempo de execução.

Agora podemos combinar a operação de laço com operação condicional e criar um filtro para uma lista:

```
>>> M = [x for x in S if x % 2 == 0]
>>> print(M)
[0, 4, 16, 36, 64]
```

No caso anterior, criamos uma nova lista com elementos que vêm da lista `S`, porém, ao final escrevemos `if x % 2 == 0`, ou seja, se o resto da divisão de `x` por 2 for exatamente 0. Nesse caso estamos pegando somente os quadrados perfeitos que são múltiplos de 2, ou pares.

O código anterior é equivalente ao seguinte código em programação estruturada:

```
>>> M = []
>>> for x in S:
...     if(x % 2 == 0):
...         M.append(x)
>>> print(M)
[0, 4, 16, 36, 64]
```

Podemos usar o `list comprehension` para filtrar palavras por tamanho em um texto, por exemplo:

```
>>> texto = "O rato roeu a roupa do rei de roma"
>>> palavras = texto.split(" ")
>>> palavras3 = [ti for ti in palavras if len(ti) > 3]
>>> print(palavras3)
['rato', 'roeu', 'roupa', 'roma']
>>> novoTexto = " ".join(palavras3)
>>> print(novoTexto)
'rato roeu roupa roma'
```

Aqui criamos a string contida na variável `texto`, em seguida, usamos o método `split(" ")` para transformar a string numa lista de palavras. Nesse caso, dentro do `split` definimos que o separador da string é o espaço em branco. Em `palavras3` usamos um `list comprehension` para criar uma nova lista com elementos de palavra, porém somente seriam usadas palavras com mais de 3 caracteres. Ao final, usamos essas palavras filtradas para criar uma nova string, usando o comando `" ".join()`.

Podemos também usar *list comprehension* no lugar de laços aninhados:

```
>>> m = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> s = []
>>> for x in m:
...     for k in x:
...         s.append(k ** 2)
>>> print(s)
[0, 1, 4, 9, 16, 25, 36, 49, 64]
```

Note que `m` é uma lista de listas, assim cada valor de `x` será uma das sublistas de `m`. Em seguida, para cada item da sublista `x` fizemos a operação de adicionar ao `s` o valor de `k` ao quadrado. Essa combinação de um `for` dentro do outro é que chamamos de laço aninhado. Uma equivalência desse tipo de aninhamento é o seguinte:

```
>>> m = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> s = [k ** 2 for x in m for k in x]
```

```
>>> print(s)
[0, 1, 4, 9, 16, 25, 36, 49, 64]
```

Nesse caso, é preciso ter muito cuidado com a legibilidade do código. Veja que adicionamos o `for` mais externo primeiro (`for x in m`), em seguida adicionamos o `for` interno (`for k in x`).

Dictionary Comprehensions

Tal como com as listas, podemos criar dicionários de forma dinâmica usando o mesmo conceito que o do *list comprehension*. Esse método está presente no Python a partir da versão 2.7. Como exemplo, considere o caso de uma lista contendo os dias da semana de forma ordenada. Queremos converter esses dias em um dicionário, no qual a chave seja um inteiro:

```
>>> dias = ["domingo", "segunda", "terça", "quarta", "quinta", "s
exta", "sabado"]
>>> dias = {i: dias[i - 1] for i in range(1, len(dias) + 1)}
>>> print(dias)
{1: 'domingo', 2: 'segunda', 3: 'terça', 4: 'quarta', 5: 'quinta'
, 6: 'sexta', 7: 'sabado'}
```

Na primeira linha, criamos uma lista contendo strings representando os dias da semana. Na segunda, fizemos um novo dicionário, e como queremos que o primeiro dia tenha a chave 1, fizemos a função `range` gerar um intervalo que começa em 1 e segue até o número total de elementos da lista. Note que o índice da lista começa com 0 e não com 1, por isso modificamos o código para pegar o dia correto fazendo `dias[i - 1]`.

7.3 GERADORES

Os *generators*, ou geradores, são funções capazes de devolver

resultados sob demanda, retornando iteradores, ou seja, o resultado só será calculado quando ele for efetivamente requisitado. São ferramentas muito poderosas e extremamente úteis quando temos que trabalhar com volumes grandes de dados. Por exemplo, quando se deseja fazer streaming de dados, enviando informações em tempo real do servidor para o usuário. Em Python temos as funções geradoras e as expressões geradoras.

Funções geradoras

No capítulo 2 vimos que uma função nada mais é do que um trecho de código que pode ser usado mais de uma vez, sendo que ela poderá retornar um conjunto de valores ou o `None`. Em linguagens procedurais como FORTRAN e C, esses trechos de código também são conhecidos como sub-rotinas. Assim, uma vez executado o trecho de código, ele retornará um determinado valor e finalizará a função. Só que em alguns casos é interessante guardar alguma informação sobre algo que aconteceu dentro da função e reutilizar a mesma função para fazer um segundo passo. Às vezes, o que se deseja é que a função não retorne apenas um valor e encerre a operação, ou seja, desejamos que uma dada função retorne um valor, porém sem interromper o seu processamento. A ideia é ir coletando os dados da função em tempo de execução.

Para entender melhor a ideia da função geradora vamos considerar o cálculo da sequência de Fibonacci. A sequência de Fibonacci tem a seguinte forma: 0, 1, 1, 2, 3, 5, 8, 13,... Vemos que, a partir do terceiro termo, a sequência é formada pela soma dos dois valores anteriores ao correspondente número da série. Por exemplo, $2 = 1 + 1$, $3 = 2 + 1$, $5 = 3 + 2$. Assim podemos escrever a sequência de Fibonacci como $F_n = F_{n-1} + F_{n-2}$, sendo que $F_0 = 0$ e

$$F_1 = 1;$$

Usando uma função podemos calcular o n -ésimo elemento da série de Fibonacci da seguinte forma:

```
>>> def fibonacci(n):  
...     a, b = 0, 1  
...     for i in range(n):  
...         a, b = b, a + b  
...     return a  
>>> sequencia = [fibonacci(n) for n in range(10)]
```

A função `fibonacci` receberá como entrada o elemento da série que queremos calcular, por exemplo, o sexto elemento da série de Fibonacci é o número 8. Dentro da função, fizemos com que os primeiros elementos da série fossem 0 e 1. Note que podemos fazer atribuição de valores a mais a uma variável de uma só vez, usando `,`. Na função fizemos com que o valor de `a` fosse igual a 0, e o valor de `b` igual a 1, usando o comando `a, b = 0, 1`. Em seguida, fizemos o cálculo da série de Fibonacci até o elemento de número `n` usando o laço `for in`. Veja que `b` será sempre o valor mais recente da série, logo, quando calculamos o novo número, o `a` acaba se tornando o `b` e para o `b` atribuímos a soma do anterior com o valor do número mais anterior. Essa troca, ou *swap*, é feita pela operação de atribuição de valores simultâneos `a, b = b, a + b`. Como queremos apenas o valor do n -ésimo termo da série, retornamos apenas o valor de `a`.

Em seguida, usamos um *list comprehension* para obter os 10 primeiros elementos da série de Fibonacci. Veja que, para obter o 10º elemento da série, é preciso fazer o cálculo de toda a série até encontrar o valor que queremos. Agora imagine que queremos construir uma série com essa função contento um milhão de elementos. O que veremos é que, se não estourarmos o limite de

memória da máquina, o tempo para calcular esses elementos será relativamente longo.

Uma alternativa para isso é que a cada passo dentro do laço a função retorne o valor de `a` e só em seguida ele continue o cálculo do próximo valor da sequência somente se for requisitado. Ou seja, nossa função deve retornar um valor, mas ao mesmo tempo precisamos de mecanismos que armazenem informações do que já foi calculado anteriormente. Uma função com o poder de gerar valores sob demanda e ao mesmo tempo reter informações sobre sua chamada anterior é conhecida em Python como função geradora. Em vez de usar a palavra `return` usamos a palavra `yield`. Vamos modificar a função `fibonacci` para que ela seja uma função geradora:

```
>>> def fibonacci(n):
...     a, b = 0, 1
...     count = 0
...     while count < n:
...         yield a
...         a, b = b, a + b
...         count += 1
>>> sequencia = [f for f in fibonacci(10)]
```

Nesse exemplo, nós trocamos o laço do tipo `for in` por um laço do tipo `while`. Também adicionamos um contador, que mapeará a posição do valor corrente do elemento da sequência que estamos calculando. Dentro do bloco do `while`, adicionamos o `yield a`, o que indica que a primeira coisa que a função deve fazer é retornar o valor de `a`. Normalmente, após um retorno a função simplesmente não calcula o que está abaixo. Porém com a aplicação do `yield`, em vez de `return`, nossa função vai continuar fazendo a operação que está logo abaixo, ou seja, calcular o próximo valor da série.

Ao final informamos ao contador que já estamos passando para o próximo valor da série. Agora observe que estamos criando uma lista da seguinte forma: `[f for f in fibonacci(10)]`, ou seja, adicione à lista o valor de `f` sendo que os valores de `f` estão na função `fibonacci` que vai gerar 10 elementos. Note que fomos capazes de colocar a nossa nova função exatamente no lugar em que ficava a função `range`. Podemos simplificar ainda mais o código usando a função de conversão de tipo `list`, ou seja, podemos converter diretamente a função `fibonacci(10)` em lista usando:

```
>>> sequencia = list(fibonacci(10))
```

Fazendo a comparação do tempo de execução para as duas implementações do cálculo dos dez mil primeiros termos da série de Fibonacci usando os dois métodos, em uma máquina com processador Intel i7 de 7ª geração, vemos que para a primeira implementação o tempo total foi de 6.07 segundos, já o cálculo com a segunda implementação, usando geradores, levou 0.004 segundos. Ou seja, o segundo método foi mais de 1500 vezes mais rápido.

Mais adiante, veremos como usar funções geradoras para criar efeitos de interação dinâmica entre dados que estão sendo processados no servidor e a página que está sendo exibida para o cliente, por exemplo, a construção de gráficos interativos e atualizados em tempo real.

Passando argumentos

Quando escrevemos um programa em Python, que deverá ser executado em sistemas tipo unix, na primeira linha do arquivo

indicamos que o programa `env` deve chamar o `python3.6`. Fazemos isso com o comando `#!/usr/bin/env python3.6`. Ao usar o terminal, podemos ter um comando, que em essência é um programa, que recebe um argumento e executa uma operação. Os módulos em Python podem atuar também dessa forma. Vamos criar um módulo que recebe como argumento um inteiro. Esse inteiro representará quantos elementos da série de Fibonacci queremos que o nosso programa exiba. Para transformar nosso módulo em um script que recebe argumentos, vamos usar o módulo embutido chamado `sys`. Nosso arquivo `fibonacci.py` terá o seguinte conteúdo:

```
#!/usr/bin/env python3.6
# -*- coding: UTF-8 -*-
"""
Script com funções relacionadas a série de Fibonacci
"""

def fibonacci(n):
    """
    Função geradora que retorna elementos da série de Fibonacci.
    Entrada:
        n: Total de elementos a serem calculados
    Saída:
        valor correspondente da série
    """
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

if __name__ == "__main__":
    import sys
    n = int(sys.argv[1])
    fib = list(fibonacci(n))
    print(fib)
```


No código anterior, logo abaixo da definição da função, adicionamos uma *docstring*, que é uma documentação sobre o que faz a função. Nossa função pode ser usada em outros programas, usando o comando `import` tal como fazemos com os módulos embutidos. Pensando em reusabilidade, ao se escrever um programa é interessante separar os trechos em funções específicas, separando o que pode ser utilizado em outros programas daquilo que será executado exclusivamente pelo módulo como um programa principal. Para fazer isso, criamos um novo bloco com o conteúdo `if __name__ == "__main__":`. A variável `__name__` vai conter o texto `__main__` se executarmos o módulo como um programa principal. Agora, se estivermos fazendo a chamada do módulo em outro módulo, a variável `__name__` conterá o nome do módulo em que ela foi definida. Assim, conseguimos isolar funções que podem ser reutilizadas.

Dentro do bloco que queremos executar como programa corrente importamos o módulo `sys`. Em seguida, usamos o `argv`, que vai fornecer uma lista com os argumentos passados ao script. Note que pegamos o argumento da posição 1. Fizemos isso pois o argumento da posição 0 sempre será o nome do arquivo que contém o nosso programa. Os argumentos são capturados na forma de string, assim usamos a função `int` para converter o primeiro argumento em inteiro. Depois, criamos uma lista com a sequência de n elementos da série de Fibonacci. Finalmente, pedimos para que o programa exiba essa lista na tela.

Para executar o programa, basta abrir o terminal e seguir até o diretório onde o nosso módulo se encontra. Se o programa tiver permissão de execução, podemos rodar nosso script com `./fibonacci.py`. Por exemplo, para exibir os 10 primeiros

elementos da série fazemos:

```
$. ./fibonacci.py 10  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

ou

```
$ python fibonacci.py 10  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Vale lembrar que no caso do Linux usamos o comando `chmod +x fibonacci.py` para dar permissão de execução.

Expressões geradoras

As expressões geradoras ou *generator expressions* são muito similares ao *list comprehension*, porém, em vez de usarmos chaves `[]`, usamos parênteses `()`. A grande diferença é que no *list comprehension* estamos armazenando na memória uma lista completa de elementos, enquanto com expressões geradoras não usamos esse recurso extra de memória. Por exemplo, imagine que queremos fazer a soma dos 1000 primeiros quadrados. Usando *list comprehension* faremos o seguinte:

```
>>> quadrados = [x*x for x in range(1, 1001)]  
>>> soma_quadrados = sum(quadrados)
```

No caso anterior, primeiramente criamos uma lista com os quadrados de 1 até 1000. Essa lista é passada como parâmetro para a função embutida `sum`, que retorna a soma dos números contidos em uma lista ou geradores. Nesse caso, foi preciso alocar memória para a lista. A expressão geradora equivalente é:

```
>>> quadrados = (x*x for x in range(1, 1001))  
>>> soma_quadrados = sum(quadrados)
```

Se pedirmos para exibir na tela o valor de `quadrados` , usando `print(quadrados)` , o Python retornará `<generator object <genexpr> at 0x7fec573b1960>` . Logo, foi guardada a informação sobre o processo de geração dos quadrados de 1 até 1000 e não os valores dos quadrados em si. É extremamente útil ao se utilizar as funções `sum` , `max` (retorna maior valor) e `min` (retorna menor valor), por exemplo.

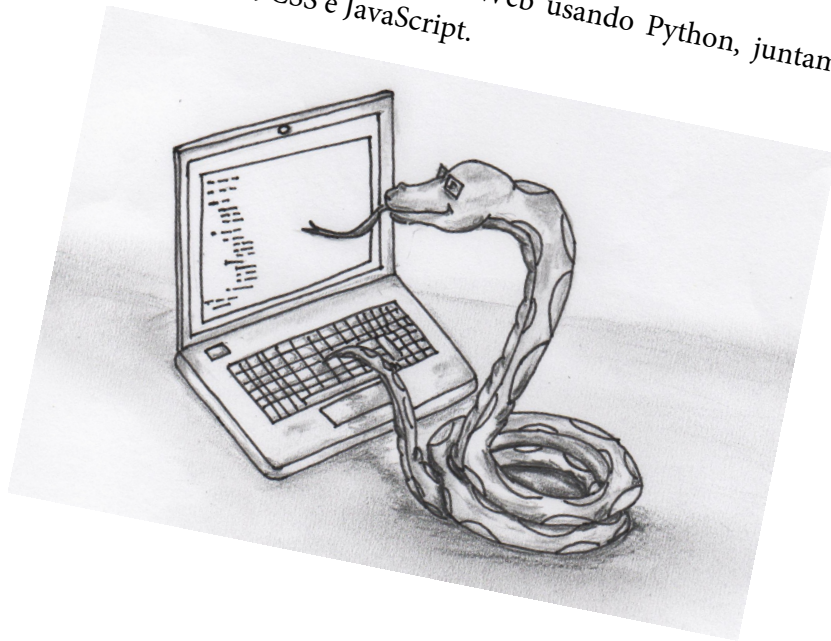
Os geradores são mecanismos que permitem a otimização de código tanto em tempo - a chamada *complexidade de tempo* - quanto otimização de uso de memória - conhecida como *complexidade de espaço*. Isso ocorre, por exemplo, ao permitir a reutilização de informações prévias de uma função e ao fazer com que uma determinada operação ocorra somente sob demanda.

7.4 CONCLUSÃO

Neste capítulo, vimos alguns tópicos avançado da linguagem, como compreensões, que permitem criar listas e dicionários de forma dinâmica, combinando estrutura condicional e de laço em um único elemento. Também foi apresentado o conceito de geradores, que retornam objetos iteráveis, sob demanda, com baixo custo de uso de memória.

Desenvolvimento Web com Python

licações Web usando Python, juntamente com o HTML5, CSS e JavaScript.



APLICATIVOS WEB

No início da internet, tínhamos basicamente páginas de texto com links e hiperlinks, que permitiam navegar ao longo do próprio texto ou ir até outras páginas, bastando clicar em palavras-chaves. Hoje conversamos com pessoas do mundo inteiro através de redes sociais, fazemos compras em lojas eletrônicas, realizamos transferência de dinheiro por meio de sites e uma série de outras tarefas. Tudo isso a apenas um clique de distância e na maioria das vezes usando um *smartphone*. Esses sites modernos são os chamados aplicativos Web, os quais nada mais são do que programas, que em geral, deixam as tarefas mais complexas para serem executadas em um servidor, enquanto, no aparelho do usuário, roda apenas o necessário para gerar interfaces gráficas.

Dessa forma, uma aplicação Web possui basicamente três camadas: i) camada de interação com o usuário final, rodando na máquina do usuário através de um *browser*; ii) a camada que está por trás da aplicação, rodando em um servidor; iii) a camada em que os dados e informações do usuário são armazenados, ou seja, no banco de dados do sistema, que também se encontra no servidor.

A camada que roda no servidor é conhecida como *back-end*, que em tradução livre seria algo como "o que fica por trás". Já a

camada que roda na máquina do usuário é chamada de *front-end* e representa tudo aquilo que é codificado para interagir diretamente com o usuário.

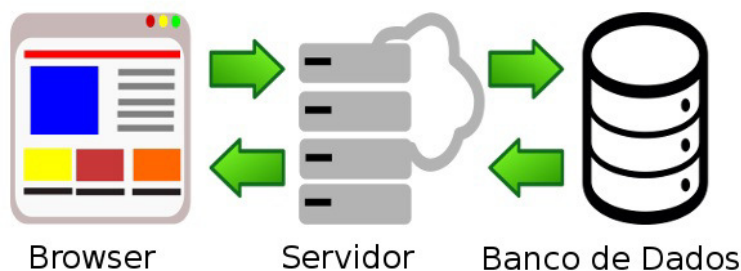


Figura 8.1: Camadas que formam um aplicativo Web e suas interações

Para o desenvolvimento *front-end*, é comum o uso de tecnologias como HTML5, CSS e JavaScript. Já para o *back-end* temos o PHP, Perl, Java, C#, Python, Ruby e outras linguagens de programação. É comum ter pessoas especializadas no desenvolvimento *back-end* ou *front-end*, porém, quando um programador possui habilidades para desenvolver tanto o que ocorre a nível de servidor quanto a nível de usuário, esse é chamado de desenvolvedor *full stack* (literalmente significa pilha completa).

A grande vantagem é que hoje temos ferramentas que nos ajudam a construir aplicações Web, que são os chamados *frameworks*.

8.1 FRAMEWORKS WEB

Como vimos, para que um aplicativo Web funcione é preciso

ocorrer um diálogo entre as interações do usuário, através do *browser*, ou navegador, com o código que está rodando no servidor. Mas será que podemos encontrar características comuns no meio desse zoológico de aplicativos Web que nos cercam? Ou seja, será que ao construir um aplicativo Web eu posso reaproveitar coisas para criar um terceiro?

Quando observamos que estamos usando coisas comuns para criar aplicativos completamente diferentes, costumamos juntar essas partes em uma estrutura única. É essa estrutura, formada por um conjunto de características comuns a uma grande variedade de programas, que chamamos de *framework* (estrutura). Assim, um *framework* Web é formado por um conjunto de ferramentas que permite criar sites mais complexos, transformando-os em softwares completos, que respondem de forma dinâmica às interações do usuário.

Já o conjunto de características que cada *framework* tem a oferecer vai variar bastante de um para outro. Porém, independente da linguagem de programação em que foram escritos, eles podem ser divididos em duas categorias: i) *microframeworks*; ii) *frameworks full stack*.

Um *framework* é chamado de *full stack* quando possui um número muito grande de componentes embutidos, fazendo com que não seja necessário adicionar bibliotecas externas para se construir uma aplicação Web. Já os *microframeworks* trazem apenas o essencial.

No caso de desenvolvimento de aplicações Web, existe uma lista grande de *frameworks* escritos em Python. Dentre eles, podemos citar o Django e o Web2py, que são *full stack*. Já

exemplos de microframeworks são o Flask e o Morepath. A escolha de qual usar vai depender do perfil de sua aplicação.

Por exemplo, o Django e o Web2py são muito bons para o desenvolvimento de aplicações com foco no uso e controle de banco de dados. Além disso, ambos possuem uma excelente interface administrativa para gerenciar banco de dados e aplicações. Já o Flask pode ser usado para desenhar qualquer tipo de aplicação, porém, para a criação de algo mais complexo será necessário importar e desenvolver soluções externas ao framework. Em geral, não dá para dizer que uma solução é melhor que a outra, tudo vai depender do perfil do projeto. Sites de jornais por exemplo, costumam usar frameworks como Django, porém sites como Pinterest, que pretendem ter funções específicas e modulares, usam o Flask.

Para poder compreender melhor o desenvolvimento de aplicações Web é interessante partir de um *microframeworks*. Assim, a partir do próximo capítulo vamos nos aprofundar no processo de construção de aplicativos Web usando Flask.

8.2 CONCLUSÃO

Neste capítulo, vimos os conceitos fundamentais do que é uma aplicação Web, quais as diferenças entre *microframeworks* e *frameworks full stack* e quais pontos considerar ao escolher entre um ou outro. Nos próximos capítulos vamos abordar a criação de uma aplicação Web, com integração com banco de dados, usando Python, Flask, JavaScript, CSS e HTML5.

APLICAÇÕES WEB COM PYTHON E FLASK

Flask é um microframework Web, que está sob licença BSD, baseado no *Werkzeug toolkit* e no gerador de templates Jinja2. No momento em que este livro foi escrito, a versão mais recente era a 0.12.2. Por ser um microframework, ele não contém uma camada de abstração de banco de dados própria, nem uma série de componentes prontos, como uma interface administrativa, presentes nos frameworks Django e Web2py. Porém, com ele é possível criar aplicações mais modulares.

Além disso, é uma excelente ferramenta para a criação de microsserviços e APIs REST (*Representational State Transfer*, Transferência de Estado Representacional). As APIs REST são frequentemente usadas para acesso a um serviço Web. Como exemplo, podemos citar o acesso a informações de um aplicativo nativo de smartphones a uma base de dados distribuídas, como um aplicativo de rede social, que aproveita as facilidades do sistema operacional do aparelho, como recursos de câmera, mas que ainda precisa acessar os dados da conta do usuário remotamente. Na API REST, podemos criar um serviço específico que retorna um JSON ou um XML, com dados de uma requisição. Esses dados são tratados pelo aplicativo nativo, sem se preocupar com a forma

como foram extraídos do servidor remoto.

O Flask é uma excelente forma de se iniciar no desenvolvimento Web com Python. Atualmente, ele é mantido por um grupo de entusiastas, chamado Pocoo, criado por Armin Ronacher.

Para entender a estrutura de um aplicativo Web escrito em Flask, vamos criar um "Hello World". O primeiro passo será o de criar o arquivo chamado `oiMundo.py` e, dentro dele, adicionar o seguinte código:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def oiMundo():
    return "Oi Mundo!"

if(__name__ == "__main__"):
    app.run()
```

Na primeira linha, importamos a classe `Flask` do módulo principal `flask`. Em seguida, instanciamos a classe `Flask`, passando para ela a variável `__name__`, criando o objeto `app`. Aqui vale lembrar que a variável `__name__` representa o nome do módulo corrente. Dessa forma, o que estiver abaixo do `if` só será executado se o módulo corrente for o principal, `__main__`, em execução.

Veja que a função `oiMundo` foi decorada com o método `route`, dado pelo código `@app.route("/")`. Esse decorador indica a rota que dará acesso ao conteúdo gerado pela função `oiMundo`. Nossas rotas são as URLs usadas, por exemplo, no

navegador para acessar páginas Web, que serão representadas por funções. A função `oiMundo` retornará a string `"Oi Mundo!"`. Ou seja, o conteúdo da página Web dado pela função `oiMundo` será apenas o texto `"Oi Mundo!"`. Em seguida, verificamos se o arquivo corrente era o que realmente está sendo executado como programa principal. Para isso, usamos o código `if(__name__ == "__main__"):`. A execução do aplicativo Web é feita através do método `run` do objeto `app`, dado por `app.run()`.

Pronto, esse é o nosso primeiro aplicativo Web com Flask. Para executá-lo usamos o comando:

```
Python oiMundo.py
```

Esse programa exibirá a seguinte saída no terminal:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Agora basta abrir o navegador e digitar o endereço `http://127.0.0.1:5000/` que veremos nossa aplicação em ação. Na figura a seguir, é apresentada a página Web gerada por esse programa:

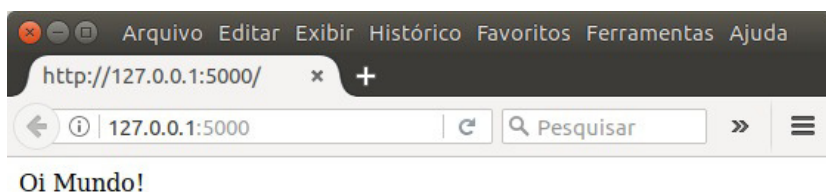


Figura 9.1: Aplicação Oi Mundo rodando no endereço <http://127.0.0.1:5000/>.

9.1 NOSSO PROJETO

Como parte prática para esta parte de desenvolvimento Web, vamos desenvolver um serviço de streaming de música chamado FPlayer. A grande vantagem é que, quando ele estiver pronto, você poderá usar essa aplicação para acessar suas músicas em todos os outros aparelhos conectados à sua rede local. Vamos construir a aplicação aos poucos, usando Python e o framework Flask para o desenvolvimento **back-end**, e construindo o **front-end** usando HTML5, JavaScript e CSS. Mais à frente veremos que podemos usar esse mesmo princípio para criar aplicativos desktop, em que a interface gráfica (*Graphical User Interface* - GUI) é feita usando as tecnologias **front-end**.

O elemento básico do nosso **front-end** será desenvolvido a

partir do seguinte código HTML, no qual importaremos as bibliotecas necessárias para usar o MaterializeCSS:

MATERIALIZECSS O MaterializeCSS é um framework front-end desenvolvido por um time de estudantes da universidade de Carnegie Mellon, nos Estados Unidos. Ele foi inspirado nos conceitos do Material Design, que é uma linguagem de design criada e desenvolvida pelo Google. A intenção do Google era o de criar um sistema de design que permita a unificação da experiência do usuário através de todos os seus produtos, em qualquer plataforma. Como ele traz essa filosofia de unificação, um site desenvolvido com esse framework será responsivo. Assim, ao se acessar a aplicação Web tanto em smartphones quanto em tablets, o usuário terá uma experiência de uso que se assemelha à de um aplicativo nativo.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
5
6      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/css/materialize.min.css">
7      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8      <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
9      <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/js/materialize.min.js"></script>
10 </head>
13 </html>
```

Na primeira linha, fizemos a declaração chamada **DOCTYPE**, que indica que o arquivo é do tipo HTML5. Na linha 2, iniciamos a estrutura do html. Na linha 3, criamos o cabeçalho do html. Na linha 4, estamos importando uma família de fontes especiais para a geração de ícones especiais do Materialize do Google. Na linha 6, estamos adicionando o framework MaterializeCSS; note que no final do atributo `href` temos o seguinte arquivo: `materialize.min.css` . Na linha 7, adicionamos um novo metadado com nome `viewport` , que permitirá que nossa página tenha a largura da tela do dispositivo que acessar o nosso site. Na linha 8, adicionamos o framework JavaScript jQuery, que é necessário para o uso do MaterializeCSS. Na linha 9, importamos os componentes JavaScript que fazem parte do próprio framework MaterializeCSS.

Com isso, fizemos a importação de todos os componentes necessários para começarmos a usar o framework CSS que será adotado neste capítulo.

9.2 GERANDO TEMPLATES DINÂMICOS

No primeiro exemplo, apenas exibimos o texto "Oi Mundo!" na nossa página. O interessante é trabalhar com HTML, mas em vez de ficar codificando HTML na forma de strings dentro do código Python, usaremos uma linguagem de template. Ao instalar o Flask, um dos requisitos é o Jinja2, que é uma linguagem específica para a criação de páginas HTML dinâmicas e de forma integrada com a linguagem Python. Para conhecer mais detalhes veja o site do projeto: <http://jinja.pocoo.org/>.

Um template Jinja nada mais é que um arquivo `.html`

contendo marcações adicionais que permitem criar elementos dinâmicos. Por padrão, o Flask procura esses arquivos na pasta `templates` localizada na raiz do projeto.

Voltando ao projeto do sistema de streaming de músicas, primeiro vamos criar a estrutura básica de pastas:

```
FPlayer
|
|----templates
|    |
|    layout.html
|    home.html
|
|----static
|    |
|    css
|    images
|    js
|----fplayer.py
```

A pasta principal do projeto, ou pasta raiz, será a `FPlayer`, e dentro dela teremos as pastas `templates`, com os arquivos `layout.html` e `home.html`. Também vamos criar a pasta `static`, a qual conterá as pastas `css`, `images` e `js` com os respectivos arquivos de imagens, CSS e JavaScript. Na pasta raiz, também está o arquivo `fplayer.py`, que terá o arquivo fonte do nosso programa em Flask.

Nosso primeiro passo será criar o arquivo de layout. Ele será responsável pela estrutura geral do site. Isso indica que todas as páginas do nosso site terão o mesmo aspecto. Assim, geramos apenas um arquivo de layout e as outras páginas vão herdar a aparência definida por esse arquivo.

O layout será o arquivo HTML mestre. Precisamos adicionar as

tags de abertura, `head` e `body` e também vamos adicionar o menu de navegação principal nesse arquivo.

Primeiramente, vamos definir a estrutura básica e adicionar o *header*. Aqui já vamos incluir a chamada do MaterializeCSS:

```
<!DOCTYPE html>
<html>

<head>
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/css/materialize.min.css">
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/js/materialize.min.js"></script>

  {% block head %}

  <title>{% block title %}FPlayer{% endblock %}</title>

  {% endblock %}

  <script type="text/javascript">
$(document).ready(function(){
  $(".button-collapse").sideNav();
  $('.materialboxed').materialbox();
});
</script>
</head>
<body>
</body>
</html>
```

Observe que foram adicionadas novas marcações, dadas pela expressão `{% %}` . Como exemplo, usamos as marcações `{% block head %}` e `{% endblock %}` . Com isso, estamos

indicando ao Jinja que estamos criando um novo bloco. Ao criar um bloco, estamos definindo que o conteúdo que estiver dentro dessa demarcação poderá ser sobrescrito por outra página. Dentro do bloco `head`, criamos outro bloco, dentro da tag `<title>``</title>`: `{% block title %}FPlayer{% endblock %}`. Com isso, poderemos criar outras páginas que herdam o nosso layout, mas cada uma delas poderá ter o seu próprio título.

Agora vamos criar o corpo do layout.

```
<body>
  <nav>
    <div class='nav-wrapper blue'>
      <a href="#" data-activates="mobile-nav" class="button
-collapse"><i class="material-icons">menu</i></a>
      <a href="#" class="brand-logo center">FPlayer</a>
      <ul id="nav-mobile" class="left hide-on-med-and-down":

        <li><a href="{ { url_for("home") }}">Home</a></li>
      </ul>
    </div>
    <ul class="side-nav" id="mobile-nav">
      <li><a href="{ { url_for("home") }}">Home</a></li>
    </ul>
  </nav>
  <div id="content" class="container">
    {% block content %} {% endblock %}
  </div>
</body>
```

Dentro de `<body></body>`, foi adicionado o menu, marcado pela tag `<nav></nav>`. O nosso menu foi construído para ser responsivo, pois criamos uma `div` que contém o menu para telas grandes. Logo abaixo dessa `div` é possível ver a tag `ul` com o id `mobile-nav`, indicando que esse será o menu exibido quando o MaterializeCSS reconhecer que a tela é de um aparelho mobile.

O ponto importante a ser salientado é a presença da expressão `{{ url_for("home") }}` no atributo `href` da tag de link `<a>`. Nesse caso, estamos informando que a URL que chamará a função `home`, escrita em código Python, deve ser gerada de forma dinâmica.

Ainda dentro da tag `<body></body>` criamos uma `div` do tipo `container` com o bloco `{% block content %} {% endblock %}`. Ou seja, vamos usar o bloco `content` para adicionar o conteúdo das páginas que vão herdar esse layout.

O conteúdo completo do arquivo de layout será:

```
<!DOCTYPE html>
<html>

<head>
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/css/materialize.min.css">
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.99.0/js/materialize.min.js"></script>
    {% block head %}

    <title>{% block title %}FPlayer{% endblock %}</title>

    {% endblock %}

    <script type="text/javascript">
    $(document).ready(function(){
        $(".button-collapse").sideNav();
        $(' .materialboxed').materialbox();
    });
    </script>
</head>
```

```

<body>
  <nav>
    <div class='nav-wrapper blue'>
      <a href="#" data-activates="mobile-nav" class="button
-collapse"><i class="material-icons">menu</i></a>
      <a href="#" class="brand-logo center">FPlayer</a>
      <ul id="nav-mobile" class="left hide-on-med-and-down":

        <li><a href="{ { url_for("home") }}">Home</a></li>
      </ul>
    </div>
    <ul class="side-nav" id="mobile-nav">
      <li><a href="{ { url_for("home") }}">Home</a></li>
    </ul>
  </nav>
  <div id="content" class="container">
    {% block content %} {% endblock %}
  </div>
  <br>
  <br>
  <br>
  <br>
  <footer class="page-footer blue center-align">
    <p class="grey-text text-lighten-4 "> E. S. Pereira,

</p>

    <p>
      <a rel="license" href="http://creativecommons.org
/licenses/by-nc/4.0/"></a> </p>
    </footer>
</body>
</html>

```

O próximo passo será o de criar o arquivo `home.html` :

```

{% extends "layout.html"%}
{% block content %}

<p> Oi Mundo </p>

{% endblock %}

```

Na primeira linha, note que foi adicionado o comando `{% extends "layout.html"%}` , no qual estamos indicando que a página `home.html` deve ser uma extensão da página `layout.html` . Em seguida, criamos o bloco `content` no qual adicionamos internamente o parágrafo `<p> Oi Mundo </p>` .

Finalmente, vamos criar o arquivo `fplayer.py` que vai usar esse template:

```
from flask import Flask, url_for, render_template

app = Flask(__name__)

@app.route("/")
def home():
    return render_template("home.html")

if(__name__ == "__main__"):
    app.run()
```

Nesse arquivo, foram chamadas as funções `url_for` , usada pelo template, e a `render_template` , que permitirá a conversa entre o Jinja e o Python. Em seguida, criamos a função `home` que será exibida na URL raiz do nosso aplicativo. Essa função retorna `render_template` , o qual tem como parâmetro o template que deve ser renderizado na máquina do usuário. Nas figuras a seguir são apresentadas as páginas resultantes, tanto para telas grandes, quanto para telas pequenas.



Figura 9.2: Versão Desktop.

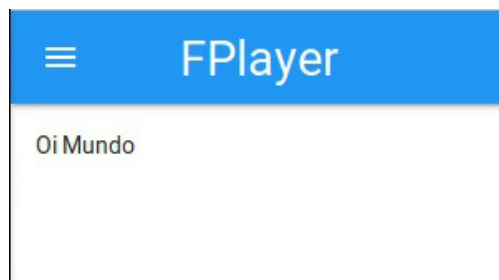


Figura 9.3: Versão Mobile.

Com isso já temos os elementos básicos para criar uma aplicação Web.

9.3 ACESSANDO AS MÚSICAS

Vamos modificar o arquivo `fplayer.py` para que nosso programa possa acessar as músicas. Nessa primeira versão, vamos adicionar a pasta `musics` dentro da pasta `static`. Usaremos o módulo nativo `glob` para listar todos os arquivos `mp3` que vamos salvar nessa pasta. Além disso, usaremos os módulos `send_file` e `request` do Flask. Para fazer o streaming de áudio, vamos criar a função `sounds`.

```
@app.route("/sounds")
def sounds():
    music = request.args["music"]
    return send_file(music, mimetype="audio/mp3")
```

No código anterior, a função `sounds` foi decorada com a rota `/sounds`, que indicará a URL de acesso aos áudios. Note que, dentro dessa função, usamos o módulo `request`, que possui o atributo `args`. O que esse módulo faz é pegar os argumentos de uma requisição, passado via URL. Por exemplo, considere a URL

`http://localhost:8000/sounds?music=static%2Fmusics%2FFuneral+March%2C+Op.+72+no.+2.mp3` . Logo após o `/sounds?` temos `music=static%2Fmusics%2FFuneral+March%2C+Op.+72+no.+2.mp3` . A URL é a requisição que estamos passando para o nosso servidor, enquanto `music` é o argumento, que tem como valor o endereço para o arquivo de áudio cujo streaming queremos fazer.

A função `send_file` permitirá fazer o streaming do arquivo, o qual será do tipo `mp3`, dado pelo comando `send_file(music, mimetype="audio/mp3")` . Nesse último comando, o argumento `mimetype` indica ao navegador o tipo de arquivo.

A função `home` será modificada para levar em conta a função de streaming de áudio:

```
@app.route("/")
def home():
    musiclist = glob.glob("static/musics/*.mp3")

    musicJ = [{'fileName': mi.split("/")[-1],
               "fileURL": url_for('sounds', music=mi)}
              for mi in musiclist]

    return render_template("home.html",
                           musicJ=musicJ)
```

Na função `home` usamos o módulo `glob` para criar uma lista contendo todos os arquivos do tipo `mp3` que estão na pasta `static/musics/` . Observe que para fazer isso usamos a expressão `*.mp3` logo após o caminho relativo da pasta em que estão os arquivos de áudio. O módulo `glob` permite encontrar todos os arquivos, ou pastas, cujo padrão case com uma expressão, de acordo com as regras usadas pela linguagem Shell do Unix. O

*.mp3 indica qualquer coisa (*) que termine com .mp3 .

Em seguida, criamos a variável `musicJ` , na qual foi utilizado um *list comprehension* para criar uma lista de dicionário, contendo o nome do arquivo e a URL gerada para o streaming de cada áudio. No retorno da função, adicionamos ao `render_template` as variáveis que queremos passar para trabalhar dentro do template.

Vamos analisar com mais detalhes o *list comprehension* da função `home` :

```
musicJ = [{ 'fileName': mi.split("/")[-1],  
            "fileURL": url_for('sounds', music=mi)}  
          for mi in musiclist]
```

A variável `mi` conterá o nome das músicas, no formato mp3, que estão na pasta `static/musics/` . Isso foi feito usando o `for mi in musiclist` . Para cada música, criamos um dicionário contendo o nome do arquivo, `'fileName'` , e o endereço de download da música, `"fileURL"` . A variável `mi` contém não só o nome da música, como o caminho completo, com a pasta em que ela se encontra. Para extrair o nome da música, usamos o método `split("/")` para transformar a string `mi` em uma lista, sendo que usamos o termo `"/"` para indicar o separador. Ou seja, se `mi` for igual a `static/musics/meuaudio.mp3` , o método `split` criará a lista `['static', 'musics', 'meuaudio.mp3']` . Usando o índice `-1` , vamos selecionar apenas o último elemento da lista criada, ou seja, apenas o nome da música. Para a URL, usamos a função `url_for` , que receberá o nome da música, e a função receberá o valor de `mi` como parâmetro de requisição.

O código completo ficou da seguinte forma:

```
from flask import Flask, url_for, render_template
```

```

from flask import send_file, request
import glob
import json

app = Flask(__name__)

@app.route("/sounds")
def sounds():
    music = request.args["music"]
    return send_file(music, mimetype="audio/mp3")

@app.route("/")
def home():
    musiclist = glob.glob("static/musics/*.mp3")

    musicJ = [{'fileName': mi.split("/")[-1],
                "fileUrl": url_for('sounds', music=mi)}
               for mi in musiclist]

    return render_template("home.html",
                           musicJ=musicJ,
                           musicJson=json.dumps(musicJ))

if(__name__ == "__main__"):
    app.run()

```

Modificaremos o template `home.html` para receber os dados que foram passados pela função `home` :

```

{% extends "layout.html"%}
{% block content %}

<audio controls>
  <source src="{{ musicJ[0]['fileUrl'] }}" type="audio/mp3">
  Your browser does not support the audio element.
</audio>

{% endblock %}

```


Para acessar o primeiro dicionário da lista `musicJ` usamos o comando `{{ musicJ[0]['fileURL'] }}`, passado como `src` para a tag `source` do nosso controlador de áudio.

O conteúdo de áudio costuma trazer em si mais informações do que somente a música. Esses arquivos carregam informações adicionais, chamadas de metadados, ou descritores dos dados, que trazem detalhes sobre o dado principal. No caso de músicas, podemos ter o nome do autor, intérprete e até mesmo os dados binários da imagem de capa do álbum de que a música faz parte. Na próxima seção, veremos como usar esses dados no nosso projeto.

9.4 MANIPULANDO METADADOS DAS MÚSICAS

Os arquivos de áudio costumam conter informações adicionais, os metadados. Para acessar essas informações, via Python, usaremos um módulo chamado Mutagen (<https://mutagen.readthedocs.io/en/latest/>), que permite acessar os metadados de arquivos multimídia, como `mp3` e `avi`. Assim, conseguiremos acessar as informações das músicas, como nome do artista e álbum. Para instalá-lo, basta usar o comando:

```
pip install mutagen
```

Com essa nova dependência no projeto, vamos modificar a função `home` para obter os metadados das nossas músicas. Porém, vamos criar uma função adicional para converter segundos em minutos, retornando esse dado no formato de string. Isso será importante para exibir tal informação no front-end do nosso sistema de streaming:

```
def sec2minString(sec):
    mi = sec / 60.0
    mi = str(mi).split(".")
    seci = int(float('0.' + mi[1]) * 60.0)
    if(seci < 10):
        seci = '0' + str(seci)
    else:
        seci = str(seci)

    return mi[0] + ":" + seci
```

A função anterior recebe o tempo total da música em segundos. Em seguida, convertemos esse tempo para minutos e separamos a parte decimal. Assim, antes do ponto temos os minutos totais da música, e depois do ponto temos a fração de minutos que deve ser convertida novamente para segundos. Nossa função vai retornar uma string com o formato Minutos:Segundos ('M:S').

Também vamos precisar criar uma função que extraia a capa do álbum, se ele estiver contido como metadado da música. Vamos importar o Mutagen e criar a função que fará o streaming dos arquivos de imagem da capa:

```
from io import BytesIO
from mutagen import File

app = Flask(__name__)

@app.route("/coverImage")
def coverImage():
    cover = request.args["music"]
    cover = File(cover)
    if("APIC:" in cover.tags.keys()):
        imgcover = cover.tags["APIC:"].data
        strIO = BytesIO()
        strIO.write(imgcover)
        strIO.seek(0)
```

```

        return send_file(strIO,
                           mimetype="image/jpg")
    else:
        return app.send_static_file('images/noCoverImage.png')

```

No código anterior, importamos o módulo `BytesIO` do módulo padrão `io`. Ele permite fazer streaming de arquivos para a memória. A função `coverImage` precisa receber como argumento de requisição o caminho do arquivo de áudio a ser processado. Note que, no início do código anterior, importamos a classe `File` do módulo `mutagen`. Assim, dentro da função `coverImage` criamos o objeto `cover` do tipo `File`.

Para verificar se o arquivo de áudio continha a capa do álbum, usamos o seguinte comando: `if("APIC:" in cover.tags.keys())`. Ou seja, estamos verificando se entre as chaves do dicionário de tags do arquivo de áudio existe a `APIC`. Essa tag representa o armazenamento da imagem de capa no arquivo de áudio.

Caso o arquivo contenha a capa, transferimos esse dado para a variável `imgcover` com o comando `cover.tags["APIC:"].data`. Para realizar o streaming, convertemos esses dados em um arquivo a ser manipulado pelo objeto do tipo `strIO`. Note que escrevemos os dados (`strIO.write(imgcover)`), usando `BytesIO`, para serem enviados como arquivo de imagem, através do comando `return send_file(strIO, mimetype="image/jpg")`. Como alternativa, caso a música não contenha o metadado de capa, usamos o comando `return app.send_static_file('images/noCoverImage.png')` para enviar um arquivo estático, que será uma imagem alternativa para a nossa capa.

Agora, vamos modificar a função `home` para acrescentar essas novas informações a serem passadas para o template do nosso projeto:

```
@app.route("/")
def home():
    musiclist = glob.glob("static/musics/*.mp3")

    musicJ = [{"fileName": mi.split("/)[-1],
                "coverURL": url_for('coverImage', music=mi),
                'fileUrl':url_for('sounds', music=mi),
                'length': sec2minString(File(mi).info.length),
                'Tags': None
               } for mi in musiclist]

    for i in range(len(musicJ)):
        tag = File(musiclist[i])
        if('TIT2' in tag.keys()):
            musicJ[i]['Tags'] = {'TIT2':tag['TIT2'].text[0], 'TPE
1':tag['TPE1'].text[0]}

    return render_template("home.html",
                           musicJ=musicJ)
```

Na lista de dicionários `musicJ`, adicionamos a criação de URL para capa, `"coverURL"`, o tempo total da música, `"length"` e deixamos inicialmente a chave `"Tags"` como `None`. Em seguida, verificamos, para cada arquivo de áudio, se existe a tag `TIT2`, que contém dados de título da música. Para mais informações sobre os significados das tags contidas no mp3 veja: <https://en.wikipedia.org/wiki/ID3>.

Com isso, temos o necessário para o back-end da nossa aplicação Web. Segue o código completo do arquivo `fplayer.py`:

```

from flask import Flask, url_for, render_template, json
from flask import send_file, request
import glob

from io import BytesIO
from mutagen import File

app = Flask(__name__)

def sec2minString(sec):
    mi = sec / 60.0
    mi = str(mi).split(".")
    seci = int(float('0.' + mi[1]) * 60.0)
    if(seci < 10):
        seci = '0' + str(seci)
    else:
        seci = str(seci)

    return mi[0] + ":" + seci

@app.route("/sounds")
def sounds():
    music = request.args["music"]
    return send_file(music, mimetype="audio/mp3")

@app.route("/coverImage")
def coverImage():
    cover = request.args["music"]
    cover = File(cover)
    if("APIC:" in cover.tags.keys()):
        imgcover = cover.tags["APIC:"].data
        strIO = BytesIO()
        strIO.write(imgcover)
        strIO.seek(0)

        return send_file(strIO,
                           mimetype="image/jpg")
    else:
        return app.send_static_file('images/noCoverImage.png')

@app.route("/")

```

```

def home():
    musiclist = glob.glob("static/musics/*.mp3")

    musicJ = [{"fileName": mi.split("/")[-1],
               "coverURL": url_for('coverImage', music=mi),
               'fileUrl':url_for('sounds', music=mi),
               'length': sec2minString(File(mi).info.length),
               'Tags': None
               } for mi in musiclist]

    for i in range(len(musicJ)):
        tag = File(musiclist[i])
        if('TIT2' in tag.keys()):
            musicJ[i]['Tags'] = {'TIT2':tag['TIT2'].text[0], 'TPE
1':tag['TPE1'].text[0]}

    return render_template("home.html",
                           musicJ=musicJ)

if(__name__ == "__main__"):
    app.run(debug=True)

```

9.5 FULL STACK: DO BACK-END PARA O FRONT-END

Agora que temos o back-end finalizado, é hora de trabalhar no front-end da aplicação. Nesse projeto em particular, será necessário trabalhar bastante com JavaScript. Para quem não estiver familiarizado com essa linguagem, recomenda-se a leitura do Apêndice 1 deste livro.

Primeiramente, modificaremos a página `home.html` para incluir a customização do nosso player, o qual terá a seguinte aparência:

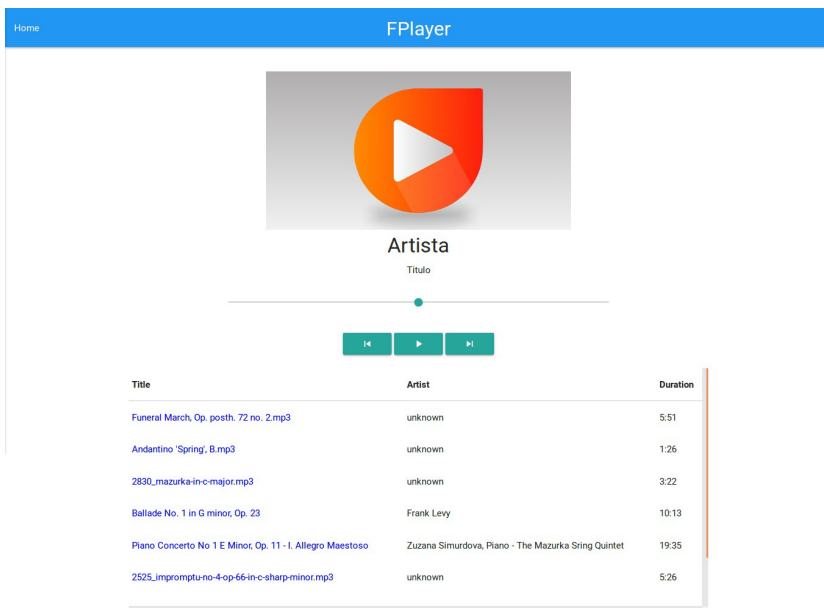


Figura 9.4: Front-end do sistema aplicativo de músicas

Note que no topo da página *home* está o player de áudio customizado, cujo código Jinja/HTML está a seguir:

```
<br>
<audio id="myplay" >
  <source id="audioSource" src=""></source>
  Your browser does not support the audio element.
</audio>

<div class="row">

  <div class="col s12 center-align ">
    

  </div>

  <div clas="col s12 center-align ">
    <h4 id="artista" class="center-align">Artista</h5>
```

```

        <h6 id="titulo" class="center-align">Titulo</h5>
    </div>

    <div class="col s2 center-align ">
        <div id="ctime"></div>
    </div>

    <div class="col s8 center-align ">
        <form action="#">
            <p class="range-field">
                <input id="rangeProgr" type="range" min="0" max="10
0" />
            </p>
        </form>
    </div>
    <div class="col s2 center-align ">
        <div id="totalTime"></div>
    </div>
</div>

```

Primeiramente, adicionamos a tag `audio` sem passar uma fonte de áudio explicitamente. Isso fará com que o browser não apresente o player que vem embutido nele. Ainda assim, poderemos acessar as funções de áudio dessa tag, para isso, vamos adicionar a id `"myplay"` na tag de áudio.

Em seguida, criamos uma estrutura de grade, na qual foi adicionada uma linha, através da marcação `<div class="row">` . Fizemos isso para posicionar a imagem de capa, o nome do artista, o título da música, usando as classes do MaterializeCSS, que permitem criar sites responsivos.

Dentro da tag `<div id="ctime"></div>` , vamos atualizar a informação de tempo da música que está sendo executada. O campo de formulário possui um campo de entrada por intervalo. Essa entrada será usada para controlar o acesso a trechos da música. Logo a seguir, teremos o mostrador de duração da música

(`<div id="totalTime"></div>`). Observe o seguinte comando:

```

```

Aqui estamos passando o primeiro dicionário da lista `musicJ` como fonte da imagem que queremos exibir. Para adicionar ao `html` uma variável que vem do código Python, usando Jinja, fazemos uso do comando `{{ variavel }}` . Neste caso estamos passando a URL da capa da primeira música que está na lista gerada na função `home` .

Em seguida, adicionaremos os botões:

```
<div class="center-align">
  <a class="waves-effect waves-light btn" onmousedown="skip_pre
vious()">
    <i class="small material-icons">skip_previous</i>
  </a>

  <a class="waves-effect waves-light btn " onclick="onPlay()">
    <i id="playbutton" class="small material-icons">play_arrow
</i>
  </a>

  <a class="waves-effect waves-light btn" onmousedown="skip_ne
xt()">
    <i class="small material-icons">skip_next</i>
  </a>

</div>
<br>
```

Observe que deixamos os botões centralizados (`<div class="center-align">`). Para os botões, foi utilizada a classe CSS do MaterializeCSS `waves-effect waves-light btn` . Note

também que estamos usando os ícones do *Material Design*, aquele padrão de design da Google para aplicativos mobile, através das tags `<i class="small material-icons">` .

Para acessar uma música prévia ou uma música posterior, usamos o atributo `onmousedown` , o qual recebe as funções JavaScript `skip_previous()` , para retornar uma música, ou `skip_next()` , para pular para a próxima música. Essas funções serão implementadas mais adiante. Para o botão de play, adicionamos o id `playbutton` . Fizemos isso, pois esse botão funcionará como play/pause. Assim, ao clicar nele e a música já estiver sendo executada, o ícone de play deverá ser substituído pelo ícone de pausa.

Logo abaixo do player, criamos uma tabela com todas as músicas que estão no nosso servidor:

```
1 <div style="height:400px; overflow:scroll;">
2   <table>
3     <thead>
4       <tr>
5         <th>Title</th>
6         <th>Artist</th>
7         <th>Duration</th>
8       </tr>
9     </thead>
10    {% for i in range(musicJ|length) %}
11      <tr>
12        {% if musicJ[i]['Tags'] != None %}
13          <td>
14            <a id="lim_{{i}}" style="color:blue;" onclick="changeMusic('{{ musicJ[i]['fileUrl'] }}')">
15              {{ musicJ[i]['Tags']['TIT2'] }}
16            </a>
17          </td>
18          <td>
19            {{ musicJ[i]['Tags']['TPE1'] }}
20          </td>
```

```

21
22         {% else %}
23         <td>
24             <a id="lim_{{i}}" style="color:blue;" onclick="changeMusic('{{ musicJ[i]['fileUrl'] }}')">
25                 {{ musicJ[i]['fileName'] }}
26             </a>
27         </td>
28         <td>
29             unknown
30         </td>
31         {% endif %}
32         <td>
33             {{ musicJ[i]['length'] }}
34         </td>
35     </tr>
36     {% endfor %}
37 </table>
38 </div>

```

Na linha 1 do código anterior definimos que a `div` contendo a tabela terá uma altura máxima, sendo que se a tabela for mais alta que a `div`, aparecerá a opção de rolagem para ver os outros elementos da tabela.

O Jinja usa elementos diferentes do Python puro. Na linha 10, estamos criando um laço de repetição para pegar cada elemento da lista de dicionários de informações das músicas. Note que, para obter o número total de elementos na lista, usamos o comando `musicJ|length`, que em Python puro seria `len(musicJ)`. Dessa forma, o comando `{% for i in range(musicJ|length) %}` fará um laço de repetição de 0 até o número total de elementos da lista menos um. Esse laço precisa receber a indicação de fechamento, ou seja, onde o bloco de repetição deve terminar. Isso é feito com o comando `{% endfor %}`, que está na linha 36.

Em seguida, verificamos se a chave "Tags" do dicionário que

está sendo analisado não é do tipo `None`, ou seja, se existe algum conteúdo marcado por essa chave. Caso a chave `"Tags"` tenha elementos, preenchemos a tabela com o nome da música, nome do artista, e tempo de duração da música. Observe que o nome da música será do tipo `link`, e que o atributo `onclick` contém a função JavaScript `changeMusic('{{ musicJ[i]['fileUrl'] }}')`. Caso a chave `"Tags"` seja do tipo `None`, usamos apenas o nome do arquivo e, na célula que deveria conter o nome do artista, adicionamos a palavra `unknown` para informar que não temos essa informação.

O código completo da página `home.html` está a seguir:

```
{% extends "layout.html"%}
{% block content %}
<br>
<audio id="myplay" >
    <source id="audioSource" src=""></source>
Your browser does not support the audio element.
</audio>

<div class="row">

    <div class="col s12 center-align ">
        

    </div>

    <div class="col s12 center-align ">
        <h4 id="artista" class="center-align">Artista</h5>
        <h6 id="titulo" class="center-align">Titulo</h5>
    </div>

    <div class="col s2 center-align ">
        <div id="ctime"></div>
    </div>

    <div class="col s8 center-align ">
```

```

        <form action="#">
            <p class="range-field">
                <input id="rangeProgr" type="range" min="0" max="10
0" />
            </p>
        </form>
    </div>
    <div class="col s2 center-align ">
        <div id="totalTime"></div>
    </div>
</div>

<div class="center-align">
    <a class="waves-effect waves-light btn" onmousedown="skip_pre
vious()">
        <i class="small material-icons">skip_previous</i>
    </a>

    <a class="waves-effect waves-light btn " onclick="onPlay()">
        <i id="playbutton" class="small material-icons">play_arro
</i>
    </a>

    <a class="waves-effect waves-light btn" onmousedown="skip_ne
xt()">
        <i class="small material-icons">skip_next</i>
    </a>

</div>
<br>

<div style="height:400px; overflow:scroll;">

    <table>
        <thead>
            <tr>
                <th>Title</th>
                <th>Artist</th>
                <th>Duration</th>
            </tr>
        </thead>
        {% for i in range(musicJ|length) %}
            <tr>
                {% if musicJ[i]['Tags'] != None %}
                    <td>

```

```

        <a id="lim_{{i}}" style="color:blue;" onclick="changeMusic('{{ musicJ[i]['fileName'] }}')">
            {{ musicJ[i]['Tags']['TIT2'] }}
        </a>
    </td>
    <td>
        {{ musicJ[i]['Tags']['TPE1'] }}
    </td>

    {% else %}
        <td>
            <a id="lim_{{i}}" style="color:blue;" onclick="changeMusic('{{ musicJ[i]['fileName'] }}')">
                {{ musicJ[i]['fileName'] }}
            </a>
        </td>
        <td>
            unknown
        </td>

    {% endif %}
</td>
    {{ musicJ[i]['length'] }}
</td>
</tr>

{% endfor %}
</table>
</div>

{% endblock %}

```

Para finalizar o nosso front-end, precisamos criar o código JavaScript para manipulação dos elementos.

9.6 CRIANDO AÇÕES NO FRONT-END COM JAVASCRIPT.

Vamos criar o arquivo `diskPage.js` dentro da pasta `static/js` do projeto. Nesse arquivo estará o código JavaScript

usado para controlar o front-end da nossa aplicação Web.

Vamos começar acrescentando a chamada a esse arquivo na página `home.html` :

```
<script type="text/javascript" src="{{url_for('static', filename='js/diskPage.js')}}"></script>
<script>
    set_ListOfMusics(JSON.parse('{{musicJ|tojson|safe}}'));
    $(document).ready(function(){
        startMusic();
    });
</script>
```

Na primeira linha do código anterior fizemos o link através do comando `src="{{url_for('static', filename='js/diskPage.js')}}"` . Aqui, estamos criando uma URL para o arquivo contendo o código JavaScript. Em seguida, adicionamos JavaScript embebido na página, com a finalidade de passar a lista de músicas, gerada via Python, para o módulo JavaScript, além de fazermos a inicialização do player, logo que a página terminar de carregar. Observe o seguinte comando que foi utilizado antes:

```
set_ListOfMusics(JSON.parse('{{musicJ|tojson|safe}}'));
```

Nesse caso, estamos usando a estrutura do Jinja, com o comando `'{{musicJ|tojson|safe}}'` , para converter a lista de dicionários com as informações de música em uma string do tipo JSON. Essa string é, então, passada para a função JavaScript, `JSON.parse` , que a converte em um conjunto de objetos JSON. Em seguida, usamos o comando `$(document).ready(function(){})` , que é uma estrutura em jQuery, que carregará a função `startMusic()` logo que a página terminar de carregar. Com isso, podemos ver em detalhes a

codificação do `diskPage.js`.

Variáveis de acesso global e inicialização

Muitas funções vão compartilhar algumas variáveis, por isso, elas serão definidas de forma a serem acessadas globalmente. Além disso, vamos precisar passar a lista de músicas para o código JavaScript, o que faremos usando a função `set_ListOfMusics`. Esses elementos estão exemplificados no código a seguir:

```
var current = 0;

var playing = false;

var mysound;
var source;
var progressBar;
var onProgChange = false;
var musics;

function set_ListOfMusics(mlist) {
    musics = mlist;
}
```

Note que criamos a variável global `musics`, que representa o JSON que foi passado à função `set_ListOfMusics`. A variável `current` representa a posição da música que está sendo tocada no momento na lista de músicas. A variável `playing` será usada para indicar se a música está em execução ou não. Já a variável `mysound` vai armazenar o objeto representado pela tag de áudio. O elemento `source` será a fonte de streaming de áudio, ou seja, a música em si, enquanto `progressBar` será a barra de progresso. Finalmente, o `onProgChange` armazenará as mudanças no progresso da música.

Na função `startMusic`, vamos inicializar as outras variáveis

globais:

```
function startMusic() {  
    mysound = document.getElementById('myplay');  
    soundImg = document.getElementById('coverImg');  
    progressBar = document.getElementById("rangeProgr");  
  
    source = document.getElementById('audioSource');  
    source.src = musics[0].fileUrl;  
    soundImg.src = musics[0].coverURL;  
    mysound.load();  
    mysound.play();  
}
```

Note que estamos associando o `myplay`, que representa a tag de áudio, à variável `mysound`, usando o método `document.getElementById`. O mesmo é feito para todas as outras variáveis. A fonte de áudio está associada à variável `source`. Logo após criar o objeto `source`, indicamos que o atributo `src` será o primeiro elemento da lista de música, que deve conter o JSON `fileUrl` (`source.src = musics[0].fileUrl`). Já a imagem fonte, que será usada como representativa da música, é feita com o comando `soundImg.src = musics[0].coverURL;`. O método `mysound.load()` carrega a música no tocador de áudio do Browser, enquanto o comando `mysound.play()` coloca a música para tocar.

O próximo passo é acompanhar a música tocando, para atualizar a barra de progresso e o texto informando o tempo corrente da música. Outra coisa que precisamos verificar é o fim da música. Para isso, vamos escutar os eventos que serão disparados. Usaremos o método `addEventListener` para escutar um evento e disparar um código JavaScript. Dentro da função `startMusic`, vamos adicionar a escuta do evento de fim de música:

```

mysound.addEventListener("ended", function() {
    if (current < musics.length - 1) {
        current += 1;

        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    }
    changeColorMusic();
    changeArtistTitle();

    if(current == musics.length - 1){
        var mylink = document.getElementById("lim_" + current.toS
tring())
        mylink.style.color = "blue";
    }

    updateTime();
});

```

Sempre que o evento `ended` for disparado, o objeto `mysound` é parado, e o código verificará primeiramente se a música em execução não é a última da lista (`if (current < musics.length - 1)`). Nesse caso, indicamos que a próxima música será a corrente (`current += 1`) e é feito o carregamento da música seguinte. Depois chamamos as funções `changeColorMusic` e `changeArtistTitle` que mudarão a cor do texto do título da música, presente na tabela abaixo do player, e o nome do artista que aparece no player. Se a música corrente for a última, o código vai mudar a cor do título da música para azul, que é o padrão de música não sendo executada. A última função vai atualizar o texto com a informação do tempo corrente, indicando qual o tempo atual da música.

Fora da função `startMusic` , vamos criar as funções

`updateTime` e `sec2minString` , sendo esta última similar à que fizemos em Python, para converter segundos em string no formato M:S .

```
function updateTime() {
    document.getElementById("ctime").innerHTML = sec2minString(my
sound.currentTime);
    document.getElementById("totalTime").innerHTML = sec2minStrin
g(mysound.duration);
}

function sec2minString(mytime) {
    var minutes = Number(mytime) / 60;
    var tmp = minutes.toString().split('.');
    minutes = tmp[0];
    seconds = "0." + tmp[1];
    seconds = Math.round(Number(seconds) * 60);
    if (seconds < 10) {
        seconds = '0' + seconds.toString();
    } else {
        seconds = seconds.toString();
    }
    return minutes + ":" + seconds;
}
```

Na função `updateTime` , pegamos os elementos de id `ctime` e `totalTime` . Para a função `sec2minString` , passamos o atributo `currentTime` do objeto `mysound` .

A função `changeColorMusic` será:

```
function changeColorMusic(){
    var mylink;

    for(i=0; i < musics.length; i++){
        mylink = document.getElementById("lim_" + i.toString());
        if(i == current){
            mylink.style.color = "red";
        }else{
            mylink.style.color = "blue";
        }
    }
}
```

```

    }
  }
}

```

Na página `home.html`, cada link recebeu um id do tipo `lim_i`, com `i` variando de 0 ao número total de músicas menos um. Se o valor de `current` for igual ao valor de `i`, então a cor do link com o nome da música ficará com a cor vermelha (`red`), do contrário ficará com a cor azul (`blue`).

Para atualizar as informações sobre o artista e nome da música na página usaremos a seguinte função:

```

function changeArtistTitle(){
  var artist = document.getElementById("artista");
  var titulo = document.getElementById("titulo");
  if(musics[current].Tags != null){
    artist.innerHTML = musics[current].Tags.TPE1;
    titulo.innerHTML = musics[current].Tags.TIT2;
  }else{
    artist.innerHTML = " unknown ";
    titulo.innerHTML = musics[current].fileName;
  }
}

```

A função verifica se o objeto `Tags` da música em execução não é do tipo nulo (`null`). Nesse caso, as informações sobre o nome da música e sobre o artista serão atualizadas. Caso contrário, usaremos o texto `unknown` e colocaremos o nome do arquivo em execução.

Com essas duas funções prontas, vamos adicionar mais um *Event Listener*, que vai atualizar as informações sobre o tempo da música sendo executada (`"timeupdate"`):

```

mysound.addEventListener("timeupdate", function() {
  var currentTime = mysound.currentTime;
  var duration = mysound.duration;

```

```

    var per = (currentTime / duration) * 100;
    if (onProgChange == false) {
        progressBar.value = per;
        updateTime();
    }

});

```

Essa função atualiza a barra de progresso. A variável `per` tem a porcentagem da execução da música. Por fim, precisamos atualizar a música quando o usuário arrastar a barra de progresso. Para isso, adicionaremos mais alguns *Event Listeners* à função `startMusic` :

```

progressBar.addEventListener('mouseup', function() {
    var newTime = this.value * mysound.duration / 100;
    mysound.currentTime = newTime;
    onProgChange = false;
    updateTime();
});

```

No código anterior, ao soltar o botão mouse (`"mouseup"`), pegamos o valor da barra de progresso e convertemos para tempo. Em seguida, atualizamos o tempo corrente para o som.

Ao pressionar o mouse, vamos indicar que estamos fazendo mudança no progresso da música de forma manual:

```

progressBar.addEventListener('mousedown', function() {
    onProgChange = true;
});

```

Para manter compatibilidade com touchscreen, como as telas de smartphones, usamos o evento `touchend` para atualizar a música ao finalizar o toque da tela:

```

progressBar.addEventListener('touchend', function() {
    var newTime = this.value * mysound.duration / 100;
    mysound.currentTime = newTime;
    onProgChange = false;
    updateTime();
});

```

Da mesma forma, indicamos que iniciamos o movimento da barra de progresso ao iniciar o toque na tela, com o seguinte código:

```

progressBar.addEventListener('touchstart', function() {
    onProgChange = true;
});

```

O código completo da função é apresentado a seguir:

```

function startMusic() {
    mysound = document.getElementById('myplay');
    soundImg = document.getElementById('coverImg');
    progressBar = document.getElementById("rangeProgr");

    source = document.getElementById('audioSource');
    source.src = musics[0].fileUrl;
    soundImg.src = musics[0].coverURL;
    mysound.load();
    mysound.play();
    changeColorMusic();
    changeArtistTitle();
    playing = true;
    mysound.addEventListener("ended", function() {
        if (current < musics.length - 1) {
            current += 1;

            source.src = musics[current].fileUrl;
            document.getElementById('coverImg').src = musics[curr
ent].coverURL;
            mysound.load();
            mysound.play();
        }
    });
}

```

```

        changeColorMusic();
        changeArtistTitle();

        if(current == musics.length - 1){
            var mylink = document.getElementById("lim_" + current
.toString())
            mylink.style.color = "blue";
        }

        updateTime();

    });

    mysound.addEventListener("timeupdate", function() {
        var currentTime = mysound.currentTime;
        var duration = mysound.duration;
        var per = (currentTime / duration) * 100;
        if (onProgChange == false) {
            progressBar.value = per;
            updateTime();
        }
    });

    progressBar.addEventListener('mouseup', function() {
        var newTime = this.value * mysound.duration / 100;
        mysound.currentTime = newTime;
        onProgChange = false;
        updateTime();
    });

    progressBar.addEventListener('mousedown', function() {
        onProgChange = true;
    });

    progressBar.addEventListener('touchend', function() {
        var newTime = this.value * mysound.duration / 100;
        mysound.currentTime = newTime;
        onProgChange = false;
        updateTime();
    });

```

```

progressBar.addEventListener('touchstart', function() {
    onProgChange = true;

});

}

```

A próxima função a ser criada é a de `onPlay` :

```

function onPlay() {
    if (playing == false) {
        document.getElementById('myplay').play();
        playing = true;
        document.getElementById("playbutton").innerHTML = 'play_arrow';

    } else {
        document.getElementById('myplay').pause();
        playing = false;
        document.getElementById("playbutton").innerHTML = 'pause'
    }
    updateTime();
}

```

Na função anterior, verifica-se se a música está em execução. Se não estiver, colocamos a música em execução e mudamos o ícone do botão de play para `play_arrow` , do contrário, paramos a música e trocamos o ícone para o tipo `pause` .

A função `changeMusic` será executada quando o link com o nome da música for clicado:

```

function changeMusic(newSound) {

    current = musics.map(function(d) {
        return d['fileUrl'];
    }).indexOf(newSound);
    changeColorMusic();
}

```



```

changeArtistTitle();

source.src = newSound;
document.getElementById('coverImg').src = musics[current].coverURL;
mysound.load();
mysound.play();
updateTime();
}

```

Para poder atualizar a variável global `current`, com o índice correspondente à música clicada, usamos a função JavaScript `map`:

```

current = musics.map(function(d) {
    return d['fileUrl'];
}).indexOf(newSound);

```

Essa função recebe como argumento uma outra função, a qual atuará em cada um dos elementos do Array original, em ordem, construindo um novo Array com base nos retornos de cada chamada. Dessa forma, teremos um outro Array contendo apenas os nomes dos arquivos de áudio.

Por fim, aplicamos o método `indexOf` para retornar o índice do Array que representará a posição da música na lista de músicas, atualizando a variável `current`. O outro código contido nessa função serve para atualizar o arquivo de áudio a ser tocado, juntamente com as informações da capa do álbum do qual a música faz parte.

Para pular para a próxima música, simplesmente verificamos o valor da variável `current`. Se o valor contido em `current` for menor que o total de músicas na lista menos um, então atualizam-se as informações com os dados da próxima música e adicionamos

mais 1 ao valor da variável `current` (`current += 1;`). Caso contrário, colocamos a primeira música novamente. O código dessa função é apresentado a seguir:

```
function skip_next() {
    if (current == musics.length - 1) {
        current = 0;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    } else {
        current += 1;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    }
    changeColorMusic();
    changeArtistTitle();
    updateTime();
}
```

Para pular para a música anterior usamos o mesmo princípio para a música seguinte; a diferença é que vamos subtrair 1 do valor da variável `current`, se a música atual não for a primeira:

```
function skip_previous() {
    if (current == 0) {
        current = 0;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    } else {
        current -= 1;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
    }
```

```

        mysound.load();
        mysound.play();
    }
    changeColorMusic();
    changeArtistTitle();
    updateTime();
}

```

O código completo do script `diskPage.js` será:

```

var current = 0;

var playing = false;

var mysound;
var source;
var progressBar;
var onProgChange = false;
var musics;

function set_ListOfMusics(mlist) {
    musics = mlist;
}

function startMusic() {
    mysound = document.getElementById('myplay');
    soundImg = document.getElementById('coverImg');
    progressBar = document.getElementById("rangeProgr");

    source = document.getElementById('audioSource');
    source.src = musics[0].fileUrl;
    soundImg.src = musics[0].coverURL;
    mysound.load();
    mysound.play();
    changeColorMusic();
    changeArtistTitle();
    playing = true;
    mysound.addEventListener("ended", function() {
        if (current < musics.length - 1) {
            current += 1;

            source.src = musics[current].fileUrl;
            document.getElementById('coverImg').src = musics[curr
ent].coverURL;

```

```

        mysound.load();
        mysound.play();
    }
    changeColorMusic();
    changeArtistTitle();

    if(current == musics.length - 1){
        var mylink = document.getElementById("lim_" + current
.toString())
        mylink.style.color = "blue";
    }

    updateTime();

});

mysound.addEventListener("timeupdate", function() {
    var currentTime = mysound.currentTime;
    var duration = mysound.duration;
    var per = (currentTime / duration) * 100;
    if (onProgChange == false) {
        progressBar.value = per;
        updateTime();
    }
});

progressBar.addEventListener('mouseup', function() {
    var newTime = this.value * mysound.duration / 100;
    mysound.currentTime = newTime;
    onProgChange = false;
    updateTime();
});

progressBar.addEventListener('mousedown', function() {
    onProgChange = true;
});

progressBar.addEventListener('touchend', function() {
    var newTime = this.value * mysound.duration / 100;
    mysound.currentTime = newTime;
    onProgChange = false;
});

```

```

        updateTime();

    });

    progressBar.addEventListener('touchstart', function() {
        onProgChange = true;
    });

}

function changeMusic(newSound) {

    current = musics.map(function(d) {
        return d['fileUrl'];
    }).indexOf(newSound);
    changeColorMusic();
    changeArtistTitle();

    source.src = newSound;
    document.getElementById('coverImg').src = musics[current].coverURL;
    mysound.load();
    mysound.play();
    updateTime();

}

function changeArtistTitle(){
    var artist = document.getElementById("artista");
    var titulo = document.getElementById("titulo");
    if(musics[current].Tags != null){
        artist.innerHTML = musics[current].Tags.TPE1;
        titulo.innerHTML = musics[current].Tags.TIT2;
    }else{
        artist.innerHTML = " unknown ";
        titulo.innerHTML = musics[current].fileName;
    }
}

function changeColorMusic(){

```

```

var mylink;

for(i=0; i < musics.length; i++){
    mylink = document.getElementById("lim_" + i.toString());
    if(i == current){
        mylink.style.color = "red";
    }else{
        mylink.style.color = "blue";
    }
}
}

function onPlay() {
    if (playing == false) {
        document.getElementById('myplay').play();
        playing = true;
        document.getElementById("playbutton").innerHTML = 'play_a
row';

    } else {
        document.getElementById('myplay').pause();
        playing = false;
        document.getElementById("playbutton").innerHTML = 'pause'
;
    }
    updateTime();
}

function skip_next() {
    if (current == musics.length - 1) {
        current = 0;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    } else {
        current += 1;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    }
}

```

```

    }
    changeColorMusic();
    changeArtistTitle();
    updateTime();
}

function skip_previous() {
    if (current == 0) {
        current = 0;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    } else {
        current -= 1;
        source.src = musics[current].fileUrl;
        document.getElementById('coverImg').src = musics[current]
.coverURL;
        mysound.load();
        mysound.play();
    }
    changeColorMusic();
    changeArtistTitle();
    updateTime();
}

function updateTime() {
    document.getElementById("ctime").innerHTML = sec2minString(my
sound.currentTime);
    document.getElementById("totalTime").innerHTML = sec2minStrin
g(mysound.duration);
}

function sec2minString(mytime) {
    var minutes = Number(mytime) / 60;
    var tmp = minutes.toString().split('.');
    minutes = tmp[0];
    seconds = "0." + tmp[1];
    seconds = Math.round(Number(seconds) * 60);
    if (seconds < 10) {
        seconds = '0' + seconds.toString();
    } else {

```

```
        seconds = seconds.toString();  
    }  
    return minutes + ":" + seconds;  
  
}
```

Com esse código finalizado, o player de áudio está completamente funcional.

Com isso, finalizamos a parte de front-end e passamos a ter um aplicativo Web completo. Veja que foi necessário bastante codificação para o front-end, talvez até mais que para o back-end, em particular para esse projeto. Assim, vemos que para se tornar um programador full stack é fundamental dominar não só a linguagem de back-end, que para esse livro estamos usando Python, como JavaScript para o front-end. Pensando nisso, o Apêndice 1 trará uma abordagem mais profunda sobre os fundamentos de JavaScript.

9.7 CONCLUSÃO

Neste capítulo, foi apresentado o desenvolvimento de uma aplicação Web usando Python, Flask, HTML5, JavaScript e CSS. Vimos que a codificação do front-end, usando JavaScript pode ser tão rica e complexa quanto o desenvolvimento back-end. Na primeira parte do capítulo, criamos a estrutura de back-end, usando Flask. Aqui utilizamos os módulos extra, chamado *Mutagen*, para obter as informações de metadados de uma dada música, e também vimos como enviar arquivos do servidor para o cliente. Em seguida, construímos a estrutura do front-end, baseado no MaterializeCSS. Para finalizar, criamos toda a parte funcional da página, usando JavaScript.

No próximo capítulo vamos falar sobre persistência de dados em banco de dados.

PERSISTÊNCIA DE DADOS

Em muitos casos, principalmente quando um sistema precisa trabalhar com muitos usuários, precisamos guardar informações. Assim, quando o sistema é fechado, ou interrompido por algum motivo, devemos acessar as informações de alguma forma. Uma das maneiras mais práticas de organizar e armazenar dados é através do uso de banco de dados.

Os bancos de dados podem ser do tipo relacional, no qual tabelas de dados são construídas com base em relações bem definidas, e do tipo não relacional. Nos bancos não relacionais, chamados de NoSQL (*Not Only SQL* - Não somente SQL), não exigem esquemas de tabelas fixas, permitindo o armazenamento de dados não estruturados, ou semiestruturados, como arquivos de logs e arquivos multimídia.

A opção por um banco de dados ou outro acaba dependendo mais do perfil e do volume de dados que seu sistema irá tratar. Se os dados guardam uma relação estrita e bem definida entre si, como no caso de um sistema de vendas para um pequeno mercado, podemos usar um banco relacional sem grandes dificuldades. Agora, se tivermos que criar uma rede social, no qual imagens, vídeos, textos e interação entre usuários, em que se tem um grande volume de pessoas interagindo com o sistema,

recomenda-se o uso de banco de dados NoSQL.

Um dos bancos de dados NoSQL mais conhecidos é o MongoDB; já entre os bancos de dados relacionais podemos citar o PostgreSQL, MariaDB, MySQL, Oracle e SQL Server. O grande problema aqui é que cada um tem uma linguagem SQL própria e, embora elas carreguem elementos em comum, não são exatamente iguais. O próprio NoSQL tem uma linguagem que vai bem além do SQL tradicional. Para quem está desenvolvendo sistemas e quer manter a compatibilidade com os mais diversos bancos, o melhor caminho é escolher um sistema de Mapeamento de Objetos Relacionais (*Object-Relational Mapper* - ORM) ou uma Camada de Abstração de Banco de Dados (*Database Abstraction Layer* - DAL). Os ORMs procuram estabelecer uma ponte entre o paradigma de Programação Orientada a Objetos com o modelo relacional dos bancos de dados. Por outro lado, o DAL acaba focando em uma estratégia que mescla Programação Funcional e Orientação a Objetos, apresentando um modelo mais abstrato de se realizar pesquisas em banco de dados.

Neste capítulo, será apresentado como fazer persistência em banco de dados em aplicações Flask, usando DAL.

10.1 CONEXÃO COM O BANCO DE DADOS

O pyDAL é uma camada de abstração de banco de dados que foi desenvolvida como componente do framework Web2py, que mais tarde foi liberado como um pacote independente do framework.

Quando o browser envia uma requisição para o servidor, a

aplicação precisa saber qual código executar. Vimos no capítulo anterior, que esse mapeamento entre a URL digitada pelo usuário, no browser, e a função a ser executada é feita pelo decorador `@app.route`. Cada função chamada será executada em sua linha de processamento específica, ou sua *thread*. Essas funções, que vão gerenciar a geração das páginas do site, são chamadas de funções de *view*. Ao utilizar o pyDAL, a cada chamada de uma função de *view*, precisamos estabelecer a conexão com o banco de dados. Isso será feito pela criação de uma função que retornará um objeto do tipo DAL.

Para podermos acessar os dados, precisamos criar conexões com o banco de dados. Em uma aplicação Web, o servidor fica à espera de uma requisição externa do usuário para só então executar uma tarefa. Em Flask, cada função de *view* será executada na sua própria linha de processamento, as chamadas *threads*. Dessa forma, podemos ter vários usuários distintos chamando uma mesma função. Cada vez que fizermos uma requisição ao sistema, teremos que fazer uma conexão com o banco de dados.

Para estabelecer conexão com o banco de dados por requisição, usando o pyDAL com o Flask, basta encapsular a criação do objeto tipo DAL em uma função, sendo que essa função deverá retornar esse objeto, que representará a conexão com o banco de dados.

Primeiramente, vamos criar um módulo chamado `model.py`, no qual criaremos as funções para: i) conexão com o banco de dados, chamada `model`; ii) definição das tabelas de que vamos necessitar para o projeto, chamada `table`.

No código a seguir temos a inicialização do módulo `model.py` juntamente com a função `model`:

```

#!/usr/bin/env python
# -*- Coding: UTF-8 -*-

import datetime
from pydal import DAL, Field

def model():
    dbinfo = 'sqlite://storage.sqlite'
    folder = "./database"

    db = DAL(dbinfo, folder=folder, pool_size=1)
    table(db)
    return db

```

Na função `model`, a variável `dbinfo` vai armazenar as informações sobre o tipo de banco de dados e informações sobre conexão, como nome de usuário, senha e banco de dados. A variável `folder` armazena o nome da pasta na qual ficam as informações sobre a manipulação do banco de dados e, no caso de se utilizar o SQLite, o local em que ficará armazenado o próprio banco de dados.

Em seguida, criamos o objeto `db=DAL(dbinfo, folder=folder, pool_size=1)`. O argumento `pool_size` permite a reutilização de uma conexão com o banco de dados. Por padrão, o valor de `pool_size` é zero, ou seja, a cada nova requisição, uma nova conexão com o banco de dados é criada. Para o caso de `pool_size` diferente de zero, estamos colocando as conexões do banco de dados em uma piscina (*pool*), de conexões concorrentes. As conexões em uma *pool* são compartilhadas sequencialmente entre linhas de processamento (threads), ou seja, poderão ser reutilizadas em uma próxima conexão, mas nunca são utilizadas simultaneamente por duas threads. No caso do Flask, quando um mesmo usuário chamar uma função de view novamente, poderemos reutilizar a mesma conexão com o banco

de dados.

O pyDAL permite conexão com uma série de banco de dados. Na tabela a seguir, apresentamos os diversos bancos de dados que podemos utilizar e como deve ser o formato da string, apresentada pela variável `dbinfo`, para fazer a conexão com o devido banco:

| Banco de Dados | Conexão |
|----------------|--|
| SQLite | sqlite://storage.sqlite |
| MySQL | mysql://username:password@localhost/test |
| PostgreSQL | postgres://username:password@localhost/test |
| MSSQL (legacy) | mssql://username:password@localhost/test |
| MSSQL (>=2005) | mssql3://username:password@localhost/test |
| MSSQL (>=2012) | mssql4://username:password@localhost/test |
| FireBird | firebird://username:password@localhost/test |
| Oracle | oracle://username/password@test |
| DB2 | db2://username:password@test |
| Ingres | ingres://username:password@localhost/test |
| Sybase | sybase://username:password@localhost/test |
| Informix | informix://username:password@test |
| Teradata | teradata://DSN=dsn;UID=user;PWD=pass;DATABASE=test |
| Cubrid | cubrid://username:password@localhost/test |
| SAPDB | sapdb://username:password@localhost/test |
| IMAP | imap://user:password@server:port |
| MongoDB | mongodb://username:password@localhost/test |
| Google/SQL | google.sql://project:instance/database |
| Google/NoSQL | google.datastore |
| | |

Assim, se quisermos usar o MySQL, por exemplo, basta trocar o conteúdo da variável `dbinfo` pelas informações do banco. Considere que o usuário do banco de dados seja `root`, com senha `password`, também assumindo que já tenha sido criado previamente no MySQL o banco de dados chamado `fplayer`. Se o nosso banco de dados estiver rodando na mesma máquina em que está a aplicação Flask, o endereço será `localhost`. Vamos assumir que o nosso banco de dados não esteja na mesma máquina da aplicação, mas sim, por exemplo, em uma máquina com endereço de IP `192.168.0.105` e na porta `3306`. Nesse caso, a variável `dbinfo` será:

```
dbinfo = 'mysql://root:password@192.168.0.105:3606/fplayer'
```

Na próxima seção será apresentada a criação das tabelas do banco de dados.

10.2 CRIANDO OS MODELOS DE TABELAS

Na função `model`, estamos chamando a função `table`, que vai receber um objeto do tipo `DAL` e é onde vamos definir as tabelas do nosso banco de dados. Vamos criar uma tabela para guardar as informações da música, outra para guardar informações de músicas preferidas e outra para guardar informações sobre o tempo de execução corrente de uma música. Com isso, se nosso servidor de streaming estiver nas nuvens, poderemos continuar a escutar uma música de onde paramos, usando outro aparelho. Também criaremos uma tabela chamada `genero`, para armazenar os possíveis gêneros musicais.

A tabela `musica` terá um campo `genero` que armazenará uma lista de referências para a tabela `genero`. Com isso, estamos indicando que uma mesma música poderá ser classificada com mais de um gênero musical. Outra tabela importante é a que conterá as informações do usuário, como nome, e-mail e senha.

Outra informação que podemos armazenar é a última vez que uma música foi tocada. Essa tabela terá o campo `tocadaem` para armazenar a data e a hora em que a música foi executada pela última vez. A vantagem de acrescentar essa tabela está em poder armazenar informações sobre o que a pessoa está escutando e o que ele mais gosta, auxiliando na criação de um sistema de recomendação, baseado nas mudanças de interesse do usuário com o tempo.

O objeto `db` possui o método `define_table`, que é usado para criar a tabela no banco de dados. A classe `Field` será usada para gerar os campos na tabela. Essa função recebe primeiramente o nome do campo, em seguida, uma string que representa o tipo de campo, `'string'` para campo do tipo *text*, `'double'` para campo de ponto flutuante, `'integer'` para números inteiros, `'date'`, para campos do tipo data, `'datetime'` para campos de data acrescidos de informações de horário.

Assim, a tabela com as informações usadas para acesso do usuário ao sistema será:

```
db.define_table("user",
    Field("name", 'string'),
    Field("email", 'string'),
    Field('password', 'password')
)
```

A função `table` com todas as outras tabelas necessárias para

o projeto é mostrada a no seguinte código:

```
def table(db):

    db.define_table("user",
        Field("name", 'string'),
        Field("email", 'string'),
        Field('password', 'password')
    )

    db.define_table("genero",
        Field("nome", 'string')
    )

    db.define_table("musica",
        Field("nome", 'string'),
        Field('cantor', 'string'),
        Field('album', 'string'),
        Field('arquivo', 'string'),
        Field('tempo', 'string'),
        Field('genero', 'list:reference estilos'),
    )

    db.define_table("preferidas",
        Field("musica", 'reference musica'),
        Field("user", 'reference user')
    )

    db.define_table("posicao",
        Field("segundos", 'double'),
        Field("musica", "reference musica"),
        Field("user", 'reference user')
    )

    db.define_table("tocada",
        Field("tocadaem", 'datetime',
            default=datetime.datetime.now()
        ),
        Field("musica", "reference musica"),
        Field("user", 'reference user')
    )
```

A classe `Field` também pode receber um valor padrão

(*default*), assim, sempre que uma nova linha na tabela for criada, o campo será preenchido com algum valor. Por exemplo, supomos que queremos guardar a informação de horário em que uma compra foi realizada. Nesse caso, criamos o campo `tocadaem`, tal que temos um campo construído da seguinte forma:

```
Field("tocadaem", 'datetime',  
      default=datetime.datetime.now()  
      )
```

Aqui, estamos usando o módulo padrão `datetime` para recuperar a hora atual do sistema. Usando um valor `default`, o campo será preenchido com algo, mesmo que não tenhamos explicitado o campo que contém o valor padrão na função em que vamos escrever dados na tabela.

Ao montar as tabelas do nosso banco de dados, pôde-se ver que o campo da tabela pode ser do tipo referência, o qual vai armazenar uma referência a outra tabela. Usamos isso para referenciar tanto a tabela `música`, como no campo `Field("musica", "reference musica")`, quanto a tabela `usuário`, ao criar o campo `Field("user", 'reference user')`. Também podemos referenciar mais de um item de uma dada tabela, ao criarmos uma lista de referência, tal como foi feito para definir o gênero da música, `Field('genero', 'list:reference estilos')`. O ideal é sempre trabalhar com tabelas menores, o que ajuda a manter o banco de dados organizados e a agilizar a busca no banco.

O código completo do módulo `model.py` é apresentado a seguir:

```
#!/usr/bin/env python  
# -*- Coding: UTF-8 -*-
```

```

import datetime
from pydal import DAL, Field

def model():
    dbinfo = 'sqlite://storage.sqlite'
    folder = "./database"

    db = DAL(dbinfo, folder=folder, pool_size=1)
    table(db)
    return db

def table(db):

    db.define_table("user",
        Field("name", 'string'),
        Field("email", 'string'),
        Field('password', 'password')
    )

    db.define_table("genero",
        Field("nome", 'string')
    )

    db.define_table("musica",
        Field("nome", 'string'),
        Field('cantor', 'string'),
        Field('album', 'string'),
        Field('arquivo', 'string'),
        Field('tempo', 'string'),
        Field('genero', 'list:reference estilos'),
    )

    db.define_table("preferidas",
        Field("musica", 'reference musica'),
        Field("user", 'reference user')
    )

    db.define_table("posicao",
        Field("segundos", 'double'),
        Field("musica", "reference musica"),
        Field("user", 'reference user')
    )

    db.define_table("tocada",

```

```
Field("tocadaem", 'datetime',
      default=datetime.datetime.now()
    ),
Field("musica", "reference musica"),
Field("user", 'reference user')
)
```

Na próxima seção, veremos como integrar o banco de dados com a aplicação Flask.

10.3 INTEGRAÇÃO ENTRE APLICAÇÃO E BANCO DE DADOS

Primeiramente, vamos adicionar as informações das músicas no banco de dados. No programa principal, vamos criar uma constante chamada `MUSICFOLDER` que vai armazenar o endereço da pasta em que salvaremos as músicas. A primeira coisa a fazer é criar uma função que fará a atualização do banco de dados:

```
MUSICFOLDER = 'static/musics/'

def updatemusic():
    db = model()
    musiclist = glob.glob(MUSICFOLDER + "*.mp3")
    musicnames = [mi.split("/")[-1] for mi in musiclist]

    indb = [msi.arquivo
             for msi in db().iterselect(db.musica.arquivo)
             if msi.arquivo in musicnames
            ]
    notindb = list(set(musicnames) - set(indb))
```

Nessa função, iniciamos com a criação da conexão com o banco de dados usando o comando `db = model()`. Em seguida, criamos uma lista com as músicas, no formato mp3, que estão no diretório de armazenamento de música. O próximo passo foi a criação de uma lista contendo apenas os nomes dos arquivos de

áudio, sem o caminho do diretório, `musicnames = [mi.split("/")[-1] for mi in musiclist]` . O comando `mi.split("/")` converte a string em uma lista de strings, na qual foi usado o `"/"` como item separador dos elementos da lista.

Assim, pegamos o último elemento dessa nova lista, usando índice `-1` , para guardar apenas o nome dos arquivos, sem o caminho do diretório. Em seguida, queremos verificar se alguma música nova foi adicionada a esse diretório, para fazer isso, comparamos a lista com os nomes dos arquivos de áudio com o que está armazenado no banco de dados. Para coletar os nomes dos arquivos de áudio que estão no banco de dados e ao mesmo tempo no diretório, usamos um `listcomprehension` :

```
indb = [msi.arquivo
        for msi in db().iterselect(db.musica.arquivo)
        if msi.arquivo in musicnames
    ]
```

O comando `db().iterselect(db.musica.arquivo)` está selecionando somente o conteúdo da tabela `musica` e coluna `arquivo` .

Para selecionar elementos em tabelas, usando o `pyDAL`, podemos usar os métodos `iterselect` ou `select` . Ao se utilizar o `iterselect` , o retorno de dados da tabela é realizado um de cada vez, sem sobrecarga de memória. Porém, ao se utilizar o `select` , estamos armazenando todos os dados da nossa busca na memória do computador, o que acaba gerando um custo computacional maior. Tanto o `iterselect` quanto o `select` recebem como parâmetro a tabela e a coluna que desejamos selecionar. Nesse caso, nós queremos apenas as colunas `arquivo` da tabela `musica` que está no banco de dados `db` .

O próximo passo foi o de comparar o conteúdo que está no banco de dados com o que não está. Para isso, usamos a função `set`, que converte uma lista em um conjunto. O comando `set(musicnames) - set(indb)` retorna os elementos que estão em `musicnames`, mas que não estão em `indb`.

Agora podemos saber se alguma música nova foi adicionada na pasta de arquivos de áudio. O próximo passo será o de atualizar as informações no banco de dados. Vamos adicionar isso à função `updatemusic`:

```
for msi in notindb:
    tag = File(MUSICFOLDER + msi)
    tempo = sec2minString(File(MUSICFOLDER + msi).info.length))

    if 'TIT2' in tag.keys():
        db.musica.insert(nome=tag['TIT2'].text[0],
                        cantor=tag['TPE1'].text[0],
                        arquivo=msi,
                        tempo=tempo
                        )
    else:
        db.musica.insert(arquivo=msi,
                        tempo=tempo
                        )
```

Aqui estamos fazendo uma iteração sobre a lista de músicas que não estão no banco de dados. Em seguida, verificamos se o arquivo contém os metadados sobre o nome da música e informação do intérprete. Se tiver, inserimos essa informação no banco de dados. Para isso, usamos o método `insert` que pertence ao objeto tabela, `db.musica.insert`. O `insert` recebe como parâmetro o nome dos campos da tabela, com os seus respectivos valores. Se utilizarmos este método para inserir dados na tabela, mas não informarmos um campo explicitamente, ou o campo ficará com valor `None` ou será adicionado o valor

default , que foi definido no momento em que criamos as tabelas no nosso arquivo `model.py` .

Outro ponto importante é o de verificar se alguma música que está no banco de dados foi removida da pasta de arquivos de áudios:

```
notindir = [msi.arquivo
             for msi in db().iterselect(db.musica.arquivo)
             if msi.arquivo not in musicnames
            ]

for msi in notindir:
    db(db.musica.arquivo == msi).delete()

db.commit()
```

Nesse caso, criamos um `listcomprehension` para verificar se uma música está no banco de dados, mas que não está no diretório. Para poder apagar um dado, precisamos fazer uma pesquisa no banco, ou *query*, para especificar exatamente o que queremos apagar. O objeto `db` é do tipo que permite ser chamado, ou *callable* , ou seja, ao mesmo tempo ele se comporta como objeto e como função. O objeto `db` recebe como parâmetro a condição de pesquisa. Por exemplo, se quisermos deletar apenas a música com o `id` igual a 1, fazemos:

```
db(db.musica.id == 1).delete()
```

Podemos fazer combinações de *query*, para deixar a seleção mais específica:

```
query = db((db.musica.id > 3) &
           (db.preferidas.musica == db.musica.id) &
           (db.preferidas.user.id == 1)
          )
query.delete()
```

Nesse caso, estamos escolhendo as músicas com id maior que três, que estão na coluna `musica` da tabela `preferidas` e que estão associadas ao usuário de id igual a 1.

O comando `db.commit()` fará a gravação efetiva do banco de dados. Sem a adição desse comando, os dados não serão alterados no banco de dados. Mas a gravação das informações do banco consome tempo, logo, em vez de fazer uma atualização a cada modificação de dado, dentro de um laço de repetição, deixamos essa operação como a última a ser realizada dentro da função. Dessa forma, gravamos todas as modificações de uma só vez, o que nos faz economizar processamento de máquina, deixando a aplicação otimizada.

O código completo da função `updatemusic` é apresentado a seguir:

```
MUSICFOLDER = 'static/musics/'

def updatemusic():
    db = model()
    musiclist = glob.glob(MUSICFOLDER + "*.mp3")
    musicnames = [mi.split("/")[-1] for mi in musiclist]

    indb = [msi.arquivo
             for msi in db().iterselect(db.musica.arquivo)
             if msi.arquivo in musicnames
            ]

    notindb = list(set(musicnames) - set(indb))

    for msi in notindb:
        tag = File(MUSICFOLDER + msi)
        tempo = sec2minString(File(MUSICFOLDER + msi).info.length
        )

        if('TIT2' in tag.keys()):
            db.musica.insert(nome=tag['TIT2'].text[0],
```



```

        cantor=tag['TPE1'].text[0],
        arquivo=msi,
        tempo=tempo
    )
else:
    db.musica.insert(arquivo=msi,
                    tempo=tempo
                    )

notindir = [msi.arquivo
            for msi in db().iterselect(db.musica.arquivo)
            if msi.arquivo not in musicnames
            ]

for msi in notindir:
    db(db.musica.arquivo == msi).delete()

db.commit()

```

Vamos modificar a função `home` e criar uma outra função que vai acessar o banco de dados para construir a lista de dicionários `musicJ`, que é passado para a view.

Vamos criar a função `get_musics`. A primeira coisa que devemos fazer é a conexão com o banco de dados. Em seguida, vamos selecionar as informações da tabela

```

def get_musics():
    db = model()
    musiclist = db().select(db.musica.arquivo,
                            db.musica.tempo,
                            db.musica.cantor,
                            db.musica.nome,
                            orderby=db.musica.arquivo|db.musica.n
ome
    )

```

Ao final do `select`, passamos o parâmetro `orderby=db.musica.arquivo|db.musica.nome`. Nesse caso, o que estamos fazendo é pedindo para ordenar a seleção de acordo

com o nome do arquivo, ou, usando o `|`, de acordo com o nome da música. Para inverter a ordem da seleção, usamos o operador de negação `~`. Assim, se quisermos que a ordem da seleção seja invertida, fazemos `orderby=~db.musica.arquivo`.

O próximo passo será o de verificar se existe alguma música registrada no banco de dados. Para fazer isso, verificamos se o total de elementos contidos em `musiclist` é maior que zero. Caso isso seja verdade, será criada a lista `musicJ`:

```
if len(musiclist) > 0:

    musicJ = [{"fileName": mi.arquivo,
               "coverURL": url_for('coverImage',
                                   music=MUSICFOLDER + mi.arq
uivo
                                   ),
               'fileUrl': url_for('sounds',
                                   music=MUSICFOLDER + mi.arqu
ivo
                                   ),
               'length': mi.tempo,
               'Tags': None
               }
               for mi in musiclist
               ]

    for i in range(len(musicJ)):
        if musiclist[i].cantor is not None:
            musicJ[i]['Tags'] = {
                'TIT2': musiclist[i].nome,
                'TPE1': musiclist[i].cantor
            }
        else:
            musicJ = []
    return musicJ
```

A função completa é apresentada a seguir:

```
def get_musics():
    db = model()
```

```

musiclist = db().select(db.musica.arquivo,
                        db.musica.tempo,
                        db.musica.cantor,
                        db.musica.nome,
                        orderby=-db.musica.arquivo|db.musica.
nome
                        )
    if len(musiclist) > 0:

        musicJ = [{"fileName": mi.arquivo,
                    "coverURL": url_for('coverImage',
uivo
                                music=MUSICFOLDER + mi.arq

                                ),
                    'fileUrl': url_for('sounds',
ivo
                                music=MUSICFOLDER + mi.arqu

                                ),
                    'length': mi.tempo,
                    'Tags': None
                    }
                    for mi in musiclist
                    ]

        for i in range(len(musicJ)):
            if musiclist[i].cantor is not None:
                musicJ[i]['Tags'] = {
                    'TIT2': musiclist[i].nome,
                    'TPE1': musiclist[i].cantor
                }
            else:
                musicJ = []
        return musicJ

```

Finalmente, modificamos a função `home` para que o banco de dados seja atualizado:

```

@app.route("/")
def home():
    updatemusic()
    musicJ = get_musics()

    return render_template("home.html",
                           musicJ=musicJ)

```

Aqui, fazemos a chamada da função `updatemusic` e obtemos a lista com as informações das músicas no banco de dados com o comando `musicJ = get_musics()`.

Vamos modificar a view `home.html` para não gerar erro, caso a lista `musicJ` esteja vazia, na inicialização da imagem de capa:

```
<div class="col s12 center-align ">
  {% if musicJ|length > 0 %}
    
  {% else %}
    
  {% endif %}
</div>
```

Veja que na versão anterior indicamos que a fonte da imagem de capa era `src="{{ musicJ[0]['coverURL'] }}"`, ou seja, estamos usando a capa da primeira música da lista. Se a lista estiver vazia, teremos erro, assim, só inicializaremos a imagem de capa se o tamanho de `musicJ` for maior que zero: `{% if musicJ|length > 0 %}`.

10.4 CONCLUSÃO

Neste capítulo, foi apresentado o `pyDAL` e como podemos modelar e conectar um banco de dados usando uma camada de abstração. Também vimos como fazer a integração do banco de dados com a aplicação `Flask`, para que nosso sistema possa trocar informações com o banco de dados.

No próximo capítulo vamos continuar a integração com o banco de dados, ao criar um sistema de login, com cadastro de

usuários e recuperação de senhas.

SISTEMA DE LOGIN

Ao desenvolver um sistema Web, podemos aproveitar a integração com banco de dados para criar acesso personalizado ao site. Ou seja, podemos ter um aplicativo multiusuários, no qual a informação de cada pessoa só será acessada através de um sistema de login com senha. As informações de acesso do usuário também serão armazenadas no banco de dados, mas a senha deve ser guardada de forma criptografada, pois se ocorrer algum vazamento de dados, ao menos as senhas de acessos estarão seguras.

Neste capítulo, vamos continuar a integração do banco de dados a nossa aplicação Flask. Aqui vamos necessitar de dois módulos adicionais, o `flask-login`, que permitirá manipular informações de seção dos usuários. O outro módulo será o `passlib`, que usaremos para criptografar a senha do usuário, que será armazenada no banco de dados, e permitir o acesso do usuário somente se com a senha digitada for possível descriptografar a senha armazenada no banco.

A extensão `flask-login` permite manipular tarefas de login e logout, e armazenar seções de usuário por um período de tempo. Sua instalação é feita pelo comando:

```
pip install flask-login
```

Outro ponto importante é que não devemos guardar a senha do usuário no banco de dados diretamente. O ideal é criptografar esse dado. Para isso, vamos usar o módulo adicional `passlib`. Ele nos permitirá criptografar os dados e comparar a senha criptografada no banco de dados com a senha digitada pelo usuário. Para instalar o `passlib` usamos o seguinte comando:

```
pip install passlib
```

Também vamos utilizar o módulo `flask-mail`, que usaremos para enviar e-mail de recuperação de senha para o usuário. Para instalá-lo, fazemos:

```
pip install flask-mail
```

Com as dependências instaladas, podemos começar a criar o sistema de login.

11.1 ORGANIZANDO O NOSSO CÓDIGO

Iniciaremos organizando o nosso código. Vamos deixar no módulo `fplayer.py` somente as funções de view, e todas as outras funções ficarão salvas em um módulo chamado `tools.py`:

```
#!/usr/bin/env python
# -*- Coding: UTF-8 -*-

'''
Módulo tools com funcoes auxiliares para o fplayer.
'''

from flask import url_for
import glob

from mutagen import File
from model import model
```

```

MUSICFOLDER = 'static/musics/'

def updatemusic():
    db = model()
    musiclist = glob.glob(MUSICFOLDER + "*.mp3")
    musicnames = [mi.split("/")[1] for mi in musiclist]

    indb = [msi.arquivo for msi in db().iterselect(db.musica.arqu
ivo)
            if msi.arquivo in musicnames]
    notindb = list(set(musicnames) - set(indb))

    for msi in notindb:
        tag = File(MUSICFOLDER + msi)
        if 'TIT2' in tag.keys():
            db.musica.insert(nome=tag['TIT2'].text[0],
                             cantor=tag['TPE1'].text[0],
                             arquivo=msi,
                             tempo=sec2minString(
                                 File(MUSICFOLDER + msi).info.length)
                             )
        else:
            db.musica.insert(arquivo=msi,
                             tempo=sec2minString(
                                 File(MUSICFOLDER + msi).info.len
gth)
                             )

    notindir = [msi.arquivo for msi in db().iterselect(db.musica.
arquivo)
                if msi.arquivo not in musicnames]

    for msi in notindir:
        db(db.musica.arquivo == msi).delete()
    db.commit()

def get_musics():
    db = model()
    musiclist = db().select(db.musica.arquivo,
                             db.musica.tempo,
                             db.musica.cantor,
                             db.musica.nome,
                             orderby=-db.musica.arquivo | db.music
a.nome

```



```

    )
    if len(musiclist) > 0:
        musicJ = [{"fileName": mi.arquivo,
                    "coverURL": url_for('coverImage',
                                         music=MUSICFOLDER + mi.arq
uivo),
                    'fileUrl': url_for('sounds',
                                         music=MUSICFOLDER + mi.arqu
ivo),
                    'length': mi.tempo,
                    'Tags': None
                    } for mi in musiclist]

        for i in range(len(musicJ)):
            if musiclist[i].cantor is not None:
                musicJ[i]['Tags'] = {
                    'TIT2': musiclist[i].nome, 'TPE1': musiclist[
i].cantor}
            else:
                musicJ = []
        return musicJ

def sec2minString(sec):
    mi = sec / 60.0
    mi = str(mi).split(".")
    seci = int(float('0.' + mi[1]) * 60.0)
    if(seci < 10):
        seci = '0' + str(seci)
    else:
        seci = str(seci)

    return mi[0] + ":" + seci

```

No código anterior, a única coisa que fizemos foi copiar as funções que estavam no arquivo `fplayer.py` para `tools.py`.

No arquivo `flplayer.py`, vamos importar os módulos: i) `Flask` - módulo principal do framework; ii) `url_for` - permite gerar URLs dinâmicas das funções que representam as páginas da

aplicação; iii) `render_template` - renderiza os templates na forma de HTML5; iv) `send_file` - permite o envio de arquivos do servidor para o usuário; v) `request` - permite acesso às informações de requisição; vi) `flash` permite o envio de mensagens de log para a camada do usuário; vii) `redirect` - faz o redirecionamento para uma determinada URL.

Além desses módulos do Flask, vamos usar o `flask_login`, para facilitar o controle de acesso ao site, `passlib` que será usado para criptografar a senha do usuário no banco de dados, `flask_mail` para o gerenciamento de envio de e-mails aos usuários. Por fim, chamaremos o módulo `datetime` que é usado para manipulação de datas.

Dessa forma, o módulo terá o seguinte conteúdo:

```
from flask import Flask, url_for, render_template
from flask import send_file, request, flash, redirect

import flask_login
from passlib.hash import sha256_crypt
from flask_mail import Mail

import datetime

from tools import *
from model import model

app = Flask(__name__)
app.secret_key = "iojsapklamnsetoiaqwerb"

login_manager = flask_login.LoginManager()

login_manager.init_app(app)

app.config.update(
```

```

DEBUG=True,
#EMAIL SETTINGS
MAIL_SERVER='smtp.gmail.com',
MAIL_PORT=587,
MAIL_USE_SSL=False,
MAIL_USERNAME = "meuemail@gmail.com",
MAIL_PASSWORD = "minhasenha"
)

```

```
mail = Mail(app)
```

Para poder usar os recursos de login, é necessário criar uma chave secreta para a aplicação. Isso foi feito usando o comando `app.secret_key = "iojsapklamnsetoiaqwerb"` . A string representando a chave pode ter qualquer valor que o desenvolvedor desejar. Em seguida, instanciamos o objeto `login_manager` e inicializamos o sistema de login, passando a aplicação Flask ao método de inicialização `login_manager.init_app(app)` . Adicionamos configurações à nossa aplicação, definindo o servidor de SMTP, que é um servidor de envio de e-mail. Atualmente, com as políticas antispam, somente com um servidor SMTP profissional será garantido que o e-mail enviado pelo seu sistema chegará ao usuário. Os servidores gratuitos de e-mail possuem uma cota diária de envio de mensagens, mas são ótimas opções para quem está iniciando. Com uma pesquisa rápida na internet, é possível encontrar boas opções. É possível usar uma conta do Gmail como servidor de SMTP, mas será preciso configurar a conta do Gmail para liberar o envio via SMTP. Por fim, instanciamos um objeto de manipulação de e-mail para o nosso sistema `mail = Mail(app)` .

11.2 CADASTRO DE USUÁRIOS

Para o cadastro de usuários, vamos criar uma função que

recebe tanto o método GET quanto o método POST :

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    return "Bem-vindo"
```

O método GET é utilizado quando se quer passar poucas ou pequenas informações através da URL `http://localhost:5500/fplayer/sounds?music=teste.mp3` . Para o método GET , ao construir a URL, colocamos o endereço principal da página que queremos acessar (<http://localhost:5500/fplayer/sounds>). Ao final desse endereço que representa a função que queremos requisitar, no caso `sounds` , colocamos a interrogação e, após ela, o valor do parâmetro que queremos passar à nossa página, que no caso do exemplo é `music=teste.mp3` .

Já o método POST passa os dados para o servidor via URI (*Uniform Resource Identifier*), permitindo a passagem de um volume maior de dados e não expõe a informação enviada, como no caso de passagem via URL. O envio de dados pelo método POST é feito através de formulários, informando na tag `<form>` que o dado deverá ser entregue ao servidor via POST .

Na função `register` , vamos verificar se a mensagem veio via POST ou via GET . Caso tenha chegado via POST , vamos extrair as informações do formulário e verificar se o e-mail do usuário, a ser cadastrado no sistema, já existe no banco de dados ou não:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    db = model()
    if request.method == 'POST':
        name = request.form['username']
        email = request.form['email']
```

```
password = request.form['pw']
query = db(db.user.email == email)
```

Aqui usamos o objeto `request` para obter as informações do formulário e qual método foi passado para a função `register`. Note que `form` é um dicionário, cuja chave será o id do campo do formulário, definido no template da view. Na última linha do código anterior, criamos uma `query` para pesquisar se o e-mail digitado pelo usuário já existe ou não no banco de dados. Se o e-mail não existir no banco, o próximo passo será fazer a criptografia da senha digitada e armazenar as informações no banco de dados:

```
if query.empty() is True:
    password = sha256_crypt.encrypt(password)
    db.user.insert(name=name,
                   email=email,
                   password=password)
    db.commit()
    return redirect(url_for('login'))
```

Se os dados foram salvos corretamente, em seguida o usuário será redirecionado para a tela de login. Caso algo de errado ocorra, o site retornará para a tela de registro. A função completa para o sistema de registro de usuários é apresentada a seguir:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    db = model()
    if request.method == 'POST':
        name = request.form['username']
        email = request.form['email']
        password = request.form['pw']
        query = db(db.user.email == email)
        if query.empty() is True:
            password = sha256_crypt.encrypt(password)
            db.user.insert(name=name,
                           email=email,
                           password=password)
            db.commit()
```

```

        return redirect(url_for('login'))
    else:
        return render_template('register.html')

    else:
        return render_template('register.html')

```

Note que, se o método passado for do tipo GET , então a página de registro será aberta.

Na pasta templates devemos criar o arquivo register.html :

```

{% extends "layout.html"%}
{% block content %}
<form action='register' method='POST'>
  <input type='text' name='username' id='username' placeholder='u
sename'></input>
  <input type='text' name='email' id='email' placeholder='email'>
</input>
  <input type='password' name='pw' id='pw' placeholder='password':
</input>
  <input type='submit' name='submit'></input>
</form>
{% endblock %}

```

No bloco content adicionamos o formulário. Veja que a tag form recebeu os atributos action='register' , indicando que, após o usuário clicar no botão de submeter, deve-se chamar a função register . O outro atributo adicionado foi o method='POST' . Por padrão, um formulário envia dados usando o método GET , por isso foi necessário indicar que queremos passar os dados usando GET . Dentro do formulário adicionamos os campos de entrada do formulário. Para cada campo, adicionamos o id que será usado no dicionário request.form usado no servidor. A seguir está a imagem da tela de registro de usuários:

Figura 11.1: Página de registro de novos usuário.

11.3 TELA DE LOGIN

Agora que foi desenvolvido o sistema de registro de usuários, podemos criar as funções necessárias para efetivar o login. Primeiramente, vamos precisar de uma classe para gerenciamento dos usuários, chamada `User`. Essa classe estenderá a classe `UserMixin` do `flask-login`.

```
from flask_login import UserMixin
from model import *

class User(UserMixin):

    def __init__(self, email, password):
        self.db = model()
        self.query = self.db(self.db.user.email == email)

        if self.query.isempty() is False:
            self.userid = self.query.select(self.db.user.id,
                                           self.db.user.password
                                           ,
                                           self.db.user.name
                                           ).first()
```

```
self.email = email
self.name = self.userid.name
self.password = password
```

A classe recebe o e-mail e a senha do usuário. No seu construtor, verificamos se o e-mail do usuário existe no banco de dados. Se sim, inicializamos os atributos da classe de que vamos necessitar.

Para a classe `User` precisamos implementar os métodos:

1. `is_authenticated`, que deve retornar `True` se o usuário estiver autenticado:

```
def is_authenticated(self):
    if(self.query.isempty() is True):
        return False
    if sha256_crypt.verify(self.password, self.userid.password):
        return True
    return False
```

Nesse método, comparamos a senha digitada pelo usuário com a senha salva no banco de dados. Usamos o método `sha256_crypt.verify` para verificar se a senha digitada corresponde à senha criptografada que está no banco de dados.

1. `is_active`, que retorna `True` caso o usuário esteja ativo no sistema:

```
def is_active(self):
    return True
```

Forçar que essa função retorne `True` indica que todo usuário cadastrado no sistema estará ativo imediatamente.

1. `is_anonymous`, retorna `True` se o sistema permitir usuário anônimo:


```
def is_anonymous(self):  
    return False
```

Aqui estamos indicando que nosso sistema não permite acesso anônimo.

1. `get_id` retorna o unicode representando o id do usuário:

```
def get_id(self):  
    if(self.query.isempty() is True):  
        return None  
  
    return self.email
```

Essa função retorna o e-mail do usuário representando um identificador único.

Além desses métodos, criaremos um método de classe. De forma geral, quando criamos uma classe, definimos os métodos como pertencentes a instâncias, ou seja, só podemos acessar os métodos através de objetos já instanciados e não pela classe. Mas, algumas vezes, pode ser importante acessar um método diretamente pela classe, quando precisamos ter um método na própria classe para a criação de objetos. No nosso caso, criaremos um método de classe que, ao receber um e-mail válido de usuário, retorna um objeto do tipo usuário (`user`). Isso será útil para carregar os dados do usuário após a efetivação do login. Esse último ponto é requerido pelo módulo `flask-login`.

```
@classmethod  
def get_user(cls, user_id):  
    db = model()  
    query = db(db.user.email == user_id)  
    if query.isempty() is True:  
        return None  
    user = query.select(db.user.email, db.user.password).first()  
    return cls(user.email, user.password)
```

No método anterior usamos o decorador `@classmethod` para indicar que esse será um método de classe. Note que, em vez de receber como primeiro argumento o `self`, agora ele recebe o `cls`. Assim, o método de classe recebe a própria classe e não uma instância de classe. Aqui também passamos o `user_id` como argumento do método. Esse id é usado para verificar se o usuário existe no banco de dados. Caso nossa query não esteja vazia, resgatamos a informação de e-mail do usuário e senha. Em seguida, criamos objeto do tipo usuário, lembrando que aqui `cls` representa a própria classe `User`.

Vamos criar um módulo chamado `user.py` e, dentro dele, a classe `User`. A seguir está o código completo desse novo módulo:

```
from flask_login import UserMixin
from passlib.hash import sha256_crypt
from model import *

class User(UserMixin):

    def __init__(self, email, password):
        self.db = model()
        self.query = self.db(self.db.user.email == email)
        self.errorlogin = 0

        if self.query.isempty() is False:
            self.userid = self.query.select(self.db.user.id,
                                            self.db.user.password
                                            ,
                                            self.db.user.name
                                            ).first()

            self.email = email
            self.name = self.userid.name
            self.password = password

    def get_id(self):
        if(self.query.isempty() is True):
            return None
```

```

        return self.email

    def is_active(self):
        return True

    def is_authenticated(self):
        if(self.query.isempty() is True):
            self.errorlogin = 1
            return False

        if sha256_crypt.verify(self.password, self.userid.password):
            return True

        self.errorlogin = 2
        return False

    def is_anonymous(self):
        return False

    @classmethod
    def get_user(cls, user_id):
        db = model()
        query = db(db.user.email == user_id)
        if query.isempty() is True:
            return None
        user = query.select(db.user.email, db.user.password).first()
        return cls(user.email, user.password)

```

Tendo a classe que representa o usuário, podemos voltar ao módulo principal da aplicação e criar as funções de login.

11.4 LOGIN

Primeiramente, criaremos uma função que será usada pelo flask-login para recarregar um objeto do tipo usuário a partir do id armazenado na seção:

```

@login_manager.user_loader
def load_user(user_id):

```

```
return User.get_user(user_id)
```

No código anterior, usamos o decorador `@login_manager.user_loader` para indicar que essa será a função usada para recarregar um objeto do tipo `User`. Essa função recebe o id do usuário que está armazenado na variável de sessão, gerenciada pelo `flask-login`. Ao final do código, fazemos o uso do método de classe. Observe que usamos o nome da classe seguida pelo método de classe. Assim, o método `get_user` é uma fábrica de instâncias dos objetos do tipo `User`.

Outras funções importantes são a de `logout` e gerenciamento de tentativa de acesso não autorizado:

```
@app.route('/logout')
def logout():
    flask_login.logout_user()
    return redirect(url_for('login'))

@login_manager.unauthorized_handler
def unauthorized_handler():
    return redirect(url_for('login'))
```

Para o `logout` simplesmente usamos o método `flask_login.logout_user()`, para encerrar a sessão do usuário. Em seguida, fazemos o redirecionamento para a tela de login.

Já para o gerenciamento de acesso não autorizado, usamos o decorador `@login_manager.unauthorized_handler` e fizemos o redirecionamento para a tela de login. Assim, sempre que uma pessoa tentar acessar, via URL, por exemplo, uma view que exige login, se ela não estiver autenticada, ela será redirecionada para a tela de login.

Finalmente, criaremos a tela de login:

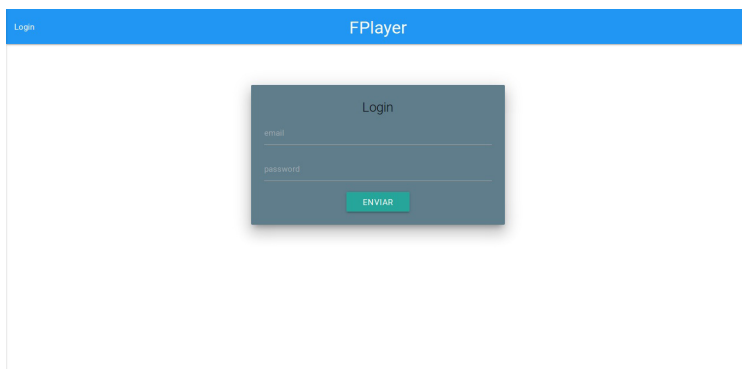


Figura 11.2: Tela de Login.

O template da tela de login é dado pelo seguinte código:

```
{% extends "layout.html"%}
{% block content %}
<br>
<br>
<br>
<div class="row">
  <div class="col s6 offset-s3 center-align">
    <div class="card blue-grey z-depth-5">
      <div class="card-content">
        <span class="card-title">Login</span>
        <form action='login' method='POST'>
          <input type='text' name='email' id='email' placeholder='email'></input>
          <input type='password' name='pw' id='pw' placeholder='password'></input>
          <button class="btn waves-effect waves-light" type="submit" name="submit">Enviar
        </button>
        </form>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

Aqui é importante ressaltar o uso da tag `<form`

`action='login' method='POST'>` que tem como ação a função `login` e que envia os dados via método `POST`.

A função de view para o login será:

```
@app.route("/login", methods=["GET", "POST"])
def login():
    if flask_login.current_user.is_authenticated:
        return redirect(url_for("home"))

    if request.method == 'GET':
        return render_template("login.html")

    email = request.form['email']
    password = request.form['pw']

    user = User(email, password)

    if user.get_id() is None:
        return render_template("login.html")

    flask_login.login_user(user)
    return redirect(url_for('home'))
```

Primeiramente, verificamos se existe atualmente algum usuário logado (`lask_login.current_user.is_authenticated`). Caso isso seja verdade, o sistema redirecionará o usuário para a página `home` . Se o método recebido for do tipo `GET` , será carregado o template da página de login. Caso sejam passadas informações via `POST` , criamos um objeto do tipo `User` , e verificamos se foi realmente possível obter o id do usuário. Caso o objeto `user` tenha sido criado corretamente, ele é passado para o `flask_login.login_user(user)` , que validará as informações. Se tudo estiver correto, uma nova seção será criada e o usuário será redirecionado para a página `home` .

Como último elemento, vamos indicar que a página `home` só será acessada após o usuário fazer login no sistema. Para isso,

usaremos o decorador `@flask_login.login_required` :

```
@app.route("/")
@flask_login.login_required
def home():
    updatemusic()
    musicJ = get_musics()
    return render_template("home.html",
                           musicJ=musicJ,
                           username=flask_login.current_user.name
    )
```

Aqui estamos passando para o template da view o nome do usuário. Com isso, podemos customizar a página `home` e adicionar uma mensagem de boas-vindas personalizada para o usuário.

11.5 CONCLUSÃO

Neste capítulo, foi abordada a criação de um sistema de cadastro e login de usuários. Vimos que informações sensíveis, como senhas, devem ser armazenadas no banco de dados de forma criptografada. Com isso, conseguimos garantir algum nível de privacidade aos dados do usuário, caso ocorra algum tipo de quebra de segurança no gerenciamento de banco de dados.

No próximo capítulo, exploraremos a configuração de um servidor de produção, usando Apache, Linux e Flask.

LAF - LINUX, APACHE E FLASK

Agora que temos nossa aplicação pronta, precisamos ser capazes de rodar nosso sistema em um servidor de produção. Uma das opções mais utilizadas atualmente é o conjunto Apache e Linux. Aqui vamos aprender como configurar um ambiente de produção usando a combinação de Linux, em particular a distribuição Ubuntu 16.04, Apache e Flask. Vamos chamar essa pilha de softwares de LAF. Vale ressaltar que, atualmente, existem inúmeros serviços de hospedagem que oferecem a opção de rodar aplicações Web em um sistema do tipo *unix* (Linux/Unix). Portanto, o conhecimento de como configurar um servidor LAF poderá ser aplicado de forma prática e profissional.

12.1 DIALOGO ENTRE FLASK E APACHE

Primeiramente, vamos precisar do sistema operacional Ubuntu com acesso de superusuário. Aqui vale ressaltar que a maioria dos comandos executados nesta seção são executados no modo de superusuário.

O servidor Apache foi criado em 1995 por Rob McCool, sendo um servidor do tipo HTTPD (*HyperText Transfer Protocol*

Daemon). A estrutura do Apache é organizada em módulos. Para rodar a aplicação Flask vamos usar o módulo `mod_wsgi`. WSGI (*Web Server Gateway Interface* - Interface de Porta de Entrada do Servidor Web) é uma especificação para uma interface para comunicação entre servidores Web e aplicações Web, ou frameworks, escritos na linguagem Python. Para instalar o Apache e o `mod_wsgi`, no Ubuntu, usamos o seguinte comando, como superusuário:

```
apt-get install apache2 libapache2-mod-wsgi
```

Vamos salvar a pasta contendo o projeto dentro da pasta `/var/www`. O caminho da nossa aplicação Web será `/var/www/FPlayer`. Na pasta raiz do projeto, vamos criar um ambiente virtual, usando `virtualenv` para instalação das dependências do nosso projeto, de forma a manter nossa aplicação de forma independente de qualquer outra que queiramos instalar no nosso servidor. Isso permite evitar conflitos ao se instalarem muitos sistemas diferentes no mesmo servidor. Como superusuário, via terminal, dentro do caminho `/var/www/FPlayer` digite o seguinte comando:

```
virtualenv env
```

Isso vai criar a pasta contendo todas as informações e programas para rodar nosso ambiente virtual. Em seguida, vamos ativar esse ambiente virtual e instalar as dependências do projeto.

```
source env/bin/activate
```

Após ativar o ambiente, basta instalar as dependências do projeto. Se durante a fase de desenvolvimento foi usado um ambiente virtual, podemos criar um arquivo de requerimentos do projeto, permitindo a instalação das dependências de forma

automática em outras máquinas. Para fazer isso, ative o ambiente virtual da fase de dependência e use o seguinte comando para gerar o arquivo `requirements.txt`, com as relações de dependências do projeto.

```
pip freeze >> requirements.txt
```

O arquivo `requirements.txt` deve estar na raiz do projeto. Para instalar as dependências no novo ambiente, após a sua ativação, basta usar o comando:

```
pip install -r requirements.txt
```

Para conseguirmos fazer com que o Apache converse com a nossa aplicação Flask, precisamos criar um programa em Python, chamado `wsgi.py`. Na raiz da aplicação, crie o programa `wsgi.py` contendo o seguinte conteúdo:

```
import sys
activate_this = "/var/www/FPlayer/env/bin/activate_this.py"

execfile(activate_this, dict(__file__=activate_this))

PROJECT_DIR = "/var/www/FPlayer"

sys.path.insert(0, PROJECT_DIR)

from fplayer import app as application
```

Na primeira linha desse programa, chamamos o módulo `sys`; em seguida, indicamos o caminho do programa que permite a ativação do ambiente virtual que contém as dependências do projeto. O comando `execfile` vai inicializar, de forma automática, o ambiente virtual, permitindo a execução da nossa aplicação Web. O comando `sys.path.insert(0, PROJECT_DIR)` adiciona o diretório do nosso projeto ao caminho de busca por módulos. Isso vai deixar nossa aplicação visível para o

módulo `wsgi.py` . Em seguida, importamos nossa aplicação Flask como `application` .

O próximo passo será o de alterar o arquivo `hosts` , que está na pasta `/etc` (`/etc/hosts`), para incluir o endereço `www.fplayer.loc` , permitindo fazer o redirecionamento dessa URL para o endereço de IP, `127.0.0.1`, no qual estamos rodando a nossa aplicação:

```
127.0.0.1 www.fplayer.loc
```

Agora vamos configurar o Virtual Host, para fazer a conexão da nossa aplicação com o Apache. Para isso, vamos criar um arquivo chamado `fplayer.conf` na pasta `/etc/apache2/sites-available` com o seguinte conteúdo:

```
<VirtualHost *:80>
    ServerName www.fplayer.loc
    WSGIDaemonProcess flaskFplayer user=www-data group=www-data
    threads=5
    WSGIScriptAlias / /var/www/FPlayer/wsgi.py
    ErrorLog /var/www/FPlayer/logs/error.log
    CustomLog /var/www/FPlayer/logs/access.log combined

    <Directory /var/www/FPlayer>
        WSGIProcessGroup flaskFplayer
        WSGIApplicationGroup %{GLOBAL}
        WSGIScriptReloading On
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

Na primeira linha, criamos um servidor virtual que aponta para a porta 80. Em seguida, indicamos o nome do servidor, o qual será usado para apontar a URL digitada no browser, ou seja, para a nossa aplicação Flask.

O comando `WSGIDaemonProcess` indica que nossa aplicação vai executar em plano de fundo, pertencendo ao usuário `www-data` e ao grupo `www-data`, em no máximo cinco linhas de processamento. O comando `WSGIScriptAlias` vai chamar o programa `wsgi.py`, executando assim a nossa aplicação Web.

Na tag `Directory`, indicamos as permissões de acesso ao conteúdo da aplicação. Nossos arquivos de log ficaram na pasta `logs`. Assim, usamos o seguinte comando para criar essa nova pasta:

```
mkdir /var/www/FPlayer/logs/
```

Para habilitar o nosso site no servidor Apache, precisamos abrir o terminal, ir para a pasta `/etc/apache2/sites-available`, e em seguida digitar os seguintes comandos:

```
cd /etc/apache2/sites-available
```

```
a2ensite fplayer.conf
```

```
service apache2 reload
```

Na segunda linha, ativamos o nosso servidor virtual, em seguida, recarregamos o Apache, para efetivar a configuração anterior.

Vamos garantir que a pasta `FPlayer` pertence ao usuário e ao grupo correto. Para isso, digitamos o seguinte comando no terminal:

```
chown -R www-data:www-data /var/www/FPlayer
```

Agora, no browser, basta digitar o endereço `www.fplayer.loc` para ver a aplicação em execução.

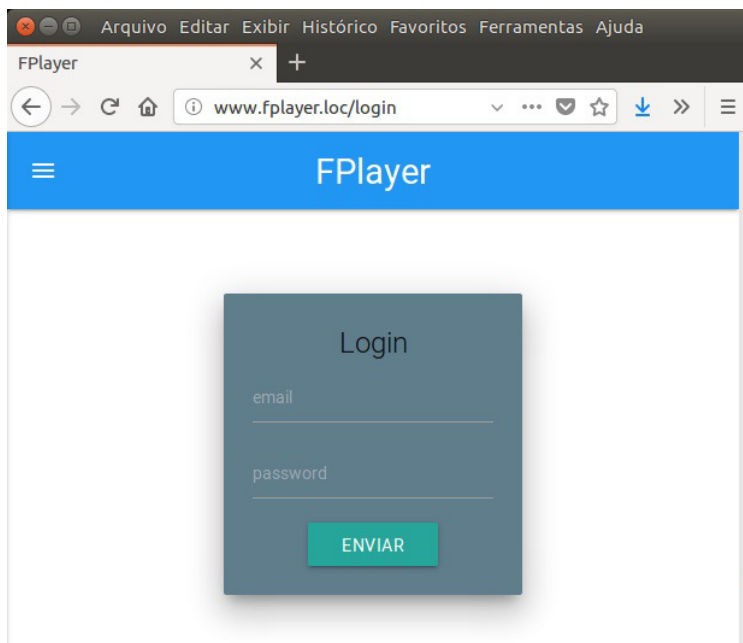


Figura 12.1: Execução da aplicação Flask via servidor Apache

12.2 CONCLUSÃO

Neste capítulo, conhecemos um pouco sobre o servidor de produção Apache e como configurar uma aplicação escrita em Flask para conversar com esse servidor usando o WSGI. Também foi apresentado como usar o ambiente virtual que permite separar as dependências da aplicação do resto do sistema. Isso é importante quando se tem várias aplicações Web rodando no mesmo servidor, pois elas poderão ter dependências conflitantes. Dessa forma, rodar cada aplicação em um ambiente isolado facilita a manutenção do servidor, ao evitar possíveis conflitos entre versões de bibliotecas.

Considerações finais

CONSIDERAÇÕES FINAIS

Ao longo deste livro, foram apresentadas as principais características da linguagem Python, buscando sempre uma abordagem prática. Nos dois primeiros capítulos, falou-se sobre os elementos fundamentais de Python e sua estrutura semântica e de dados. Vimos conceitos como funções, estruturas condicionais e tipos primitivos. Em seguida, usando fundamentos de desenvolvimento de jogos, apresentamos como é a Programação Estruturada nessa linguagem.

Como a lógica de programação não é algo universal, na segunda parte do livro buscou-se apresentar de forma mais detalhada o significado de se ter uma linguagem multiparadigma. Vimos que, na verdade, não podemos dizer lógica de programação, como algo generalizado, mas sim, que existem formas diferentes de se resolver problemas. A esse conjunto de regras e métodos de solução de problemas, aplicados à programação, é que chamamos de *paradigmas*. Assim, foi abordado o paradigma de Orientação a Objetos e Programação Funcional, dentro do escopo da linguagem Python.

Na terceira parte do livro, vimos como organizar os conhecimentos das partes anteriores, em conjunto com o framework Flask, para o desenvolvimento de aplicações Web.

Nesse último caso, vimos que para desenvolver uma aplicação Web é necessário ir além de programação em Python, pois na camada do usuário é preciso usar recursos de HTML, CSS e JavaScript. Devido à importância do JavaScript no desenvolvimento Web, há um apêndice cobrindo os principais pontos dessa linguagem ao fim deste livro.

REFERÊNCIAS BIBLIOGRÁFICAS

ABREU, Luís. *JavaScript*. Lisboa: FCA-Editora de Informática, 2011.

CRUZ, Felipe. *Python: Escreva seus primeiros programas*. São Paulo: Casa do Código, 2015.

FARHAT, Ayman. *A guide to Python's function decorators*. Disponível em <https://www.thecodeship.com/patterns/guide-to-python-function-decorators/>. Acessado em 26 de julho de 2017.

FERREIRA, Silvio. *Guia prático de HTML5*. São Paulo: Universo dos Livros, 2013.

GORDON, Richard et al. *Web2py Application Development Cookbook: Over 110 recipes to master this full-stack Python Web framework*. Birmingham: Packt Pub., 2012.

GUARDIA, Carlos de la. *Python web frameworks*. Sebastopol: O'Reilly Media, 2016. <http://www.oreilly.com/web-platform/free/python-web-frameworks.csp>

MERTZ, David. *Functional Programming in Python*. Sebastopol: O'Reilly Media, 2015.

<https://www.oreilly.com/programming/free/functional-programming-python.csp>

OSTERWALDER, Alexander, Pigneur, Yves. *Business model generation: Como desenvolver diferenciais inovadores*. São Paulo: Alta Books, 2010.

PIERRO, Massimo Di. *Web2py: Enterprise Web Framework*. New Jersey: John Wiley & Sons, 2008.

PILGRIM, Mark. *HTML5: Up and Running*. Sebastopol: O'Reilly Media, Inc. 2010.

RIES, Erick. *A startup enxuta: Como os empreendedores atuais utilizam a inovação contínua para criar empresas extremamente bem-sucedidas*. São Paulo: Texto Editores Ltda., 2012.

SANTANA, Osvaldo, Galesi, Thiago. *Python e Django: Desenvolvimento ágil de aplicações web*. São Paulo: Novatec, 2010.

SEBRAE. *O quadro de modelo de negócios: Um caminho para criar, recriar e inovar em modelos de negócios*. Em: [http://www.bibliotecas.sebrae.com.br/chronus/ARQUIVOS_CHRONUS/bds/bds.nsf/be606c09f2e9502c51b09634badd2821/\\$File/4439.pdf](http://www.bibliotecas.sebrae.com.br/chronus/ARQUIVOS_CHRONUS/bds/bds.nsf/be606c09f2e9502c51b09634badd2821/$File/4439.pdf). Acessado em 02 de julho de 2016.

CSS: Cascading Style Sheets. Disponível em: http://www.tutorialspoint.com/css/css_tutorial.pdf. Acessado em 30 de junho de 2017.

Lista de elementos do HTML5: https://developer.mozilla.org/pt-BR/docs/Web/HTML/HTML5/HTML5_element_list. Acessado em 30 de junho de 2017.

Apêndice

APÊNDICE A: JAVASCRIPT

JavaScript é uma linguagem script orientada a objetos, desenvolvida por Brendan Eich na Netscape. Foi utilizada pela primeira vez nos navegadores Netscape, sendo seus padrões definidos pela Associação Europeia para Padronização de Informação de Sistemas (ECMA). O nome oficial atual da linguagem é ECMAScript, sendo baseada tanto no JavaScript da Netscape quanto na variante JScript, criada pela Microsoft. JavaScript tem uma sintaxe que lembra muito o Java e o C, mas são linguagens muito distintas. Apesar de ser dita orientada a objetos, o JavaScript não suporta o conceito de classe, tal como vemos em Python ou no próprio Java. Porém, ela suporta o conceito de função construtora.

As especificações da ECMAScript iniciaram-se no fim de 1996, sendo que a segunda edição foi aprovada em junho de 1998. Essas especificações estão disponíveis em <https://goo.gl/to7fHK/>, sendo a 1.5 a versão mais recente da linguagem.

Neste anexo, faremos uma introdução ao JavaScript e seus principais elementos.

15.1 COMENTÁRIOS, PALAVRAS

RESERVADAS E TIPOS

O JavaScript possui uma sintaxe muito parecida com a do Java e do C/C++, no qual blocos são separados por chaves `{}` e o final de uma linha é demarcado por `;`, enquanto comentários são escritos entre `/* */` ou `//`. Porém, não podemos usar o construtor de comentários dentro de outro comentário:

```
/* Isso é um comentário */
//Isso é um comentário.

/* Isso gera um erro /* */

/* Isso gera um erro /* */ */
```

Um código JavaScript pode estar embebido no HTML ou escrito em um arquivo separado, tal como o CSS. Para escrever o script direto no HTML usamos a tag `<script>`

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Título do Arquivo</title>
  </head>
  <body>
    <script type="text/javascript">
      /* Isso é um comentário */
      // Isso é um comentário
      var i = 0;
    </script>
  </body>
</html>
```

Na primeira linha do código anterior definimos que o arquivo é do tipo HTML5. Na segunda linha, iniciamos a estrutura `html` e indicamos que o texto presente no arquivo está em português brasileiro. Em seguida, foi adicionado o cabeçalho geral com o

metadado indicando o tipo de conteúdo e a codificação do arquivo, que está no formato UTF-8, facilitando a compreensão de acentuações pelo navegador.

Também no cabeçalho, foi adicionado o título do documento. Após definir o corpo do arquivo, com a tag `<body>`, adicionamos a tag `<script>` e indicamos que o conteúdo do script é do tipo texto e que representa a codificação do JavaScript. Em seguida, foram adicionados dois comentários. Nesse exemplo também foi definida uma variável, chamada `i`, contendo o inteiro de valor 0.

Para chamar um script escrito em um arquivo externo, fazemos um link da seguinte forma:

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Título do Arquivo</title>
    <script type="text/javascript" src="js/myScript.js"></script>
  </head>
  <body>
  </body>
</html>
```

Ao olhar a tag `<script>` que está dentro do `<head>` veja que foi adicionado o atributo `src` (fonte). Nele, indicamos o caminho, ou o link, em que se encontra o arquivo com nome `myScript.js`, no qual temos o código JavaScript.

O JavaScript é dinamicamente tipado, assim o que se faz é definir variáveis genéricas, usando por exemplo a palavra `var`. Apesar de genéricas, a linguagem determina em tempo de execução qual é o tipo que está sendo atribuído a uma variável.

Dessa forma, os tipos de dados primitivos são: números inteiros e de ponto flutuante, cadeias de caracteres ou strings, lógicas ou booleanas, sem valor (nula, `null`) ou indefinida (`undefined`) e `Symbol`, sendo que essa última indica instâncias únicas e imutáveis. A seguir está um exemplo da definição de variável primitivas:

```
/* Definindo a variável generica a */
var a; // Variável com valor indefinido ou undefined;
a = 1; // Inteiro
a = 1.0; // Ponto Flutuante
a = "Oi Mundo"; // Cadeia de Caracteres
a = true; // Boleana ou lógica
a = null; // Nula
var b = 42; // Definição de variável com um valor definido
```

Além do `var`, podemos usar o `let` e o `const`, sendo que `let` declara uma variável local de escopo do bloco, enquanto que `const` cria uma constante usada apenas para leitura.

Na primeira linha do código anterior, declaramos a variável `a`, mas não fizemos nenhuma atribuição. Nesse caso, a variável tem valor `undefined`, ou seja, ela é indefinida. Em seguida, fomos mudando o valor de `a`. Como o JavaScript é dinamicamente tipado, a variável `a` foi modificada para número inteiro, em seguida, para ponto flutuante, passando por cadeia de caracteres. Depois, a variável `a` foi convertida para uma variável lógica e finalmente fizemos com que ela fosse do tipo nula. Na última linha do código anterior criamos a variável chamada `b` e atribuímos a ela o valor inteiro 42.

JavaScript possui algumas palavras reservadas, que não devem ser usadas para definir variáveis ou funções. Segue uma lista com elas:

default, return, var, case, delete, if, switch, void, catch, do, in, this, while, const, else, instanceof, throw, continue, finally, let, try, debugger, for, typeof, new, with, break, function.

15.2 FERRAMENTA DE DESENVOLVIMENTO DO NAVEGADOR

Os navegadores modernos trazem uma ferramenta de desenvolvimento com várias opções. Uma delas é a de exibir mensagens em um console de log. No caso do JavaScript, podemos escrever algo nessa saída usando o comando `console.log`. Para acessar a console de log no Firefox, por exemplo, basta clicar no menu, em seguida no botão `Developer` e finalmente no menu `Web Console`. Na figura a seguir temos o *Web Console* do Firefox 54.0 aberto:

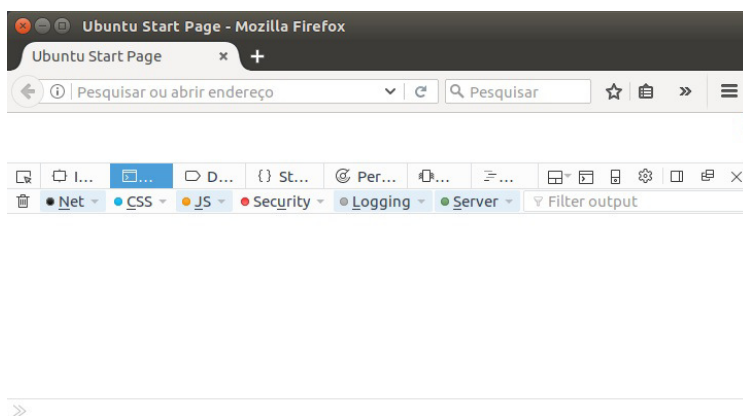


Figura 15.1: Console usado para visualizar as saídas geradas pelo JavaScript através do comando `console.log` no Firefox 54.0

Com essa ferramenta podemos analisar trechos do nosso código, ver informações de erros gerados ao codificar o JavaScript, ajudando a debugar o nosso programa. Também conseguimos

mapear o comportamento do CSS aplicado na página, além de poder ver as informações que são geradas pelo log, obtendo informações do nosso programa em tempo de execução.

15.3 ESCOPO DE VARIÁVEIS

Uma variável declarada fora de qualquer função terá escopo global, ou seja, ela poderá ser modificada por outras funções a qualquer momento. Se usarmos `var` para declarar variáveis dentro de um bloco, ela terá escopo global, porém se usarmos `let` ela terá escopo local:

```
// Caso 1:
if(true){
  //Declaração de variável em escopo global
  var z = 42;
}
console.log(z);

// Caso 2:

if(true){
  //Declaração de variável em escopo local para um bloco;
  let w = 54;
}

console.log(w); //ReferenceError: w não está definido
```

No primeiro caso, `z` foi declarado com valor 42, porém seu acesso é global, logo, ao se utilizar o `console.log` não será levantado nenhum erro e o valor da variável aparecerá no *Web Console*. Porém, ao usarmos `let` para declarar `w`, essa variável passa a existir somente dentro do bloco do `if` demarcado pelas chaves. Ao se tentar exibir a variável `w` no console, fora do escopo do `if`, será retornado um erro, indicando que `w` não está

definido.

Outro ponto exótico em JavaScript é que podemos usar uma variável e declará-la em seguida, sem que isso levante um erro. Tal procedimento é conhecido como **hoisting**. Contudo, a variável sempre terá valor `undefined` antes de ser efetivamente declarada.

```
console.log(x); //Exibe undefined
var x = 42;
```

Note que pedimos para exibir a variável `x` no console. Nesse caso, o resultado será a palavra `undefined`, indicando que a variável ainda não foi definida. Só na linha seguinte é que fazemos efetivamente a declaração de `x`.

O código anterior é equivalente ao seguinte:

```
var x;
console.log(x);
x = 42;
```

Já o uso da palavra `const` cria uma variável que não terá seu valor modificado:

```
const y = 42;
console.log(y); // Exibe 42
y = 54;
console.log(y); // Exibe 42
```

Na primeira linha do código anterior fizemos com que `y` fosse constante e igual a `42`. Em seguida, pedimos para exibir esse valor no console. Na linha seguinte tentamos mudar o valor de `y` para `54`, porém, ao verificar se o novo valor foi atribuído a `y` veremos que seu valor continua sendo `42`.

15.4 STRINGS E CONVERSÃO DE TIPOS

Uma string é uma cadeia de caractere, na qual podemos acessar um determinado elemento usando a notação vetorial, porém para capturar trechos da string temos que usar o método `slice`, que retornará fatias da string:

```
var s = "Oi mundo";  
console.log(s[0]); // retornará o 0  
console.log(s.slice(3, 6)); // retornará mun;
```

Na primeira linha do código anterior, definimos a variável `s` como `"Oi mundo"`. Agora queremos pegar apenas a primeira letra dessa string, então usamos `s[0]`, sendo que o colchete indica que queremos pegar um determinado elemento da string. O elemento capturado foi o 0, ou seja, começamos nossa contagem a partir do 0. Em seguida, foi capturado o trecho `mun`, que representa o intervalo fechado de 3 até 5, ou seja, pedimos para fatiar começando da posição vetorial 3 até um ponto que represente um total de 6 elementos da string.

Por ser dinamicamente tipado, ao trabalharmos com números e strings em JavaScript podemos ter comportamentos diferentes para os operadores matemáticos. Por exemplo, para concatenar duas strings usamos o operador `+`, mas veja o que acontece nos seguintes casos:

```
console.log("37" + 3);  
// A resposta será "373";  
console.log("37" - 3);  
// A resposta será 34
```

No primeiro caso, o número 3 será convertido em string e em seguida concatenado com a string `"37"`, gerando como saída a string `"373"`. No segundo caso, a string `"37"` será convertida em número e, em seguida, será subtraído o valor 3. Este último caso vale para os operadores de multiplicação (`*`) e divisão (`/`);

Para converter uma string para número podemos usar as funções internas `parseInt` , `parseFloat` :

```
var a = "1.3";
var b;
b = parseInt(a, 10); // b tem valor numérico Inteiro igual a 1
b = parseFloat("1.3") // b tem valor numérico de ponto flutuante
igual a 1.3
```

O `parseInt` recebe como parâmetro a string a ser convertida em número e a base na qual a string se encontra. Por exemplo, imagine que temos o número 3 na forma binária e queremos convertê-lo para inteiro de base 10:

```
var a = "11"; // 3 em binário
var b = parseInt(a, 2); // 0 2 indica que a está na forma binária
, base 2;
//b terá valor 3.
console.log(b);
```

No primeiro caso, escrevemos o número 3 na forma binária 11 na forma de string. Em seguida, passamos essa string, referenciada por `a` , para a função `parseInt` e também indicamos que essa variável está na base 2. Assim a variável `b` passou a armazenar o inteiro de valor 3.

Além das funções internas, podemos usar o operador `+` para converter string em número:

```
+"1.1"; // String convertida para float 1.1
+"1.1" + +"2.9"; // O valor resultante dessa operação será o inteiro 4
```

Nesse caso, o que fizemos foi adicionar o sinal de `+` ao início da string e isso a converteu em número. No segundo caso, fizemos a soma de 1.1 com 2.9, que estavam na forma de string, que resultou no valor 4.

15.5 ARRAYS OU LISTAS

Os Arrays são listas capazes de armazenar outros valores, colocados entre colchetes e separados por vírgula:

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
console.log(cafeDaManha);
// Saída: ["ovos", "pão", "leite", "café"]
```

Assim como para strings, podemos acessar um elemento da lista através de índices:

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
console.log(cafeDaManha[1]); // Retorna o "pão"
```

Veja que `cafeDaManha[1]` permite acessar o segundo elemento da lista representado pela string "pão" .

Para obter o número de elementos na lista usamos o atributo `length` , já para obter trechos da lista usamos o método `slice` :

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
console.log(cafeDaManha.length); // Retornará 4
console.log(cafeDaManha.slice(0,2)); // Retornará [ 'ovos', 'pão' ]
```

O uso de `cafeDaManha.length` retornará o valor 4, representando o total de elementos contidos na lista. Já o código `cafeDaManha.slice(0,2)` retornará uma nova lista contendo os elementos ovos e pão .

Para concatenar duas listas usamos o método `concat` :

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
cafeDaManha.concat(["suco de laranja", "maçã"]);
console.log(cafeDaManha);
//Retornará:
// [ 'ovos', 'pão', 'leite', 'café', 'suco de laranja', 'maçã' ]
```

Se quisermos remover o último elemento de uma lista usamos o método `pop` :

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
cafeDaManha.pop();
console.log(cafeDaManha);
//Retornará:
// [ 'ovos', 'pão', 'leite' ]
```

Para adicionarmos um novo elemento à lista usamos o método `push` :

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
console.log(cafeDaManha.push("biscoito")); // Retorna 5
console.log(cafeDaManha);
//Retornará:
// [ 'ovos', 'pão', 'leite', 'café', 'biscoito' ]
```

O método `push` retorna como valor o número de elementos contidos na lista, após adicionado o novo integrante.

Com o método `reverse` temos uma cópia da lista em ordem invertida:

```
var cafeDaManha = ["ovos", "pão", "leite", "café"];
cafeDaManha.reverse();
console.log(cafeDaManha);
//Retornará:
// [ 'café', 'leite', 'pão', 'ovos' ]
```

Nos navegadores mais modernos, temos ainda a opção de recuperar o índice da primeira ocorrência de um elemento em uma lista, usando o método `indexOf` . Por exemplo, na lista `['biscoito', 'café', 'leite', 'pão', 'ovos', 'café']` usando o `indexOf` para recuperar a posição da palavra `'café'` teremos como retorno o índice 1:

```
var cafeDaManha = [ 'biscoito', 'café', 'leite', 'pão', 'ovos', 'café' ]
```

```
console.log(cafedeManha.indexOf('café'));  
//Retornará: 1
```

Além disso, podemos remover um elemento de uma dada posição da lista usando o método `splice`. O que esse método faz é retornar uma cópia da lista original sem o elemento removido:

```
var cafedeManha = [ 'biscoito', 'café', 'leite', 'pão', 'ovos', 'café' ]  
var semBiscoito = cafedeManha(0, 1);  
console.log(semBiscoito);  
//Retornará: [ café, 'leite', 'pão', 'ovos', 'café' ]
```

15.6 OBJETOS

Um objeto é formado por uma lista de zero ou mais pares de nomes de propriedades e valores, colocado entre chaves. Porém, obtemos um valor usando o operador `.`, típico de Orientação a Objetos:

```
var carro = {marca: "Renault", modelo: "Sandero", potenciaDoMotor : "1.0"}  
console.log(carro.marca); // Retorna a string "Renault"  
console.log(carro.potenciaDoMotor); // Retorna a string "1.0"
```

No código anterior criamos o objeto `carro` contendo as propriedades `marca`, `modelo` e `potenciaDoMotor`. Os valores foram, respectivamente as strings "Renault", "Sandero" e "1.0". Para acessar a string "Renault" usamos a operação `carro.marca`. Para capturar a potência do motor usamos `carro.potenciaDoMotor`.

Também podemos usar `[""]` para recuperar um valor:

```
var carro = {marca: "Renault", modelo: "Sandero", potenciaDoMotor : "1.0"}  
console.log(carro["marca"]); // Retorna a string "Renault"  
console.log(carro["potenciaDoMotor"]); // Retorna a string "1.0"
```


Se quisermos modificar um valor de um dado atributo, basta fazer a sobrescrita do mesmo:

```
var carro = {marca: "Renault", modelo: "Sandero", potenciaDoMotor : "1.0"}
carro.modelo = "Stepway";
console.log(carro.modelo); // Retorna a string "StepWay"
```

JSON: JavaScript Object Notation

Quando queremos trocar informações entre o servidor e o navegador, fazemos isso através de textos. No caso de querer passar um objeto JavaScript para o navegador, teremos que usar a notação de Objetos JavaScript (*JavaScript Object Notation* - JSON). Assim, um JSON é sempre um texto. Ao receber um JSON, basta convertê-lo para objetos JavaScript:

```
var carro = {marca: "Renault", modelo: "Sandero", potenciaDoMotor : "1.0"}
var myJSON = JSON.stringify(carro);
window.location = "www.teste.com?carro=" + myJSON;
```

Na primeira linha do script anterior criamos o objeto `carro`. Na segunda, convertemos esse objeto para JSON, em seguida passamos esse JSON ao argumento `carro` da requisição ao site `www.teste.com`. Para converter um JSON em objeto usamos o método `parse`:

```
dado = localStorage.getItem("testJSON");
carro = JSON.parse(dado);
console.log(carro.potenciaDoMotor); // Retorna a string "1.0"
```

Aqui pegamos as informações contidas no arquivo local `testJSON` e em seguida transformamos o JSON no objeto `carro` usando o método `parse`.

15.7 OPERADORES LÓGICOS

Os operadores lógicos são usados para fazer comparações e testar se algo é verdadeiro ou falso.

Na tabela a seguir, temos a lista desses operadores:

| Operador | Descrição |
|----------|---|
| == | Verifica se dois elementos são iguais |
| === | Verifica se dois elementos são iguais e do mesmo tipo |
| != | Verifica se dois elementos são diferentes |
| !== | Verifica se dois elementos são diferentes e de diferentes tipos |
| > | Maior que |
| < | Menor que |
| >= | Maior ou igual a |
| <= | Menor ou igual a |

Além desses, temos o operador *e* (*and*), `&&`, ou *ou* (*or*), `||`, e o operador de negação (*not*), `!`.

Vamos analisar melhor esses operadores ao utilizá-los em estruturas condicionais e de laço.

15.8 ESTRUTURAS CONDICIONAIS E DE LAÇO

As estruturas condicionais em JavaScript são o `if/else`, `if/else`, e `switch/case`.

O `if` avalia em tempo de execução se uma determinada

condição é verdadeira ou não. Se uma dada condição não for verdadeira e quisermos avaliar uma segunda condição, usamos o `else if`. Agora, se nenhuma das duas condições forem verdadeiras usamos o `else`.

```
var a = prompt("Digite um número", 0);
if(a < 10){
    alert("O valor de a é menor que 10 .");
}else if (a == 10){
    alert("O valor de a é igual a 10");
}else{
    alert("O valor de a é maior que 10");
}
```

Na primeira linha do programa anterior foi utilizada a função `prompt`, que abre uma caixa de entrada de valores pedindo para o usuário digitar um número. Nessa caixa, vai aparecer o número 0 com valor padrão. Se o número for menor que 10, um alerta será exibido, informando que o valor entrado pelo usuário é menor que 10. Se o número for exatamente 10, o comando `else if (a == 10)` será alcançado e um alerta será exibido informando que o valor foi igual a 10. Caso nenhuma das duas condições anteriores for alcançada, o número é maior que 10 e o que estiver no bloco do `else` será executado. Nesse caso, o alerta indicando que o número é maior que 10 será exibido.

A seguir está um código HTML5 válido contendo o JavaScript do exemplo anterior:

```
<html lang="pt-br">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <script type="text/javascript">
        var a = prompt("Digite um número", 0);
```

```
if(a < 10){
    alert("O valor de a é menor que 10 .");
}else if (a == 10){
    alert("O valor de a é igual a 10");
}else{
    alert("O valor de a é maior que 10");
}
</script>
</body>
</html>
```

Observe que adicionamos na tag `html` o `lang="pt-br"` , além disso adicionamos o metadado indicando que o texto foi escrito no formato `UTF-8` . Se não fizermos isso, ao adicionar a acentuação no código JavaScript, as mensagens serão exibidas para o usuário com erro de formatação. Na figura a seguir temos o exemplo do `html` anterior sendo executado.

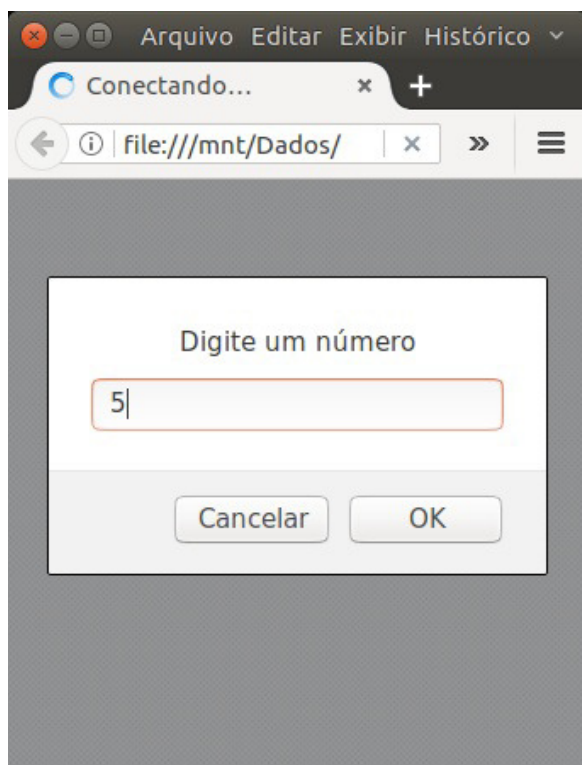


Figura 15.2: Caixa de entrada de dados gerado pelo comando prompt.

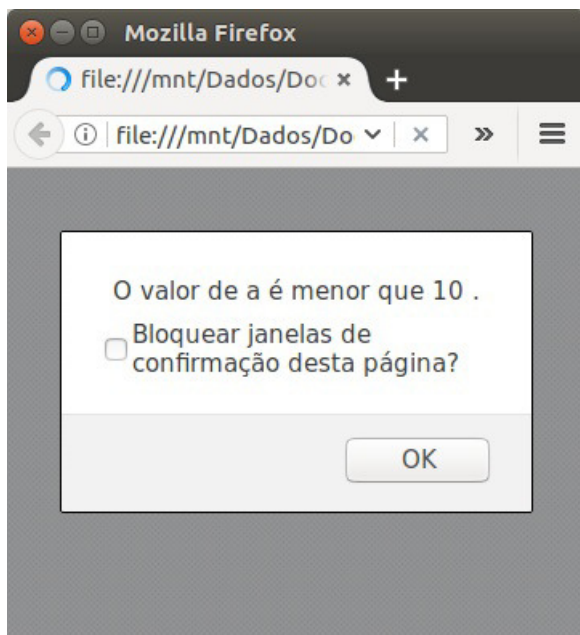


Figura 15.3: Resultado gerado após o usuário informar um número menor que 10.

O `switch/case` pode ser compreendido como uma sequência de `if s` que recebe uma expressão e realiza comparação, executando um caso específico para uma dada declaração:

```
var fruta = prompt("Digite o nome de uma Fruta", "laranja");
switch (fruta) {
    case "laranja":
        alert("Você escolheu laranja.");
        break;
    case "Banana":
        alert("Você escolheu Banana");
        break
    default:
        alert("A fruta que você escolheu não está disponível.")
}
```

Veja que após cada caso foi adicionado o termo `break` . Se ele não for adicionado, ao final será executado o que está abaixo do bloco `default` . Cada `case` representa o equivalente a um `if` ou `else if` ; já a palavra `default` equivale a um `else` .

Para iterações em laço ou loops, podemos usar o `for` , `for/in` , `while` e `do/while` . Para cada uma delas temos:

1. `for` :

```
for(var i=0; i < 10; i++){  
    console.log(i);  
}  
//Saída será de zero a nove:  
//0  
//1  
//2  
//3  
//.  
//.  
//.  
//9
```

No caso anterior, definimos o comando `for` como se fosse uma função. Dentro dos parênteses, criamos a variável inteira com nome `i` , e em seguida definimos a operação `i<10` . Isso indica que queremos que o processo dentro do bloco do `for` se repita enquanto o valor de `i` for menor que 10. O último termo dentro dos parênteses é `i++` , o qual indica que queremos que seja somado uma unidade ao valor de `i` ao final de cada execução do bloco demarcado pelo `for` .

Veja que cada parte do comando escrito dentro dos parênteses do `for` foi separado por `;` . Dessa forma podemos traduzir o comando `for(var i=0; i<10; i++)` como: *Para i iniciando com valor igual a zero e enquanto i for menor que 10, faça o que*

está dentro do bloco e, ao final, adicione uma unidade inteira ao valor de i .

1. `for/in` . Não deve ser usado sobre Arrays, pois ele interage de forma arbitrária:

```
var carros = {a:"fiat", b:"vw", c:"honda"};
for(var prop in carros){
    console.log("O carro é: " + carros[prop]);
}
//Saída:
//O carro é: fiat
//O carro é: vw
//O carro é: honda
```

Esse comando captura as propriedades do objeto `carros` , permitindo-nos fazer operações sobre os valores contidos em cada item. No caso anterior, criamos o objeto `carros` com as propriedades `a` , `b` e `c` . Para retornar o valor contido em cada propriedade, usamos o comando `for(var prop in carros)` . Note que dentro do bloco capturamos o valor armazenado na propriedade usando o comando `carros[prop]` . Nesse caso, podemos traduzir o comando `for(var prop in carros)` como: *Para cada propriedade contida em `carros` crie a variável `prop` que a armazene, em seguida faça a operação contida dentro do bloco.*

1. `while` e `do/while` :

```
var a = 0;
while(a < 10){
    console.log(a);
    a++;
}
```

O `while` continuará a executar o que está dentro do bloco enquanto a condição que ele estiver analisando for verdadeira. No

exemplo anterior, criamos a variável `a` com valor igual a zero. Em seguida, dissemos que, enquanto o valor de `a` for menor que 10, realize a operação dentro do bloco. Em seguida, pedimos para que o programa exiba no console o valor de `a`, e, ao final, que o valor de `a` seja acrescido de uma unidade.

Note que se em vez de ter a condição `a<10` tivéssemos escrito a palavra `true`, o `while` sempre seria executado, criando um loop infinito. O comando `while(a<10)` pode ser interpretado como: *Enquanto a condição analisada for verdadeira, que no caso é verificar se o valor de `a` é menor que 10, execute o que está dentro do bloco.*

A seguir temos um exemplo do `do/while`:

```
var a = 0;
do{
    console.log(a);
    a++;
}while(a < 10)
```

A grande diferença entre o `while` e o `do/while` é que no segundo caso o código do bloco é executado primeiro e só em seguida é que o comando analisa se o valor armazenado em `a` é menor que 10. Para o caso do `do/while` temos: *Faça a operação contida dentro do bloco, em seguida verifique se a condição é verdadeira, que no caso corrente é ver se o valor de `a` é menor que 10.*

15.9 FUNÇÕES

As funções são blocos de códigos que podem ser reutilizados. São declaradas com a palavra `function`:

```
function adicionaDoisNumeros(x, y){  
    return x + y;  
}  
console.log(adicionaDoisNumeros(2,3));  
//Irá retornar 5
```

No código anterior, criamos a função `adicionaDoisNumeros` que recebe as variáveis `x` e `y`. Ao final, a função retornará o valor da soma das duas variáveis.

15.10 CONCLUSÃO

Neste capítulo fizemos uma introdução ao JavaScript e seus principais elementos como estrutura de dados, chamada de funções e *JavaScript Object Notation* - JSON. Esses são elementos fundamentais para quem precisa trabalhar no desenvolvimento *front-end* de aplicações Web.