ALGORITMOS E PROGRAMAÇÃO

Mark Joselli

IESDE
Ad Maiora Semper!

Algoritmos e Programação

Mark Joselli

IESDE BRASIL S/A
2017

CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

J72a Joselli, Mark

Algoritmos e programação / Mark Joselli. - 1. ed. - Curitiba, PR : IESDE Brasil, 2017.
140 p. : il. ; 21 cm.
Inclui bibliografia
ISBN 978-85-387-6349-9

1. Algorítimos. 2. Matemática (Programação). I. Título.

17-45178

CDD: 519.92
CDU: 519

Capa: IESDE BRASIL S/A.
Imagem da capa: tj-rabbit/iStockphoto

IESDE BRASIL S/A.
Al. Dr. Carlos de Carvalho, 1.482. CEP: 80730-200
Batel – Curitiba – PR
0800 708 88 88 – www.iesde.com.br

Computadores evoluíram e hoje em dia fazem parte do dia a dia das pessoas. Celulares, computadores, relógios, carros e até eletrodomésticos possuem programas e, para criar programas, é necessário dominar a linguagem de programação.

Esta obra pretende introduzir uma linguagem de programação específica, o Python, com o objetivo de auxiliá-lo a desenvolver seus próprios programas. São abordados os conceitos básicos dessa linguagem, de modo que você possa utilizar o mesmo conceito apresentado em outras linguagens de programação.

No primeiro capítulo da obra, apresentamos os conceitos básicos dos computadores para explicar como os softwares funcionam. Dessa forma, esperamos que você consiga entender a importância da programação. Além disso, esse capítulo explica como você deve configurar seu computador para desenvolver os programas que serão realizados nos capítulos posteriores.

No capítulo seguinte, introduzimos um conceito básico de linguagem de programação, a variável, que é um espaço de memória onde podemos guardar informação. Nesse mesmo capítulo observamos que, com as variáveis, também são realizadas operações.

No terceiro capítulo, o conceito de condições é apresentado. Os programas executam códigos capazes de realizar decisões e modificar sua execução de acordo com as condições em que o programa se encontra.

No quarto capítulo é trabalhado o conceito de laço de repetição, ou em inglês loops, nos quais partes de código podem ser copiados, possibilitando a realização de tarefas repetidas.

Uma das características importantes dos programas é interagir com os usuários. Nesse sentido, o quinto capítulo explica como você pode mostrar mensagens e receber entrada do usuário.

No sexto capítulo as listas são apresentadas. Elas têm um funcionamento parecido com as listas na vida real, nas quais há diversos elementos em sequência.

Uma das vantagens de programas e código é você poder reutilizá-los com o uso de funções, como mostrado no sétimo capítulo.

No último capítulo, você é convidado a desenvolver um programa do começo ao fim.

Mostramos, assim, os princípios básicos da programação para que você possa desenvolver programas simples.

Desejamos uma boa leitura!

Mark Joselli

Doutor e mestre em Computação pela Universidade Federal Fluminense (UFF). Especialista em Gerência de Projetos pela Pontifícia Universidade Católica do Paraná (PUCPR) e em Psicologia Transpessoal pela Unipaz Paraná. Bacharel em Engenharia elétrica pelo Centro Federal de Educação Tecnológica do Rio de Janeiro (Cefet-RJ) e em Filosofia pela Universidade do Sul de Santa Catarina (Unisul). É professor, pesquisador, consultor e desenvolvedor de softwares e games.

Sumário

1 Introdução à programação

Este capítulo é uma introdução a algoritmos e programação, e serão abordados os princípios que embasam esse conteúdo. Para acompanhar não é necessário ter um conhecimento prévio de programação ou ser um *hacker* de computadores. Parte-se do preceito que o leitor sabe somente como interagir com o computador, ligar, desligar, instalar e utilizar programas e acessar a internet. Sendo assim, pretende-se com este capítulo estimular a curiosidade por esta disciplina, introduzindo os conceitos básicos, de forma que o leitor entenda a importância da programação e da linguagem de programação Python. Além disso, pretende-se preparar o ambiente, instalando o programa que utilizaremos chamado PyCharm.

1.1 Conceitos básicos de programação

Vídeo

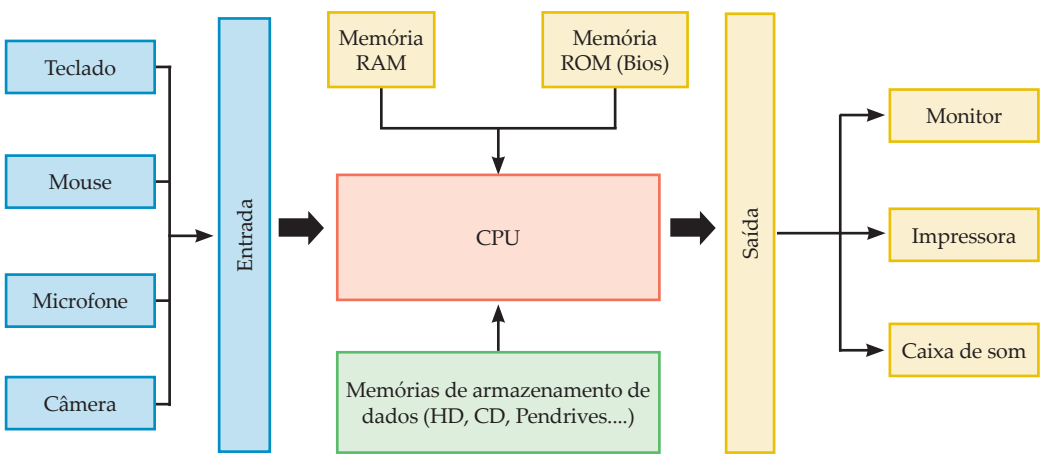
Esta seção tem como finalidade apresentar alguns conceitos básicos, como o hardware e software que formam os computadores, conceitos básicos de programação, e como são organizadas as linguagens de programação.

Computadores têm uma grande importância no mundo atual, e estão integrados a tudo, incluindo hardware e software. Os hardwares são os componentes físicos, como placa-mãe, monitor, teclado e mouse. Já os softwares consistem no sistema operacional e nos programas que rodam nesse sistema, como o iTunes, Office e Firefox. O hardware e software se complementam e formam o que chamamos de computador.

1.1.1 Hardware e software

O hardware consiste em três componentes principais: a unidade central de processamento (ALU, que é conhecido normalmente pela sigla em inglês CPU), a memória e os dispositivos. A CPU fica responsável por realizar as instruções dos softwares realizando cálculos aritméticos, lógicos, de controle ou operações de entrada e saída, de acordo com a instrução que está sendo processada. A memória é onde os dados são guardados, sempre usando a unidade básica de bits. Ela pode ser do tipo RAM, que são as memórias voláteis, isto é, necessitam de energia para manter a informação armazenada; ou também pode ser do tipo ROM/Flash (ou outros), que guardam os dados a todo momento, sem necessidade de retroalimentação, como o hard-drive e os pen drives. Esses diferentes tipos de hardware podem ser vistos na Figura 1.

Figura 1 – Ilustração do funcionamento do hardware.



Fonte: Elaborada pelo autor.

No topo do hardware roda um sistema operacional, como o Microsoft Windows, Mac OS ou Linux. Esses sistemas conectam os softwares com o hardware, criando uma interface básica para os softwares ou aplicativos realizarem processamento e controle no hardware. Dessa forma, os sistemas operacionais devem ter os recursos de hardware mapeados e ter

instruções de como usar esses recursos, o que normalmente é feito por drivers que dizem ao sistema como utilizar o hardware. Os softwares rodam em cima do sistema operacional, e devem saber como ele opera, por isso os programas normalmente servem para um sistema operacional, e para manter a multiplataforma (isto é, rodar em diferentes plataformas) eles devem endereçar aspectos específicos de cada sistema operacional.

Um software é escrito a partir de um código-fonte, que contém as instruções de acordo com uma linguagem de programação. Esse código é lido e processado, de forma a se tornar as ações do computador. A criação desse código-fonte é realizada a partir da programação.

1.1.2 Programação

A programação é o desenvolvimento de software por instruções de comando que o hardware deve realizar. As linguagens de programação transformam essas instruções de hardware em uma linguagem mais simples para os programadores. Finalmente os programadores são as pessoas que sabem ler e escrever instruções em alguma linguagem de programação – e esta obra é o primeiro passo para ajudar o leitor a se tornar um programador.

Todas as linguagens usam instruções como base. Essas instruções são seguidas literalmente pelos computadores. Dessa forma, se metaforicamente mandarmos o computador pular, teríamos que definir para ele diversas variáveis, como a forma de realizar o pulo, a qual altura ele deve pular, qual o impulso ele deve ter, onde ele deve cair etc.

Como exemplo de instruções, poderíamos colocar alguns exemplos comuns: “faça isso; depois faça aquilo”; “se essa condição for verdadeira: faça isso; caso contrário faça aquilo”; “repita essa ação um número determinado de vezes”; “realize uma ação até eu mandar parar”.

Existem diversas linguagens de programação e a cada ano aparecem mais. Algumas são melhorias para desenvolvimento web, outras para desenvolvimento de aplicativos e outras para desenvolvimento científico. As linguagens podem ser divididas em três tipos gerais:

- Linguagem de máquina: são diretamente entendidas pelos computadores (linguagem binária em 1s e 0s), conforme o código, eles executam as instruções necessárias.
- Linguagem Assembly: elas abstraíram os códigos de máquina em instruções que representam as operações elementares que o computador realiza.
- Linguagens de alto nível: têm uma linguagem mais próxima da linguagem humana, aumentando a performance que um programador leva para escrever um software. As linguagens modernas de programação são desse tipo, sendo Python uma delas.

As linguagens de alto nível podem ser compiladas, transformado a linguagem diretamente em código de máquina, um processo que pode tomar tempo, dependendo do tamanho do código a ser compilado, mas tem a melhor *performance* quando comparado com as outras alternativas. Essas linguagens também podem ser interpretadas, e não têm tempo de compilação, pois o código é interpretado em tempo de execução e em código de máquina, mas podem ter uma *performance* pior (a velocidade de execução) que as linguagens compiladas, pois perde-se tempo de processamento com a interpretação.

Com o desenvolvimento do hardware, a execução em código de máquina ou Assembly não se tornou mais viável, pois as instruções se tornaram grandes, e pela falta de claridade, ininteligíveis. Sendo assim, as linguagens de alto nível começaram com a criação das programações estruturadas, que foram criadas para ser sequências de código claras, corretas e facilmente modificadas. As mais famosas foram o Pascal e o C, que se focam em ações, verbos, e são sequencialmente executadas, de forma a dizer o que o hardware deve realizar.

Mesmo um código criado em programação estruturada torna difícil a manutenção e a reutilização do código depois que a sua arquitetura se torna complexa. Dessa forma, foi criada a linguagem orientada a objetos, que foca em objetos em vez de verbos. Esses objetos podem ser estruturados em bibliotecas, sendo reutilizados em projetos futuros, gerando economia de tempo e esforço. A linguagem Python possui orientação a objeto, mas esta obra abordará somente a parte estrutural dele.

1.2 Python: histórico e importância

Vídeo

Esta obra tem como meta explicar os conceitos de programação. De forma a facilitar o aprendizado, utilizará da linguagem Python de forma a aplicar os conceitos. Mesmo assim, os conceitos têm formas equivalentes nas mais diversas linguagens de programação, logo esta seção apresentará os conceitos básicos de forma a ambientar o leitor na linguagem usada.

Python é uma linguagem de programação que permite a você criar o trabalho rapidamente e integrar sistemas mais eficientemente. Foi criado em 1990 por Guido Van Rossum. O seu nome, apesar de ser similar ao nome de uma cobra em inglês, vem da comédia Monty Python. É utilizado por várias grandes empresas do mundo, como Google, Yahoo, Nasa e RedHat (PYTHON SOFTWARE FOUNDATION, 2017).

O Python é uma linguagem de código aberto, isto é, o código é totalmente disponível para download na internet. Dessa forma, qualquer um pode baixar e modificar o código. Isso faz com que a linguagem tenha uma grande comunidade de contribuidores. Essa comunidade se organiza hierarquicamente e qualquer pessoa pode submeter um erro achado, ou uma tentativa de conserto. Isso faz com que o desenvolvimento, revisão e conserto de eventuais *bugs* (nome dado a um artefato defeituoso no código) sejam realizados de forma distribuída e dinâmica.

O Python tem uma série de vantagens para o desenvolvimento de software, uma das mais marcantes é a facilidade de aprendizado e leitura da linguagem. O Python é considerado uma das linguagens que mais se aproxima da linguagem natural, com regras simples e claras, com os comandos se aproximando bastante do inglês. Isso faz com que tenha um tempo de desenvolvimento e manutenção menor, isto é, o tempo que o programador necessitará para codificar um programa e dar manutenção (consertar *bugs* e aprimorar).

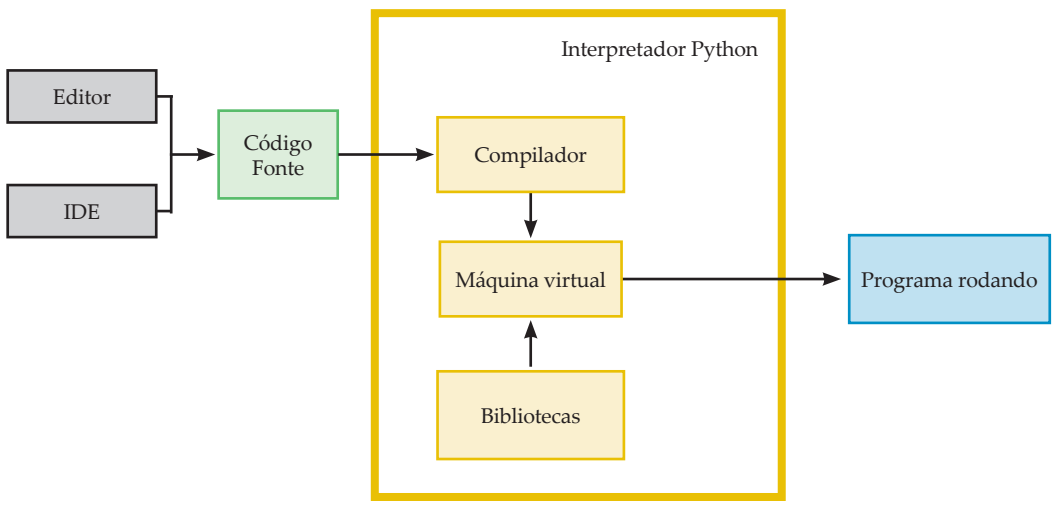
Uma das características do Python é que, apesar de ser simples o suficiente para programadores iniciantes aprenderem a programar pela linguagem, também é poderoso o suficiente para atrair programadores avançados. Dessa forma, o leitor pode continuar a se aprimorar na linguagem, tornando-se um programador avançado.

O Python é extensível por módulos, criando bibliotecas e *frameworks*, acrescentando novas funcionalidades à linguagem original. Esses módulos podem ser desenvolvidos por qualquer programador Python, estendendo a linguagem e provendo novas funcionalidades. Além disso, ele tem diversas bibliotecas e *frameworks*, o que possibilita que o programador possa usá-la desde desenvolvimento de servidores, como *e-commerce*, até mesmo aplicações interativas, como *games*.

1.2.1 Estrutura de execução de um código Python

O código Python é muitas vezes chamado de script, pois ele é interpretado, não necessitando ser compilado anteriormente. Dessa forma, os arquivos .py (extensão normalmente usada pelos arquivos Python) podem ser criados por editores de texto (como o bloco de notas do Windows). Esses arquivos contêm instruções na linguagem Python e são chamados do código-fonte da sua aplicação. Quando o interpretador Python lê esses arquivos, realiza uma compilação do código, transformando em código a ser usado pela máquina virtual do Python. Essa máquina virtual faz o processo de transformar o código em algo que o sistema operacional e consequentemente o hardware entenda, e execute seu programa de acordo com as instruções que estão no código-fonte. Esse fluxo de execução pode ser visto na imagem abaixo.

Figura 2 – Ilustração da execução do Python.



Fonte: Elaborada pelo autor.

1.3 Preparando o ambiente

Vídeo

Para o desenvolvimento de nossos códigos Python, precisamos de um ambiente que contenha o interpretador e máquina virtual Python. Dessa forma, esta seção se concentra em mostrar o ambiente utilizado.

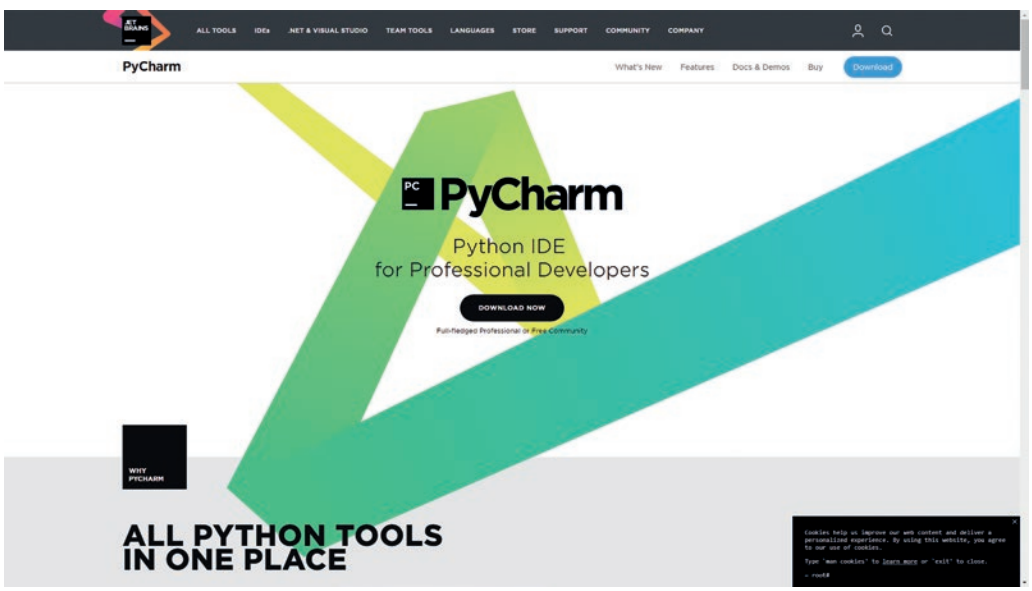
Uma das formas de se utilizar o Python é ir diretamente ao *site* <http://www.python.org>, criar os códigos a partir do shell ou de um arquivo texto, e executá-los diretamente do console. Uma outra forma é utilizando um ambiente de desenvolvimento, chamado PyCharm, que possui diversas facilidades para o desenvolvimento.

1.3.1 Instalando a IDE PyCharm

A PyCharm é uma IDE (*Integrated Development Environment*), ou em português ambiente integrado de desenvolvimento, criada pela empresa JetBrains, famosa pelo desenvolvimento de IDEs que facilitam a programação. Uma IDE oferece um ambiente completo para o desenvolvimento, provendo autocompletar, mostrando erros no código e permitindo inspeção de código, além de refatoramento e debuging (conceitos mais avançados, que serão abordados futuramente).

Para utilizarmos essa IDE, iremos abrir nosso navegador de internet na página: <https://www.jetbrains.com/pycharm>, nesta página o leitor deve clicar sobre o ícone "*download now*".

Figura 3 – Download.

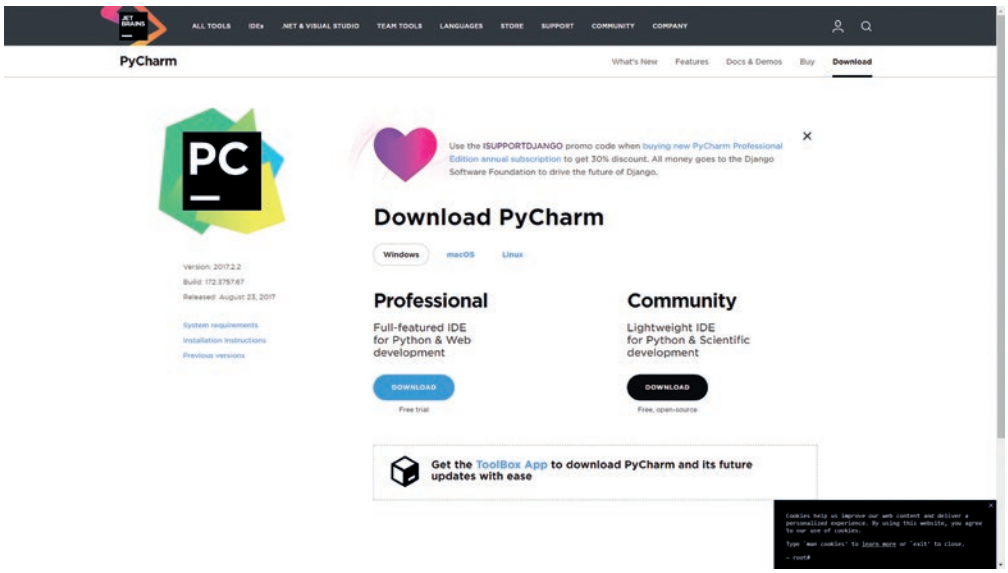


Fonte: JET BRAINS SRO, 2017.

O PyCharm possui versão nos três principais sistemas operacionais, então é necessário baixar na versão do sistema operacional que você utiliza. Além disso, o PyCharm possui

uma versão profissional, que é paga (mas pode se conseguir uma licença de estudante que tem custo zero), ou a Community, que é gratuita. A Professional tem alguns recursos a mais que não serão utilizados nesta obra. Dessa forma será escolhida a versão Community, como mostramos a seguir.

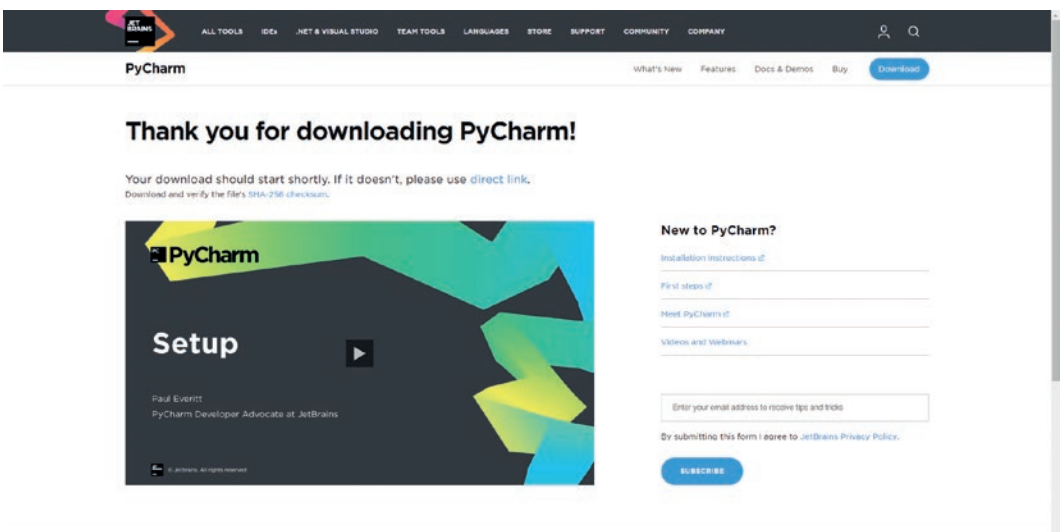
Figura 4 – Versão Community.



Fonte: JET BRAINS SRO, 2017.

E finalmente aparece a tela a seguir, que confirma o começo do download.

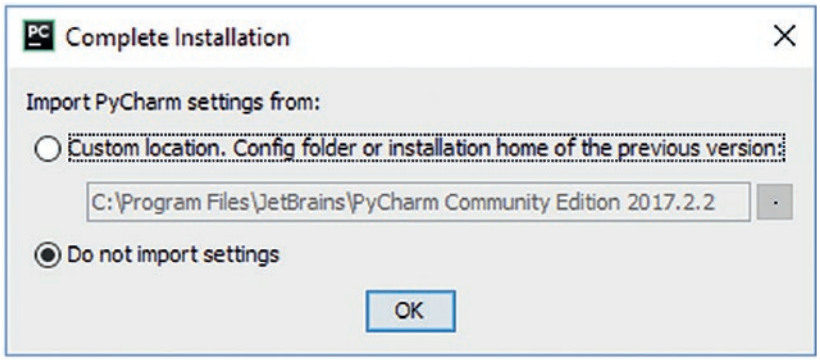
Figura 5 – Fazendo download.



Fonte: JET BRAINS SRO, 2017.

Tendo feito o download, clique duas vezes no executável e proceda normalmente com a instalação. Em seguida, o programa pode ser aberto, tendo a seguinte tela:

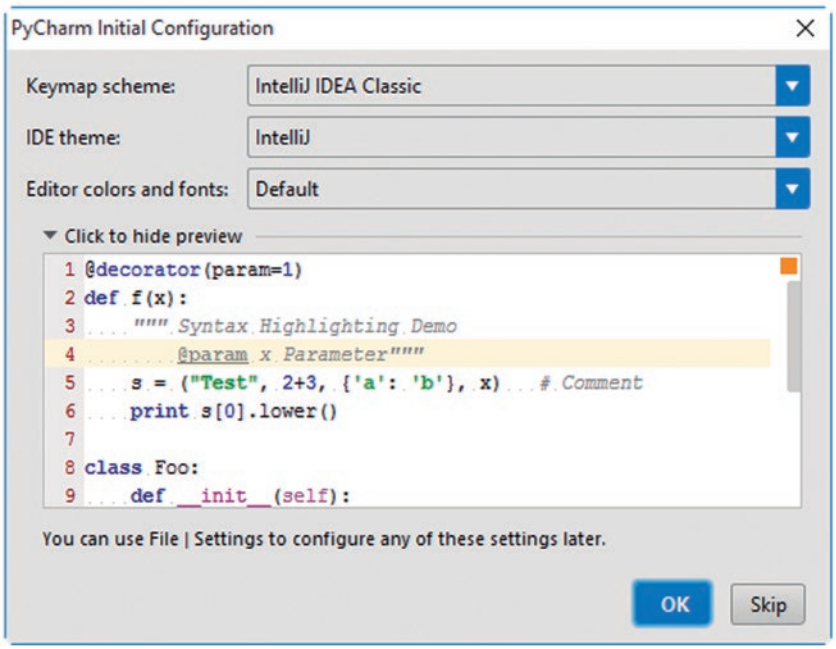
Figura 6 – Instalação.



Fonte: Elaborada pelo autor.

Essa tela configura se o PyCharm deve importar as configurações de alguma versão anterior do programa. No caso de essa ser a primeira vez que se instala o programa, deve-se escolher de não importar nenhuma configuração. Clicando no ok, aparece a seguinte tela:

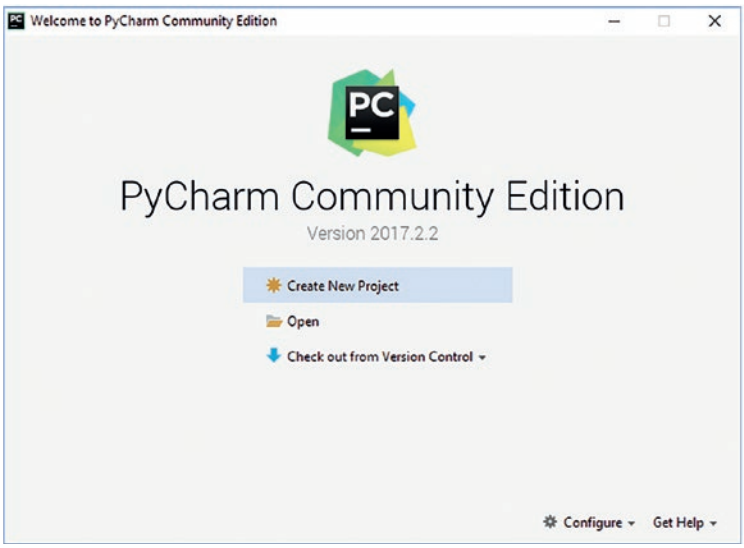
Figura 7 – Tela inicial.



Fonte: Elaborada pelo autor.

Essa tela configura o estilo das teclas de atalho, o tema em que o PyCharm será mostrado e como serão as fontes e cores no aplicativo. Sendo assim, iremos para a tela de boas-vindas, onde podemos criar os projetos, abrir projetos antigos ou configurar nossa IDE.

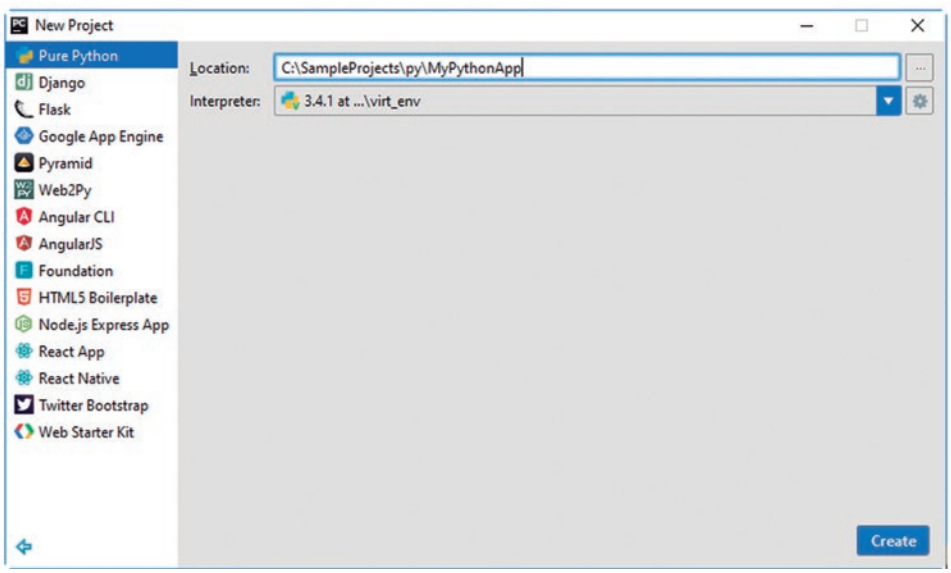
Figura 8 – Tela inicial.



Fonte: Elaborada pelo autor.

Clicando na tela mostrada aparece a opção "*Create New Project*", que irá para uma tela onde seleciona o tipo de projeto, "Pure Python", ou Python puro, um projeto que se utilizará de Python. Nessa tela também deve ser salvo o projeto e qual a versão do Python que será usada (escolheremos a 3.6.1).

Figura 9 – Pure Python.

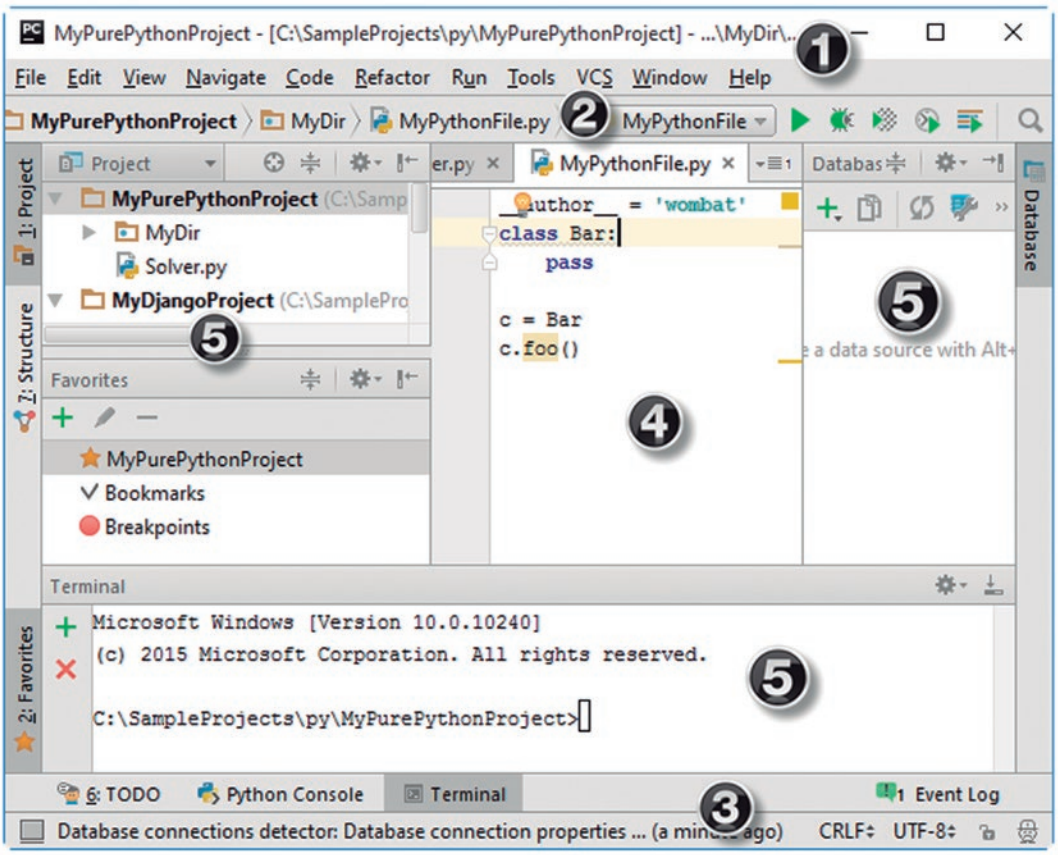


Fonte: Elaborada pelo autor.

Indo para a tela do projeto, mostrada a seguir, em que se pode enviar comandos pelo menu superior (mostrado como 1 na figura); mudar entre os diferentes arquivos na barra de navegação (mostrado como 2 na figura). Na barra de estado podemos visualizar mensagens de informações relevantes (mostrado na figura como 3); o editor, em que cria-se o código-fonte do nosso programa, com ajuda do editor que faz autocompletar e colore a sintaxe das

instruções do código, facilitando o desenvolvimento (mostrado na figura como 4); e finalmente as telas secundárias, que têm ferramentas interessantes, sendo o console utilizado para visualizar o resultado dos códigos desenvolvidos.

Figura 10 – Tela do projeto.



Fonte: Elaborada pelo autor.

Outra opção seria instalar o programa Python, que interpreta os códigos e roda em tela de console. Para criar os códigos, o leitor deverá usar um editor de texto, tipo o bloco de notas, e mandar executar no console. Nesse caso o editor não contará com recursos de autocompletar, cores nas instruções e outros recursos que a IDE provém.

Para instalar essa opção basta abrir o navegador no endereço <https://www.python.org/downloads/windows/> para a versão Windows, escolher a versão 3.6.1 e fazer o download do Windows x86 Executable Installer.

Para o Mac OS e para o Linux, normalmente o Python vem instalado, mas em uma versão anterior a que será usada nos exemplos, sendo necessário baixar a nova, 3.6.1. Para a versão Mac OS é necessário baixar o PKG no endereço: <https://www.python.org/downloads/mac-osx/>, e para Linux é necessário baixar um novo pacote, que na distribuição Linux Ubuntu é feito pelo comando "sudo apt-get install python3".

Para testar se sua instalação foi feita com sucesso, deve-se abrir o terminal/console e digitar o comando "python -V", que mostra qual a versão do Python que está instalada.

Ampliando seus conhecimentos

Programação e algoritmos: a base da computação

(SANTOS; COSTA, 2002, p. 3-4)

[…]

Ao contrário do que se apregoava há alguns anos, a atividade de programação deixou de ser uma arte para se tornar uma ciência, envolvendo um conjunto de princípios, técnicas e formalismos que visam à produção de produtos de software bem estruturados e confiáveis. Cite-se, dentre estes, os princípios da abstração e do encapsulamento e as técnicas de modularização e de programação estruturada.

Portanto, o estudo de programação não se restringe ao estudo de linguagens de programação. As linguagens de programação constituem-se em uma ferramenta de concretização de produtos de *software*, que representa o resultado da aplicação de uma série de conhecimentos que transformam a especificação da solução de um problema em um programa de computador que efetivamente resolve aquele problema. Deve ser dada ênfase aos aspectos funcionais e estruturais das linguagens, em detrimento aos detalhes de sintaxe. O estudo de linguagens deve ser precedido do estudo dos principais paradigmas de programação, notadamente a programação imperativa, a funcional, a baseada em lógica e a orientada a objetos.

O desenvolvimento de algoritmos e o estudo de estruturas de dados devem receber especial atenção na abordagem do tema programação. Igualmente, deve ser dada ênfase ao estudo das técnicas de especificação, projeto e validação de programas. Deve- se entender que ensinar programação não é apenas ensinar uma linguagem de programação. Este ensino envolve entender problemas e descrever formas de resolução, de maneira imparcial, para que então sejam codificadas em uma linguagem. Ou seja, somente após o aprendizado dos conceitos de algoritmo e fundamentos de lógica, o estudante pode travar contato com uma linguagem de programação. Além disso, o estudo de programação deve passar efetivamente pelo estudo de estruturas de dados.

[…]

Assim, a programação é, pois, um dos pontos chaves em um curso de Computação, pois é a atividade que supre o computador com meios para

servir ao usuário. Seu uso adequado e o entendimento de conceitos, princípios, teorias e tecnologias podem conduzir à produção de produtos de software de qualidade, objetivo final, tanto quando se tem a computação como um meio como quando a computação é tida como um fim. Com esse intuito, Setúbal (2000) apresenta algumas recomendações genéricas a serem aplicadas no ensino de disciplinas relacionadas a algoritmos:

- **coerência com os objetivos fundamentais.** Por coerência, entende-se que o professor deve: i) expressar claramente as ideias, os conceitos e as técnicas perante os alunos (se o professor coloca algoritmos confusos na lousa ou em transparências, ele não pode esperar algoritmos claros nas respostas dos alunos); ii) destacar a importância dos resultados teóricos e mostrar rigor formal toda vez que isto se fizer necessário; e iii) valorizar o uso de técnicas na resolução de problemas;

- **ênfase no pensamento crítico.** Deve-se ter um cuidado especial, pois os alunos que têm pouca maturidade matemática tendem a acreditar em qualquer demonstração. Tal comportamento deve ser desestimulado. É essencial que os alunos duvidem daquilo que é apresentado a eles e é com dúvidas saudáveis e sua resolução que a percepção da importância do resultado teórico poderá ser consolidada. Nesse sentido, considera- se um recurso valioso o conjunto de exercícios que pedem para os alunos identificarem falhas de argumentação, erros em algoritmos ou erros em notícias da imprensa;

- **a teoria na prática.** A experiência mostra que os alunos em geral não se sentem atraídos, por considerarem-na muito abstrata. Por esse motivo, crê-se ser importante usar como recurso didático exemplos reais. A inclusão de projetos de implementação, em disciplinas teóricas ou em disciplinas específicas, torna a matéria menos abstrata. De resto, é importante salientar para os alunos o grande impacto que os resultados teóricos têm alcançado na prática.

Atividades

1. Responda verdadeiro ou falso, e justifique as alternativas falsas:

( ) Hardware são instruções que comandam a CPU a realizar tarefas.

( ) A memoria ROM serve para guardar informação e é volátil.

( ) Uma linguagem de alto nível deve ser sempre compilada para linguagem Assembly.

( ) Python foi batizada utilizando o nome de um programa de TV.

2. Quais são os três componentes principais de um computador?

3. Diga com suas palavras o que é uma IDE.

4. O que é código-fonte?

Referências

JET BRAINS SRO. **Getting started**. Disponível em: <https://www.jetbrains.com/pycharm/documentation/>. Acesso em: 4 jun. 2017.

PYTHON SOFTWARE FOUNDATION. **Documentation**. Disponível em: <http://www.python.org>. Acesso em: 4 jun. 2017.

SANTOS, R. P; COSTA, H. A. X. **TBC-AED e TBC-AED/Web: Um desafio no ensino de algoritmos, estruturas de dados e programação**. IV Workshop em educação em computação e informática do estado de Minas Gerais. 2005. Disponível em: <http://cos.ufrj.br/~rps/pub/completos/2005/WEIMIG.pdf>. Acesso em: 9 jul. 2017.

Resolução

1.

F – Software são instruções que comandam a CPU a realizar tarefas. Hardware são os dispositivos físicos que em conjunto com o software formam o computador.

F – A memória ROM serve para guardar informação, mas ela não é volátil. A memória volátil é a RAM.

F – Uma linguagem de alto nível pode ser compilada para a linguagem de máquina, mas ela também pode ser interpretada.

V

2. Hardware e software.

3. Uma IDE é um ambiente integrado de desenvolvimento em que é possível se codificar novos softwares.

4. Código-fonte é um conjunto de instruções, que se executadas pelo computador e fazem com que ele realize as ações especificadas nas instruções.

2
Criando meu primeiro programa

Neste capítulo pretende-se introduzir conhecimentos básicos de qualquer linguagem de programação. Dessa forma, irá abordar inicialmente como realizar a construção do primeiro programa que o leitor irá se deparar em praticamente qualquer linguagem de programação, o "Olá Mundo". Com isso, este capítulo irá apresentar e discutir como criar este primeiro programa usando a IDE PyCharm. Partindo desse conhecimento, o capítulo continuará introduzindo importantes aspectos fundamentais da programação, como as variáveis e os operadores. Variáveis seriam um local na memória do computador que guarda alguma informação, e os operadores são símbolos que representam transformações nas variáveis, como cálculos, atribuições, ordem e relações.

Vídeo

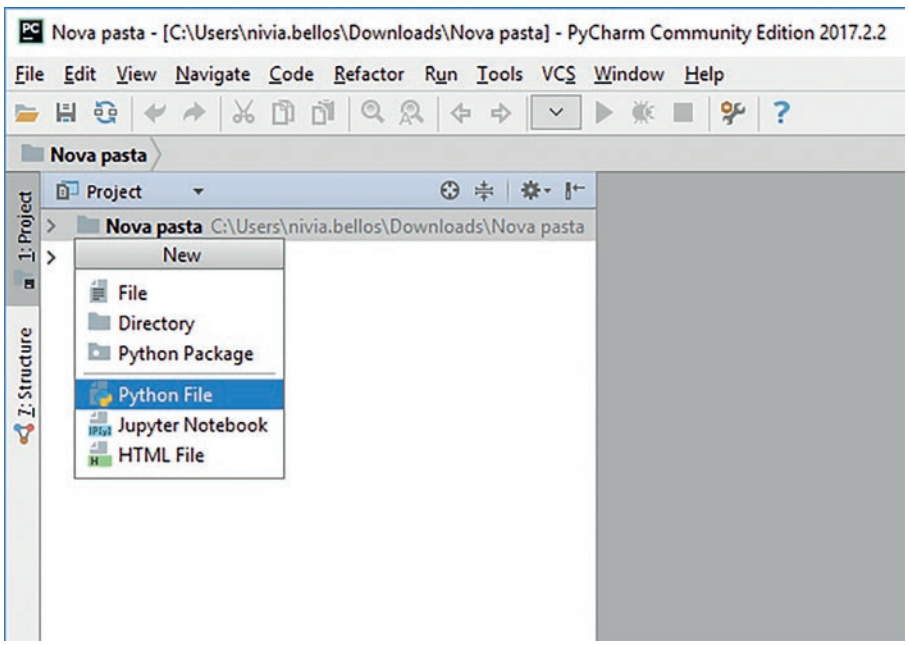
2.1 O primeiro programa: Olá Mundo

Esta seção tem como finalidade apresentar ao leitor como criar o seu primeiro programa com a IDE PyCharm, aprender a executá-lo e verificar os resultados.

2.1.1 Criando o primeiro programa

Para começar, indicamos ao leitor clicar com o botão direito do mouse em cima do arquivo do seu projeto (na área direita da tela, em cima da pasta do seu projeto criada como indicado na Figura 1), de forma a criar o arquivo do código-fonte de nosso primeiro programa Python.

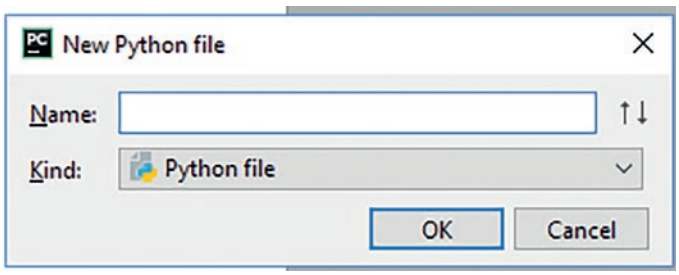
Figura 1 – Criando um arquivo de código-fonte Python no PyCharm.



Fonte: Elaborada pelo autor.

Tendo clicado para criar esse novo arquivo Python, a IDE irá mostrar um diálogo para inserção do nome do arquivo (que será chamado nesse exemplo de "olaMundo", mas o leitor pode colocar o nome que achar melhor). Essa janela pode ser vista na Figura 2.

Figura 2 – Diálogo para criação de um novo arquivo Python.



Fonte: Elaborada pelo autor.

Com esse arquivo criado, o leitor pode digitar o código que irá realizar a impressão de uma linha de texto (com o texto "Olá Mundo"). Esse código está presente no Código 1, e deve ser colocado na tela do editor da IDE.

Código 1 – Código do primeiro programa.

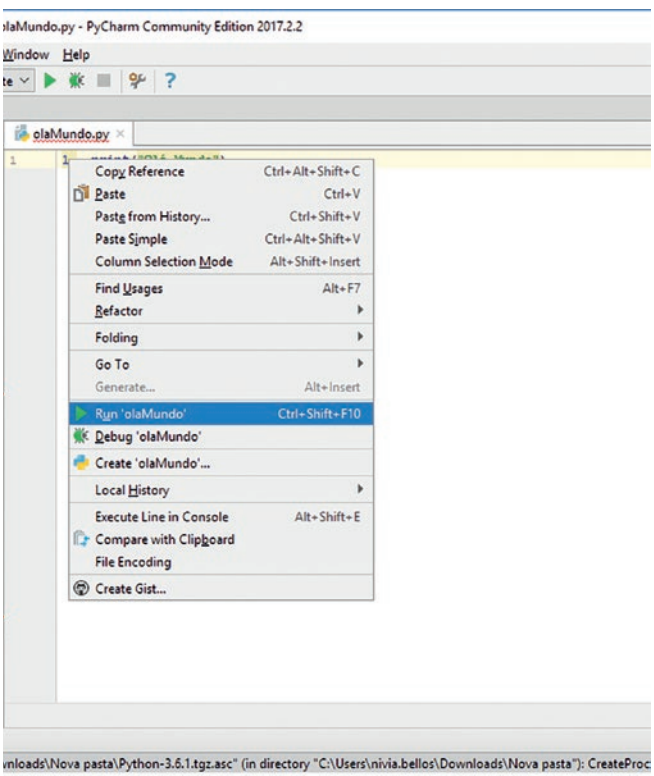
```
1. print("Olá Mundo")
```

Fonte: Elaborado pelo autor.

Esse código possui um comando (a função **print**) que instrui ao computador a mostrar uma **string** (o nome dado para uma sequência de caracteres formando um texto). Uma **string** sempre deve ser colocada dentro de aspas duplas ("") e tem valor ("Olá Mundo"). Cada linha em Python geralmente possui uma instrução, e ela acaba quando a linha em que ela foi escrita acaba, sendo a linha seguinte uma nova instrução.

Tendo feito isso, deve-se mandar compilar e executar esse código, clicando com o botão direito do mouse e selecionando a opção "Run 'oláMundo'" como mostra a Figura 3.

Figura 3 – Rodando o código olaMundo.py.



Fonte: Elaborada pelo autor.

Tendo feito isso, aparecerá o resultado da execução no console da IDE, que na primeira linha sempre mostra o caminho completo para o arquivo que está sendo executado, e na última linha uma instrução dizendo que o processo foi terminado com o código 0 (normalmente os processos dão um código de retorno, e 0 quer dizer que ele foi rodado sem problemas). Entre esse começo e fim, o leitor poderá verificar o resultado do seu programa, que nesse caso é uma **string** com o texto <Olá Mundo>.

Código 2 – Resultado do programa.

```
1. /Library/Frameworks/Python.framwork/Version/3.6/bin/python3.6 "/Users/BEPID/
Dropbox (DEPiD)/iesde/algoritmos 1/cap2/code/olaMundo.py"
2. Olá Mundo
3. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

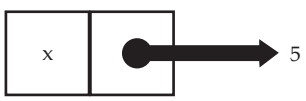
Vídeo

2.2 Variáveis

Tendo aprendido sobre como construir um simples programa, esta seção tem como foco explicar um dos fundamentos mais básicos da programação: a variável, um nome que representa um dado que está guardado na memória. Em Python, o nome das variáveis também pode ser chamado de *identificador*. O programador pode usar caracteres alfabéticos e dígitos (exceto para o primeiro caractere). Sendo assim o nome <Casa1> é valido, mas <1Casa> não. Alguns nomes também são reservados em Python, pois eles servem para comandos específicos da linguagem.

As variáveis podem ser imaginadas como pequenas caixas onde o programa pode guardar informação. Cada variável guarda somente uma informação, mesmo sendo guardada na linguagem de máquina como 1s e 0s na memória. Isso facilita a vida dos programadores, e mantém a informação segura. Cada nome de variável fica em uma tabela de ligação, que coloca o nome e o local na memória onde está o valor da variável. Uma ilustração da memória pode ser vista na Figura 4, que mostra o nome da variável (x) apontando para a região de memória com o valor que está guardado na variável (5).

Figura 4 – Exemplo de uma variável de nome (x) apontando para uma região de memória que contém o valor (5).



Fonte: Elaborada pelo autor.

Para guardar uma informação em uma variável, o Python utiliza o operador de atribuição igual ("="), como mostra o Código 3, em que o valor 5 é guardado na variável de nome x. Muitas vezes em programação usa-se o termo *receber* em vez de *guardar*.

Código 3 – Declaração de uma variável x.

```
1. x = 5
```

Fonte: Elaborado pelo autor.

Esse código manda a instrução para o computador guardar o valor 5 em uma região de memória que será acessada usando o identificador x. Diz-se que a linha tem uma instrução que declara uma variável x e coloca o valor 5 nela. Um dos detalhes de Python, ao contrário de outras linguagens, como C++ e Java, é que não se pode declarar uma variável sem colocar um valor nela. Outro detalhe da linguagem Python é que não é necessário especificar o tipo, pois a linguagem descobre o tipo da variável de acordo com o valor a ser colocado nela.

Como pode-se ver, o Python na declaração de variável utiliza da lógica:

Nome_da_variavel = dado

2.2.1 Tipos de dados

Existem diversos tipos de dados em Python, e como visto anteriormente esse tipo de dado é guardado em um tipo específico de memória, pois tipos diferentes requerem espaços diferentes de memória.

2.2.1.1 Tipos número inteiros

No exemplo anterior, em que se colocou o valor 5 para x, temos que a variável x é do tipo numérico, mais especificamente inteiro (ou ainda mais especificamente **int**), pois seu conteúdo é um número inteiro. Os tipos **int** podem ser positivo ou negativo, isso quer dizer que os números inteiros têm sinal. Na maioria das plataformas, as variáveis do tipo **int** podem ter valor mínimo de -9.223.372.036.854.775.808 e valor máximo de 9.223.372.036,854.775.807, que é o valor máximo que pode ser colocado numa região de memória de 64-bits (esses valores podem mudar com o sistema operacional ou com a arquitetura do computador).

2.2.1.2 Tipos número com ponto flutuante

Além disto, o Python tem outros tipos numéricos, como o **float**, que guarda um número com ponto flutuante (se utilizará a notação americana e das linguagens de programação, que utiliza o ponto (.) em vez da vírgula para indicar o ponto flutuante), ou seja, guarda também sua parte decimal, como 1.0.

A diferença entre os números do tipo **int** e os números do tipo **float** são que este tem a parte flutuante. Além disso, o **float** pode guardar números imensamente pequenos e imensamente grandes. Mas assim como o **int**, o **float** também tem um limite a essa capacidade. No caso, o maior número que pode ser guardado em um **float** é $\pm 1.7976931348623157 \times 10^{308}$ e o menor número guardado no **float** é $\pm 2.2250738585072014 \times 10^{-308}$. Um exemplo de código onde se guarda um número flutuante (3.5) em uma variável (numero_float) pode ser visto no Código 4.

Código 4 – Declaração de uma variável float.

```
1. numero_float = 3.5
```

Fonte: Elaborado pelo autor.

2.2.1.3 Tipo número complexo

Um outro tipo de número que pode ser usado em Python é o tipo número complexo, ou **complex** como é colocado em Python. Esse tipo de número tem uma parte real e outra parte imaginária e é bastante usado em cálculos científicos. Um exemplo de código em que se guarda um número complexo (4 + 3j) em uma variável (**numero_complex**) pode ser visto no Código 5.

Código 5 – Declaração de uma variável complex.

```
1. numero_complex = 4 + 3j
```

Fonte: Elaborado pelo autor.

2.2.1.4 Tipo **booleano**

Um tipo mais simples, mas bastante importante, é o **booleano** (**bool**). Nele, a variável pode ter somente dois valores: verdadeiro (**True**) e falso (**False**). Ele também pode ser entendido como verdadeiro sendo o valor 1 e 0 quando o valor é **False**. Esse tipo é bastante usado para realizar operações lógicas e relacionais entre as variáveis, como será visto na próxima seção. Um exemplo de código onde se guarda um **bool** (**True**) em uma variável (**booleano**) pode ser visto no Código 6.

Código 6 – Declaração de uma variável bool.

```
1. booleano = True
```

Fonte: Elaborado pelo autor.

2.2.1.5 Tipo **string**

Um outro tipo visto na seção anterior que ainda não foi visto junto com variáveis é a **string**. Uma **string,** como comentado anteriormente, é uma sequência de caracteres dentro das aspas duplas (“”). Um exemplo de código em que se guarda uma **string** (“teste”) em uma variável (texto) pode ser visto no Código 7.

Código 7 – Declaração de uma variável string.

```
1. texto = “teste”
```

Fonte: Elaborado pelo autor.

2.2.2 Conversão entre tipos

Podemos converter entre os tipos. Por exemplo, pode-se converter uma **string** “1”, ou o **float** 1.0, ou o número **booleano True** no número **int** 1 usando o comando **int**, como pode ser visto no Código 8.

Código 8 – Declaração de uma variável string.

```
1. int1 = int(“1”)
2. int1 = int(1.0)
3. int1 = int(True)
```

Fonte: Elaborado pelo autor.

Existe ainda conversão de outros tipos para o tipo **string** usando o comando **str** e de outros tipos para o tipo **float** usando o comando **float**, como pode ser visto no Código 9.

Código 9 – Convertendo entre tipos.

```
1. textoFloat = str(1.2)
2. floatTexto = float(“2.5”)
```

Fonte: Elaborado pelo autor.

Vídeo

2.3 Operadores

Tendo visto as variáveis, podemos utilizar de operadores para realizar, como o nome já diz, operações ou manipulações nas variáveis. Os operadores são a base do controle e gerenciamento das informações dos programas. Para usar um operador é necessário fornecer uma variável ou expressão, que pode ser uma equação ou fórmula.

Foi visto já na seção anterior o operador "=", que realiza uma operação de atribuição de uma variável. O Python possui também diversos tipos de operadores, como os matemáticos, de comparação, de função lógica, de negação, unário de indexação e binários.

2.3.1 Operadores unários

Esses tipos de operadores requerem somente uma variável ou expressão como entrada. Eles podem ser usados para mudar o valor de uma variável. Temos 3 tipos de operadores unários: o ~, que realiza a inversão dos bits da variável (então se ela tinha bit 0 vira 1 e vice-versa); o –, que nega o valor original (então se a variável é positiva se torna negativa e vice-versa); e o +, que não modifica a variável, seria colocado somente para facilitar a visualização que o valor é positivo.

Para facilitar a visualização das operações matemáticas, o leitor pode visualizar no Quadro 1 esses operadores, uma descrição, um exemplo e o resultado do uso.

Quadro 1 – Operadores unários.

Operador	Descrição	Exemplo	Resultado
~	Inverte os bits do valor.	~4	-5
–	Nega o valor original.	-(-4)	4
+	Positiva o numero.	+4	4

Fonte: Elaborado pelo autor.

No Código 10 pode-se visualizar a variável (**meuInt**) recebendo um valor (+3) na linha 1. Na linha 2 a variável (**meuInt**) tem seu valor negado, virando –valor (-3). E na linha 3 a variável é impressa no console. O resultado do programa é -3.

Código 10 – Exemplos de operadores unários.

```
1. meuInt = +3
2. meuInt = -meuInt
3. print(meuInt)
```

Fonte: Elaborado pelo autor.

2.3.2 Operadores aritméticos

Esses operadores são capazes de realizar operações matemáticas nas variáveis ou expressões. O Python nativamente já provê diversas operações matemáticas, como adição, multiplicação, divisão e operações semelhantes. Para operações mais complexas, o Python provê bibliotecas adicionais que podem ser usadas.

Para facilitar a visualização das operações matemáticas, o leitor pode visualizar no Quadro 2 os principais operadores matemáticos, uma descrição, um exemplo e o resultado do uso.

Quadro 2 – Operadores matemáticos.

Operador	Descrição	Exemplo	Resultado
+	Adiciona dois valores.	3+4	7
-	Subtrai do valor da direita o valor da esquerda.	4-3	1
*	Multiplica o valor da direita pelo da esquerda.	3*4	12
**	Calcula o exponencial tendo o valor da direita como base e pelo valor da direita.	7**2	49
/	Divide o valor da direita pelo da esquerda.	7/2	3.5
//	Divide o valor da direita pelo da esquerda e somente retorna o valor inteiro do resultado.	7//2	3
%	Divide o valor da direita pelo da esquerda, e retorna o resto da operação.	7%2	1

Fonte: Elaborado pelo autor.

Os operadores podem ser combinados e pode-se utilizar os operadores parênteses – "(" para começar a expressão e ")" para fechar a expressão – para indicar a precedência das operações que serão realizadas. No caso do Código 11, temos operações de soma, multiplicação e divisão separadas pelos parênteses.

Código 11 – Exemplos de operadores matemáticos.

```
1. ((3+4)*2)/3
```

Fonte: Elaborado pelo autor.

No Código 11, primeiramente o Python irá executar a soma (3+4), depois com o resultado fará a multiplicação (7*2) para finalmente realizar a divisão com o resultado (14/3). Os parênteses facilitam a legibilidade do código e são muito úteis para os operadores relacionais e lógicos que veremos a seguir.

2.3.3 Operadores relacionais

Como o nome diz, os operadores relacionais verificam as relações entre as variáveis ou expressões, comparando-as. Essa comparação retorna um **bool**, sendo verdadeiro (**True**) se a comparação for verdadeira ou retorna sendo falso (**False**) caso a relação seja falsa. Essas

comparações são bastante comuns e importante nas lógicas dos programas de computadores, como será visto em capítulos posteriores.

Para facilitar a visualização das operações relacionais, o leitor pode ver no Quadro 3 os principais operadores, uma descrição, um exemplo de uso e o resultado.

Quadro 3 – Operadores relacionais.

Operador	Descrição	Exemplo	Resultado
==	Verifica se os dois valores são iguais.	3 == 4	**False**
		3 == 3	**True**
!=	Determina se os dois valores são diferentes.	3 != 4	**True**
		3 != 3	**False**
>	Verifica se o valor da direita é maior que o valor da esquerda.	3 > 4	**False**
		4 > 3	**True**
<	Verifica se o valor da direita é menor que o valor da esquerda.	3 < 4	**True**
		4 < 3	**False**
>=	Verifica se o valor da direita é maior ou igual ao valor da esquerda.	3 >= 4	**False**
		3 >= 3	**True**
<=	Verifica se o valor da direita é menor ou igual ao valor da esquerda.	3 <= 4	**True**
		4 <= 3	**False**

Fonte: Elaborado pelo autor.

Os operadores relacionais também podem ser colocados entre chaves, como mostra o Código 12.

Código 12 – Exemplos de operadores relacionais.

```
1. ((3 > 4) == True) != True
```

Fonte: Elaborado pelo autor.

No Código 12, primeiramente o Python irá executar a comparação de "maior que" (3>4), que retorna **False**. Desse resultado fará uma comparação de igualdade (False == True) retornado **False** e finalmente fará uma comparação de desigualdade (False != True), retornando **True** como resultado final.

2.3.4 Operadores lógicos

Os operadores lógicos realizam operações lógicas entre as variáveis ou expressões. Eles pegam as variáveis como valores de **True/False** de forma a determinar se o seu resultado também como **True/False**. Esses operadores formam a matemática **booleana** e são bastante usados em processos de decisão, como será visto no capítulo seguinte.

Para facilitar a visualização das operações lógicas, o leitor pode visualizar no Quadro 4 os principais operadores, uma descrição, exemplo e o resultado de uso.

Quadro 4 – Operadores lógicos.

Operador	Descrição	Exemplo	Resultado
and	Operador **e.** Verifica se os dois valores são **True** (verdadeiros).	**True and True**	**True**
		True and False	**False**
		False and True	**False**
		False and False	**False**
or	Operador **ou.** Verifica se um dos dois valores são **True** (verdadeiros).	**True or True**	**True**
		True or False	**True**
		False or True	**True**
		False or False	**False**
not	Nega o valor.	**not True**	**False**
		not False	**True**

Fonte: Elaborado pelo autor.

Os operadores relacionais também podem ser colocados entre chaves e usados juntamente com os operadores relacionais, como mostra o Código 13.

Código 13 – Exemplos de operadores lógicos combinado com relacionais.

```
1. ((3 > 4) and True) or True
```

Fonte: Elaborado pelo autor.

No Código 13, primeiramente o Python irá executar a comparação de maior que (3>4), retornando **False**. Desse resultado, fará a operação **and** (False and True), retornando **False**. E finalmente realizará a operação **or** (False or True) retornando **True**.

2.3.5 Operadores de atribuição

Os operadores de atribuição mudam os valores das variáveis para os novos valores, como visto no começo do capítulo para o operador "=". Mas o Python possui muitos operadores que combinam a atribuição com um operador matemático.

Para facilitar a visualização das operações de atribuição, o leitor pode ver no Quadro 5 os principais operadores, uma descrição, exemplo de uso e o resultado. Neste quadro será considerado que a variável x tem o valor inicial de 3.

Quadro 5 – Operadores de atribuição.

Operador	Descrição	Exemplo	Resultado
=	Coloca o valor da direita na variável à esquerda.	x = 2	Variável x com valor 2
+=	Soma o valor da direita com o valor da variável da esquerda, e coloca o valor resultante na variável da esquerda.	x += 2	Variável x com valor 5

Operador	Descrição	Exemplo	Resultado
-=	Subtrai do valor da variável à esquerda pelo o valor da direita, e coloca o valor resultante na variável da esquerda.	x -= 2	Variável x com valor 1
*=	Multiplica o valor da direita com o valor da variável da esquerda, e coloca o valor resultante na variável da esquerda.	x *= 2	Variável x com valor 6
/=	Divide o valor da direita com o valor da variável da esquerda, e coloca o valor resultante na variável da esquerda.	x /= 2	Variável x com valor 1.5
%=	Divide o valor da direita com o valor da variável da esquerda, e coloca o resto do valor resultante na variável da esquerda.	x %= 2	Variável x com valor 1
**=	Calcula o exponencial do valor da variável da esquerda elevado ao valor a direita, e coloca o valor resultante na variável da esquerda.	x **=2	Variável x com valor 9
//=	Divide o valor da direita com o valor da variável da esquerda, e coloca a parte inteira do valor resultante na variável da esquerda.	x //= 2	Variável x com valor 1

Fonte: Elaborado pelo autor.

2.3.6 Precedência de operadores

Quando se tem mais de um operador na mesma linha do Python, o interpretador precisa definir qual deles deve realizar primeiro. Inicialmente sempre se verifica os parênteses, depois as operações. Cada operador tem uma certa precedência sobre o outro, mas caso se tenha duas operações com a mesma precedência em sequência, primeiro o Python processa a operação mais à direita.

Para facilitar a visualização da ordem dos precedentes, o leitor pode visualizar no Quadro 6 os precedentes de maior para menor.

Quadro 6 – Precedência de operadores.

Operador	Descrição
**	Exponencial.
~ + -	Operadores unários.
* / % //	Multiplicação, divisão, resto e divisão inteira.
+ -	Adição e subtração.
< <= > >=	Operadores de comparação.
== !=	Operadores de igualdade.

Operador	Descrição
= %= /= //= -= += *= **=	Operadores de atribuição.
not or and	Operadores lógicos.

Fonte: Elaborado pelo autor.

As precedências seguem geralmente as mesmas das operações matemáticas, sendo importante levantar alguma dessas regras:

- Os parênteses mais aninhados são processados primeiros.
- Múltiplas operações com exponencial são aplicadas da direita para a esquerda.
- Múltiplas divisões, multiplicações e resto são aplicadas da esquerda para a direita.
- Múltiplas adições e subtrações são aplicadas da esquerda para a direita.

Ampliando seus conhecimentos

Aprendizagem de algoritmos

(CRISTOVÃO, 2008, p. 31-32)

Um dos problemas mais relatados pela literatura e também facilmente vivenciado por qualquer professor que já tenha alguma experiência no ensino de algoritmos é quando este ensino ocorre sob a falta de recursos computacionais onde o aluno possa receber *feedback* imediato das suas tentativas em resolver os problemas propostos. Ou seja, o aluno necessita de ambientes que possa experimentar, simular, investigar possibilidades e sobretudo errar bastante para que tenha percepção sobre os limites de um algoritmo. A falta de ambientes de teste logo no início da aprendizagem de algoritmo supõe que o estudante já possui um excelente nível de abstração em conjunto com um bom raciocínio hipotético-dedutivo uma vez que ele deve conseguir enxergar os resultados de um programa apenas inferindo sobre o código. É claro que esta abstração deverá ser desenvolvida ao longo de sua trajetória, pois bons programadores são também bons investigadores de erros, e esta formação se faz, por exemplo, com exercícios do tipo "teste de mesa" onde o estudante percorre o programa fazendo o papel do computador.

A atividade de programar possui uma cognição muito rica, pois faz o aluno percorrer o ciclo descrever-executar-refletir-depurar apresentado

há bastante tempo por Papert (1994) e muito conhecido e aplicado pela comunidade acadêmica. Este ciclo é muito útil na formação do profissional, porque o faz desenvolver competências associadas às necessidades atuais do mercado de trabalho como as capacidades de planejar, antecipar e simular resultados entre outros.

Na construção de algoritmos o aluno está ativo, ele age, explora, brinca, faz arte, realiza experimentos, antecipa procedimentos, controla suas ações, tem a oportunidade de realizar trocas continuadas entre os colegas, e coordenar uma variedade de conteúdos e de formas lógicas de acordo com a sua capacidade perante o domínio (FAGUNDES, 1997). Escrever um programa é análogo a escrever um texto, pois o aprendiz realiza tentativas, experimentos até chegar a um consenso e compreender o final do programa ou do texto que está escrito. De fato:

> Existe a crença de que só se pode programar o que se compreende perfeitamente. Essa crença ignora a evidência de que a programação, como qualquer outra forma de escrita, é um processo experimental. Programamos, como redigimos, não porque compreendemos, mas para chegar a compreender. (Joseph Weizenbaum) citado por (Vitale, 1991).

Este processo experimental gera descobertas, e consequentemente a construção de conhecimentos por parte do aprendiz e oportuniza a vivência do ciclo descrever-executar-refletir-depurar.

O domínio da linguagem adotada para se escrever o algoritmo é fundamental para que o autor consiga se expressar de forma correta para solucionar um problema. Tal como enfatiza Delgado (2004) a formalização em linguagem natural facilita o aluno a expor sua solução, a um grupo de colegas. Assim, o grupo é estimulado pelo professor a explorar, questionar e validar a solução apresentada pelo colega. Essa verbalização constrói, sob a intervenção dos componentes da turma, progressiva e interativamente, a formalização considerada satisfatória pelo grupo. O papel do professor nessa etapa é o de minimizar o nível de competitividade e manter o grupo em ação colaborativa e investigativa.

[...]

Atividades

1. Escreva com suas palavras o que é variável.

2. Cite três nomes de variáveis que são aceitas pelo Python.

3. Qual o valor resultante das expressões a seguir:

a. 1 + 3 * 2 – 5

b. 1 // 2 + 5 // 6

c. 1 + 5 – 2**3

d. 2 ** 3 * 4

4. O que será imprimido pela IDE nos códigos a seguir:

a.
```
x = 2
y = 3
x += y
x + y
print(x)
```

b.
```
x = 3
y = 2
x -= 2
print(y)
y = x
```

Referências

CRISTOVÃO, H. M. Aprendizagem de algoritmos num contexto significativo e motivador: um relato de experiência. SBC (2008): 30. **Anais do XXVIII Congresso da SBC**. WEI – Workshop sobre Educação em Computação Unidade de Computação e Sistemas Faculdades Integradas Espírito-santenses (FAESA). Julho 2008. Disponível em: <http://www2.sbc.org.br/csbc2008/pdf/arq0123.pdf>. Acesso em: 6 jul. 2017.

MUELLER, J. P. **Começando a programar em Python para leigos**. Stalin, 2016.

PYTHON SOFTWARE FOUNDATION. **Documentation**. Disponível em: <http://www.python.org>. Acesso em: 4 jun. 2017.

JET BRAINS SRO. **Getting started**. Disponível em: <https://www.jetbrains.com/pycharm/documentation/>. Acesso em: 4 jun. 2017.

☑ Resolução

1. Variável é um nome que representa um dado guardado na memória.

2. Resposta pessoal. Pode ser qualquer nome com caracteres alfabéticos e dígitos (exceto para o primeiro caractere da variável).

3.

a. 1 + 3 * 2 – 5

>>1 + 6 - 5

>>1 + 1

>>2

b. 1 // 2 + 5 // 6

>>0 + 0

>>0

c. 1 + 5 – 2**3

>> 1 + 5 – 8

>> 1 - 3

>>-2

d. 2 ** 3 * 4

>> 8 * 4

>>32

4.

a. x = 2

y = 3

```
x += y

x + y

print(x)

>>5
```

b.
```
x = 3

y = 2

x -= 2

print(y)

y = x

>>2
```

3 Condições

Neste capítulo pretende-se introduzir as condições, recursos essenciais para a construção da maioria dos programas, que faz com que o programa tome decisões em tempo de execução. Dessa forma, o texto irá abordar inicialmente como as condições funcionam e como elas integram a lógica de programação, apresentando a estrutura básica do comando **if**. O capítulo continuará apresentando como usar a combinação desse comando com operadores, principalmente os relacionais. E por fim a combinação de condições com os operadores lógicos, que transformam o comando **if** em um comando poderoso para ajudar o computador a tomar decisões.

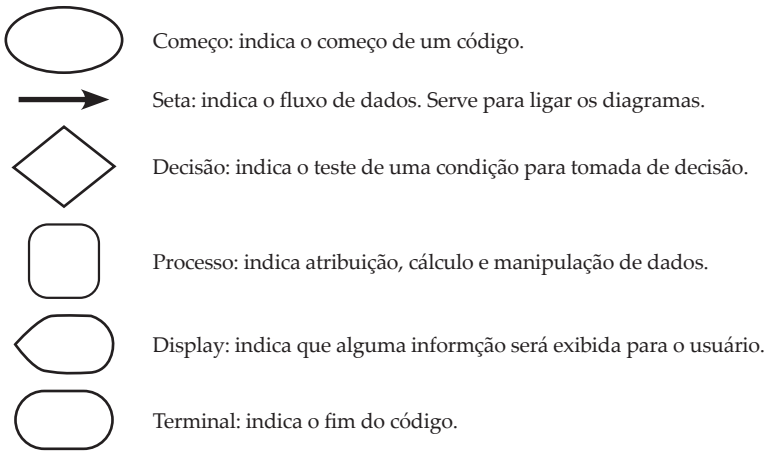
Vídeo

3.1 Introduzindo condições

Esta seção tem como finalidade apresentar ao leitor o conceito de condições e criar a base para o entendimento de códigos mais complexos. O poder da programação não vem de colocar o computador para realizar uma instrução atrás da outra como feito no capítulo passado, mas sim criar um processo decisório, em que expressões são avaliadas e o programa, a partir dessas expressões, pode tomar caminhos diferentes.

Para esta seção vamos usar um esquema de diagrama de fluxos para explicar melhor o funcionamento do processo de decisão. Dessa forma, a sequência de instruções que o leitor visualiza na sua IDE se transforma em uma representação visual de acordo com o símbolo. Os símbolos que serão usados neste capítulo são mostrados na Figura 1.

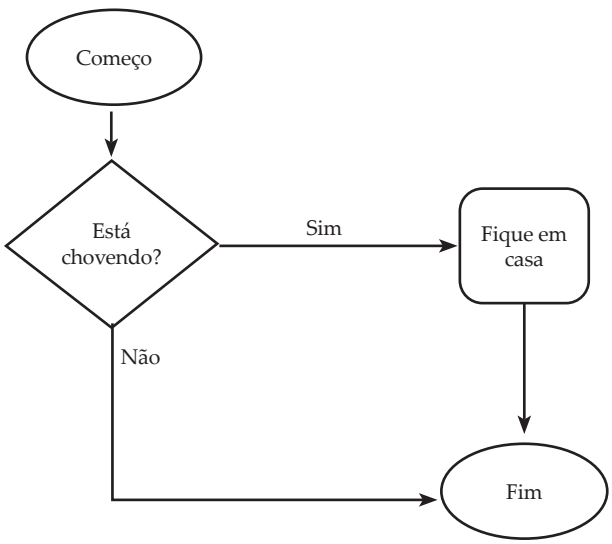
Figura 1 – Símbolos para o uso em fluxograma de diagramas para representar códigos.



Fonte: Elaborada pelo autor.

Com esses símbolos, a Figura 2 mostra em um fluxograma o processo de decisão de uma criança. Nesse caso, a criança deve ficar em casa se estiver chovendo. Um dos principais aspectos dessa figura é o símbolo de decisão, que realiza a pergunta "está chovendo?", se for verdadeiro, a criança deve ficar em casa.

Figura 2 – Fluxograma simples de um processo decisório com if.



Fonte: Elaborada pelo autor.

Esse fluxograma se traduziria para o código Python como mostra o Código 1 (que considera que estaChovendo é uma variável do tipo **bool**).

Código 1 – Código 1 do fluxograma da Figura 2.

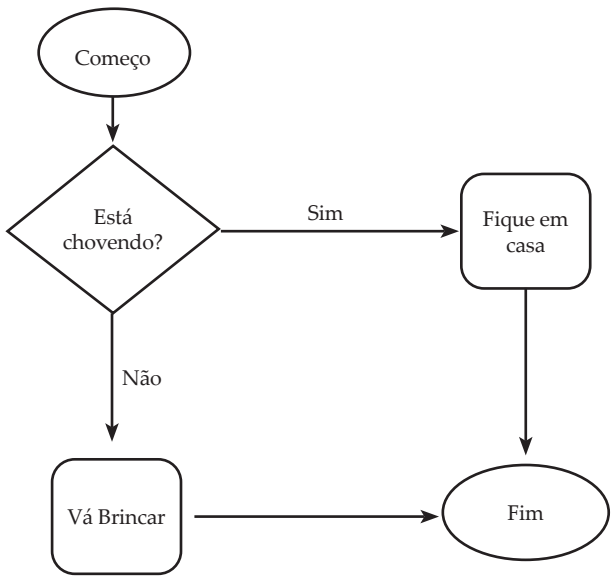
```
1. if estaChovendo:
2.     fiqueEmCasa ()
```

Fonte: Elaborado pelo autor.

Esse Código 1 mostra um comando novo da linguagem Python: **if**. Ele teria uma tradução de *se*, e verifica se a expressão é **True** (no caso estaChovendo). Caso a expressão seja **True**, ele executa a instrução dentro do bloco aninhado (endentado) do **if** (que é fiqueEmCasa( )), isto é, separado por um tab do teclado.

Mas o que a criança fará caso não esteja chovendo? Nesse caso a criança irá brincar, como mostra a Figura 3.

Figura 3 – Fluxograma simples de um processo decisório com if-else.



Fonte: Elaborada pelo autor.

Esse fluxograma se traduziria para o código Python como mostra o Código 2 (que considera que estaChovendo é uma variável do tipo **bool**).

Código 2 – Fluxograma da Figura 3.

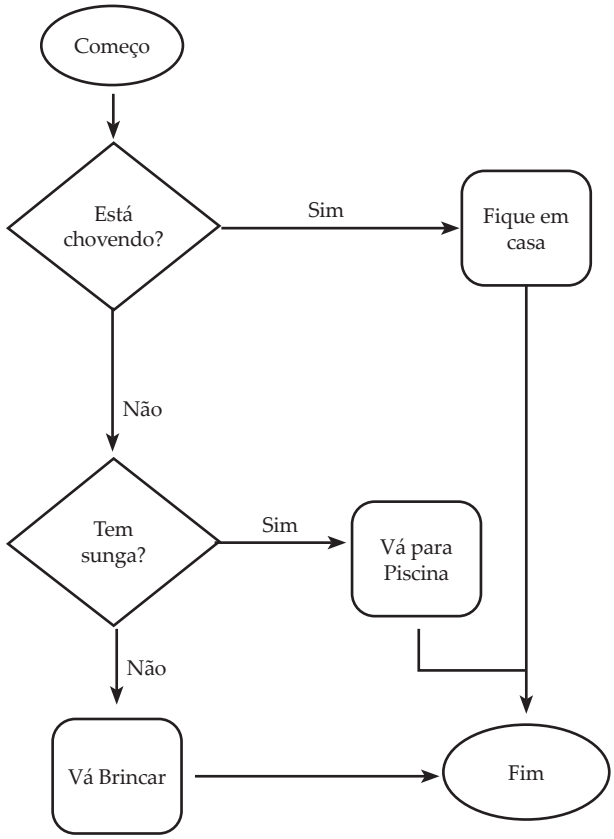
```
1. if estaChovendo:
2.     fiqueEmCasa ()
3. else:
4.     vaBrincar()
```

Fonte: Elaborada pelo autor.

Esse Código 2 mostra um comando novo da linguagem Python: o **else**, que teria tradução de *caso contrário*, e somente é executado se a condição (expressão) do **if** for **False**, nesse caso ele executa a instrução dentro do bloco aninhado do **else** (no caso vaBrincar( )).

Temos ainda a opção de, caso a expressão inicial do **if** for **False**, testar uma nova condição. Sendo assim, a criança, se estiver chovendo, fica em casa, caso contrário ela verifica se tem uma sunga, e se for **True** ela vai para piscina, e finalmente caso contrário ela vai brincar. Esse fluxo pode ser visto na Figura 4.

Figura 4 – Fluxograma simples de um processo decisório com if-elif-else.



Fonte: Elaborada pelo autor.

Esse fluxograma se traduziria para o código Python como mostra o Código 3 (que considera estaChovendo e temSunga como variáveis do tipo **bool**).

Código 3 – Código do fluxograma da Figura 4.

```
1. if estaChovendo:
2.     fiqueEmCasa()
3. elif temSunga:
4.     vaParaPiscina()
5. else:
6.     vaBrincar()
```

Fonte: Elaborado pelo autor.

O Código 3 mostra um comando novo da linguagem Python: o **elif**, que seria como **Else if** e combina os dois comandos vistos anteriormente. Esse comando teria tradução de *caso contrário se*, e somente é executado se a condição (expressão) do **if** for **False**, nesse caso ele verifica a condição (temSunga) e caso seja **True** ele executa a instrução dentro do bloco aninhado do **elif** (vaParaPiscina( )), caso contrário ele executa a instrução do **else** (vaBrincar( )).

Esse mesmo Código 3 poderia ser reescrito usando somente **if-else**, como mostra o Código 4. Nesse caso, o **elif** viraria um **else**, com um **if** aninhado, e o **else** viraria o **else** desse segundo **if**.

Código 4 – Código fluxograma da Figura 4 usando somente if-else.

```
1. if estaChovendo:
2.     fiqueEmCasa()
3.     else:
4.         if temSunga:
5.             vaParaPiscina()
6.     else:
7.             vaBrincar()
```

Fonte: Elaborado pelo autor.

Resumidamente, a estrutura **if** permite ao programa realizar decisões baseadas em uma condição. Caso essa condição seja verdadeira, as instruções aninhadas do **if** são executadas, caso contrário executa-se o **elif** ou **else**, caso tenha no código. Para facilitar a visualização da forma geral **do if-elif-else**, pode-se avaliar o Código 5.

Código 5 – Forma geral do comando if-elif-else.

```
1. if <expressão>:
2.     <instrução1>
3. elif <expressão2>:
4.       <instrução2>
5. else:
6.       <instrução3>
```

Fonte: Elaborado pelo autor.

Vídeo

3.2 A instrução if com operadores relacionais

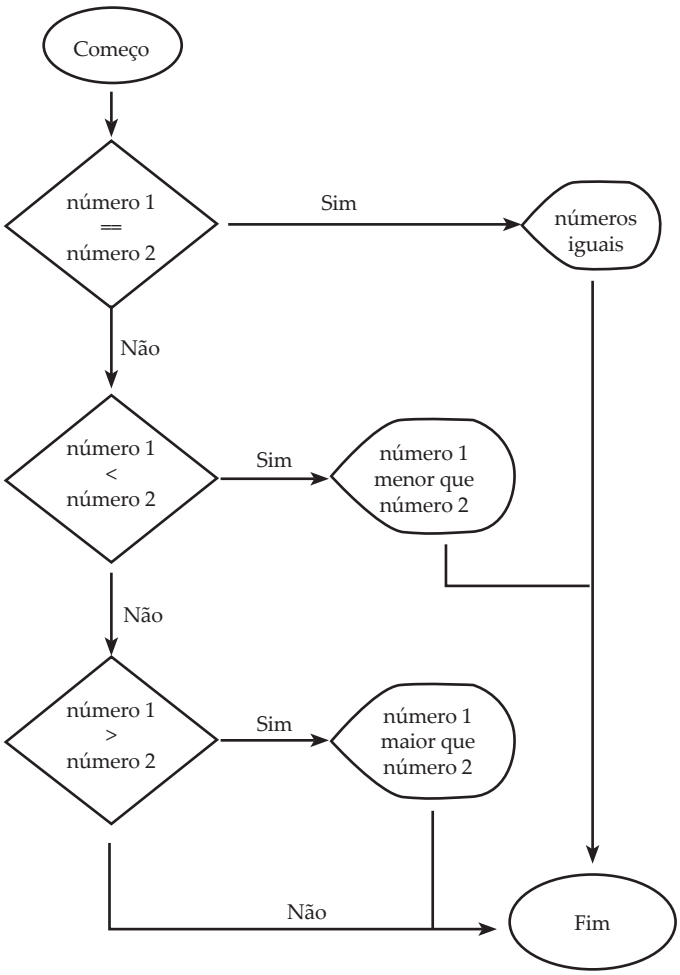
Tendo aprendido sobre como trabalhar com condições no Python, esta seção tem por finalidade aprofundar o conhecimento do leitor, mostrando a relação dos operadores relacionais com as condições.

Como visto no capítulo anterior, um operador relacional compara um valor ou expressão do lado esquerdo a um valor ou expressão do lado direito, e retorna **True** ou **False**.

Como exemplo, pode-se ter um código que tenha uma variável do tipo **int** de nome "numero1" e outra variável do tipo **int** de nome "numero2" e solicita a relação entre os dois números.

Primeiramente verifica-se se esses números são iguais com o operador de igualdade (==). Caso seja verdadeiro, escreve-se no console "números iguais", com o comando **print**. Caso contrário, testará se o "numero1" é menor que o "numero2" (com **elif** e o operador <), caso **True** imprime no console "numero1 menor que número 2". Caso contrário, testará se o "numero1" é maior que o "numero2" (com **elif** e o operador >), caso **True** imprime no console "numero1 maior que número 2". Pode-se ver como seria o fluxo desse código na Figura 5, e o código Python no Código 6.

Figura 5 – Fluxograma de um processo decisório com operadores relacionais.



Fonte: Elaborada pelo autor.

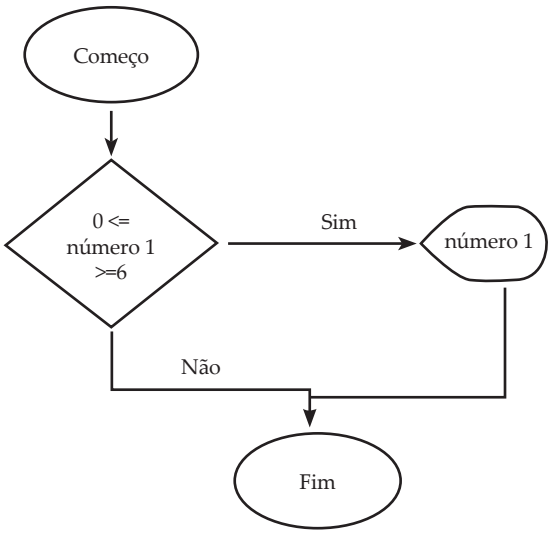
Código 6 – Fluxograma de um processo decisório com operadores relacionais.

```
1. if numero1 == numero2:
2.     print("numeros iguais")
3. elif numero1 < numero2:
4.     print("numero1 maior que numero2")
5. elif numero1 > numero2:
6.     print("numero1 menor que numero2")
```

Fonte: Elaborado pelo autor.

Também pode-se verificar se um número está em um determinado intervalo em um mesmo **if**, como mostra o fluxo da Figura 6 e do Código 7.

Figura 6 – Fluxograma de condição verificando o intervalo de um número.



Fonte: Elaborada pelo autor.

Código 7 – Fluxograma de condição verificando o intervalo de um número.

```
1. if 0 <= numero1 <= 6:
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

No Código 7, o programa irá comparar se a variável chamada "numero1" é maior ou igual a 0 e se ele é menor ou igual a 6, e caso verdadeiro irá imprimir o "numero1".

3.2.1 Condições com variáveis

Os números também podem ser usados como condição em um **if**, se o número for diferente de 0, ele é considerado **True**, e caso ele for 0 ele é considerado **False**. No caso de **strings**, a condição é considerada **False** caso ela esteja vazia e **True** caso contrário. O Código 8 mostra exemplos desses casos.

Código 8 – Exemplos de testes com variáveis.

```
1. numero1 = 5
2. numero2 = -2.0
3. numero3 = 0
4. texto = "abcd"
5. textoVazio = ""
6.
7. if numero1:
```

```
8.     print(numero1)
9.  if numero2:
10.    print(numero2)
11. if numero3:
12.    print(numero3)
13. if texto:
14.    print(texto)
15. if textoVazio:
16.    print(textoVazio)
17.    print("A variavel está vazia")
```

Fonte: Elaborado pelo autor.

Nesse código pode-se ver a declaração das variáveis das linhas 1-5. Na linha 7 verifica-se a variável "numero1", que como tem valor 5, imprime o valor. Na linha 9 verifica-se a variável "numero2", que como tem valor -2.0, imprime o valor. Na linha 11 verifica-se a variável "numero3", que como tem valor 0, não irá imprimir o valor. Na linha 13 verifica-se a variável "texto", que como ela tem valor abc, irá imprimir o valor dela. Na linha 15 verifica-se a variável "textoVazio", que como tem valor "", não irá executar o código aninhado a esse **if** (linhas 16 e 17), que imprimem o valor e um texto na tela. O resultado desse programa pode ser visto no Código 9.

Código 9 – Resultado do programa com Código 8.

```
1. /Library/Frameworks/Python.framwork/Version/3.6/bin/python3.6 "/Users/BEPID/
Dropbox (DEPiD)/iesde/...
2. 5
3. -2.0
4. abcd
5. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

3.2.2 Tipos de erros

Existem diversos tipos de erros que um programa pode ter. Por exemplo, se o programador por algum descuido se esquece de endentar a linha 17 do Código 8. Nesse caso, sempre o programa irá executar o **print**, e mostrará o texto ("A variável está vazia"), o que não era planejado. Isso é um exemplo de erro de lógica.

O programador pode realizar dois tipos de erros: de sintaxe e lógica. Um erro de sintaxe é quando o programador viola alguma regra da linguagem de programação, como esquecer de colocar o ':' depois do comando **if**, ou escrever **true** em vez de **True**. Normalmente esse tipo de erros a IDE ajuda a descobrir, mostrando e sublinhando onde ocorrem.

O outro tipo de erro, o erro lógico, tem como consequência ter um programa que acaba realizando resultados não esperados de funcionamento, e não são percebidos pela IDE. Esse tipo

de erro pode ser separado em duas categorias: fatal e não fatal. Um erro de lógica fatal faz com que o Python termine a execução do programa prematuramente e mostre uma mensagem de erro para o usuário. Um erro não fatal mostra resultados errados e continua a sua execução.

Vídeo

3.3 Combinando if com operadores lógicos

Até o momento foi visto o teste de condições únicas, em que somente uma variável ou expressão é usada como condição. Com o uso de operadores lógicos, pode-se realizar várias comparações em um mesmo comando **if**, dando maior poder ao programador.

Para realizar múltiplas condições, o programador usa variáveis e comparadores relacionais combinados com operadores lógicos. Dessa forma, o Código 6 poderia ser reescrito de forma que tivesse duas operações de relação combinadas com um operador lógico **and**, como mostra o Código 10.

Código 10 – Operador lógico and.

```
1. if 0 <= numero1 and numero1 <=6:
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

No Código 6, primeiramente o programa irá avaliar se 0 <= numero1, depois se numero1 <=6 para somente depois disso comparar os dois resultados com uma operação **and**.

3.3.1 Múltiplas condições com **or** e **and**

As condições podem ser combinadas com os parênteses, de forma a direcionar o programa qual operação deve realizar primeiro. Também múltiplos testes de comparação podem ser realizados dentro de um mesmo **if**. No código a seguir, diversas condições são efetuadas, com **ifs** aninhados.

Código 11 – Código com múltiplos ifs, relacionais e operadores lógicos.

```
1. nota = 80
2. presenca = 70
3.
4. if nota < 70 or presenca < 70:
5.         print("Você não passou")
6.         if nota < 70:
7.            print("Tente melhorar sua nota")
8.         if presenca < 70:
9.            print("Você deveria ter ido mais às aulas")
```

```
10. else:
11.           print("Você passou")
12.       if nota == 100 and presenca == 100:
13.           print("Aprovado com louvor.")
14.       elif (nota < 90 and nota > 80) and (presenca < 90 and presenca > 80):
15.           print("Bom trabalho")
16.
17.       if nota >= 90:
18.           print("Excelente Nota.")
19.       elif (nota < 90 and nota >= 80):
20.           print("Boas notas")
21.
22.       if nota < 75:
23.           print("Quase que fica reprovado por nota. Tente se dedi-
car mais as suas notas.")
24.
25.       if presenca < 75:
26.           print("Quase que fica reprovado por presença. Tente compare-
cer mais às aulas.")
```

Fonte: Elaborado pelo autor.

No Código 11, pode-se visualizar um programa, que verifica se a nota e presença estão de acordo com um padrão mínimo para a aprovação na disciplina. Dessa forma, na linha 4, testa-se se a nota ou a presença estão dentro dos critérios mínimos para a aprovação, caso isso ocorra, executa-se o bloco aninhado a esse **if** (linhas 5-9) testa-se se a reprovação foi por nota (linha 6) ou/e por presença (linha 8). Caso cumpra os critérios para a aprovação, executa-se o bloco aninhado ao **False** (linhas 11-26). Primeiramente mostra uma mensagem falando que ele foi aprovado. E após verifica-se comentários a serem efetuados sobre a nota e presença. Na linha 12 verifica se ele foi aprovado com louvor e na linha 14 se ele fez um bom trabalho. Na linha 17 verifica se as notas foram excelentes e na linha 19 se as notas foram boas. Na linha 22 verifica se ele passou por pouco em relação à nota e na linha 25 se ele passou por pouco em relação à presença. No Código 12 pode-se ver o resultado desse programa.

Código 12 – Resultado do programa com Código 9.

```
1. /Library/Frameworks/Python.framwork/Version/3.6/bin/python3.6 "/Users/BEPID...
2. Você passou
3. Boas notas
4. Quase fica reprovado por presença. Tente comparecer mais as aulas.
5. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

3.3.2 O comando **pass**

O comando **if** permite que tenha múltiplas ou uma única instrução aninhada a seu código. O Python não permite que não tenha nenhuma instrução aninhada a sua estrutura, mas permite que um comando seja colocado nesse local, que é o comando **pass**, como mostra o Código 13.

Código 13 – Mostrando o comando pass.

```
1. if numero1 <=6:
2.     pass
```

Fonte: Elaborado pelo autor.

Esse comando é interessante para ser usado somente se quiser implementar a lógica das condições para somente depois implementar a lógica das instruções e manter o código funcionando. Ele pode ser usado também com laços, classes e funções, como será visto no conteúdo mais avançado.

Ampliando seus conhecimentos

Algoritmo

(MIRANDA, 2014)

De acordo com Manzano e Oliveira (1996), o termo algoritmo data do ano de 830 d.C. devido a um estudioso e matemático originário da Pérsia, de nome Mohammed Ibn Musa Abu Djefar, pelo mesmo ter escrito um importante livro sobre álgebra. Sendo conhecido por Al-Khwarismi, seu nome passou a ser muito usado, o que foi causando mudanças na pronúncia. De Al-Khwarism passou a Al-Karismi, Algarismi, chegando a Algarismo. Algarismo é a representação numérica do sistema de cálculos utilizado nos dias atuais, e deste radical provêm o termo Algoritmo, utilizado em computação.

Conforme Souza et al. (2000b), a idéia de utilizar algoritmos para controlar o funcionamento de um computador deve-se à Ada Augusta. Sua principal contribuição foi a introdução dos conceitos de sub-rotina, laços e salto condicional.

Salvetti e Barbosa (1998) definem algoritmo como uma seqüência finita de instruções ou operações básicas, que, quando executadas, resolvem um problema computacional de qualquer instância, apoiando-se na estratégia de ordenação da seqüência de instruções estabelecida durante a análise do problema. O desenvolvimento do mesmo não pode perder-se nos tipos de dados e sua representação.

Já Manzano e Oliveira (1996) conceitua-o como "um processo matemático ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições". Pode-se ainda dizer que são regras formais com o intuito de obter um resultado ou solucionar um problema, através de fórmulas de expressões aritméticas.

Para a matemática, de acordo com Souza et al. (2000a), o algoritmo trata de um processo de cálculo, ou da resolução de um grupo de problemas semelhantes, no qual são estipuladas regras formais para se chegar a solução de um determinado problema, com generalidades e sem restrições.

Conforme Dazzi (1998), para a solução de um determinado problema podem existir diversos caminhos. Algoritmo é, exatamente, um dos caminhos para solucionar o mesmo.

Algoritmo é, para Oliveira (1998), uma sequência finita de passos lógicos escrito numa linguagem natural, em caso particular aqui pseudo-linguagem.

Dando continuidade ao seu pensamento, Oliveira (1998) afirma que o melhor exemplo para a compreensão de um algoritmo é uma receita de bolo. Neste caso são descritos todos os passos para a construção de um bolo numa seqüência lógica, como por exemplo: um bolo não vai para o forno sem estar feito; os ovos não vão com a casca, pois antes são quebrados. Estes detalhes são de conhecimento da pessoa que está fazendo o bolo, e para isso ele segue uma seqüência de passos lógicos, que denomina-se receita (o algoritmo).

Manzano e Oliveira (1996) destaca que algoritmo, na realidade, é uma "receita" de como fazer. Informa, ainda, que é a transcrição, passo a passo, de um determinado problema, para chegar a solução do mesmo.

Segundo Dazzi (1998), o algoritmo deve possuir duas virtudes: legibilidade e portabilidade. Legibilidade trata da clareza do algoritmo, ou seja, ser compreendido por qualquer pessoa que não o tenha construído. Por sua vez, portabilidade diz questão ao algoritmo não ser orientado a uma determinada linguagem de programação, pois são várias as existentes. O algoritmo deve se preocupar com a lógica em si, e poder ser implementado em qualquer linguagem de programação.

Wilt apud Souza et al. (2000b) aponta três fatores importantíssimos na elaboração de algoritmos: correção, eficiência e facilidade de implementação.

Estes fatores exigem um projeto bem determinado, onde haja uma conjunção dos fatores: robustez (sem perder a eficiência), legibilidade e facilidade de execução das ações. Para mantê-los na solução de problemas complexos, é necessário haver algum tipo de metodologia para diminuir a complexidade do desenvolvimento.

Segundo Knuth apud Souza et al. (2000b), as seguintes características distinguem um algoritmo de um conjunto de ações:

- um algoritmo sempre termina;
- cada ação é descrita sem ambigüidades e precisamente;
- cada ação é muito simples, sendo que pode ser executada em um intervalo de tempo; e
- um algoritmo sempre produz um ou mais resultados, podendo não exigirdados de entrada.

Salvetti e Barbosa (1998) destaca que sob o ponto de vista lógico, um algoritmo é constituído por três estruturas: sequencial, repetitiva e seletiva. A constituição de um algoritmo baseia-se em qualquer combinação dessas três estruturas.

O desenvolvimento de um algoritmo, segundo Salvetti e Barbosa (1998), pode ser feito por meio da técnica *top-down*, a qual identifica partes ou etapas na resolução do problema. Inicialmente elabora-se um esboço, detalhando cada etapa, até obter uma seqüência de operações básicas sobre os tipos de dados considerados. A implementação pode ser realizada em qualquer linguagem de programação, podendo ser trivial (simples transcrição de operações básicas) ou trabalhosa (dependendo das características da linguagem escolhida e dos tipos de dados nela definidos).

Atividades

1. Escreva um programa que verifica se uma variável do tipo **int** de nome "numero1" é negativa ou positiva. Ela deve imprimir "numero1 positivo", caso o número seja positivo, ou "numero1 negativo" caso o número seja negativo.

2. Escreva um programa que verifica se uma variável do tipo **int** de nome "numero1" é par ou **ímpar**. Ela deve imprimir "numero1 par", caso o número seja par, ou "numero1 impar" caso o número seja **ímpar**.

3. Escreva um programa que, dado uma variável hora do tipo **int**, verifica se está de manhã, tarde ou noite, imprimindo Bom Dia/Boa Tarde/Boa Noite de acordo com a variável.

4. Escreva um programa que, dado duas variáveis do tipo **int** (de valor 0.0 até 10.0), calcula e imprime a média e verifica se ela ficou acima de 7.0, e imprime aprovado caso afirmativo, e reprovado no caso contrário.

5. Complemente o exercício 4, dando o conceito da nota. Conceito A, para notas acima de 9.0, B entre 8.0 e 9.0, C entre 7.0 e 8.0, D entre 5.0 e 7.0 e E para valores abaixo de 5.0.

Referências

MIRANDA, Elisangela Maschio de. **Uma ferramenta de apoio ao processo de aprendizagem de algoritmos**. (2004). Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/86766/209429.pdf?sequence=1>. Acesso em: 11 set. 2017.

Resolução

1.

```
1. numero1 = -1
2.
3. if numero1 > 0:
4.         print(“numero1 positivo”)
5. elif numero1 < 0:
6.         print(“numero1 negativo”)
```

2.

```
1. numero1 = 1
2.
3. if (numero1%2) == 0:
4.         print(“numero1 par”)
5. else:
6. print(“numero1 impar”)
```

3.

```
1. hora = 10
2.
```

```
3. if 6 < hora <= 12:
4.         print(“Bom Dia”)
5. elif 12 < hora < 18:
6.         print(“Boa Tarde”)
7. elif 18 <= hora < 23:
8. print(“Boa Noite”)
```

4.

```
1. nota1 = 7.0
2. nota2 = 8.0
3.
4. media = (nota1+nota2)/2
5.
6. print(“Media:”,media)
7.
8. if media >= 7.0:
9.         print(“Aprovado”)
10. else:
11.        print(“Reprovado”)
```

5.

```
1. nota1 = 7.0
2. nota2 = 8.0
3.
4. media = (nota1+nota2)/2
5.
6. print(“Media:”,media)
7.
8. if media >= 7.0:
9.         print(“Aprovado”)
10. else:
11.        print(“Reprovado”)
12.
13. if media >= 9.0:
14.        print(“Conceito A”)
```

```
15. elif media >= 8.0:
16.         print(“Conceito B”)
17. elif media >= 7.0:
18.         print(“Conceito C”)
19. elif media >= 5.0:
20.         print(“Conceito D”)
21. else:
22.         print(“Conceito E”)
```

4
Laços

Neste capítulo pretende-se introduzir o conceito de laços de repetição, ou **loops** pelo termo em inglês. Laços são recursos que permitem que o programa realize repetições em blocos de instruções endentados ao comando do laço. Com isso cria-se um maior poder para os códigos criados pelo programador, permitindo a combinação de laços com outros comandos da linguagem. Dessa forma, o texto irá abordar inicialmente como os laços funcionam e integram a lógica de programação, apresentando a estrutura básica do comando **while**. O capítulo continuará apresentando uma outra forma de laço com o comando **for**, que tem uma estrutura de repetição um pouco diferente do comando **while**. E por fim a combinação de condições com os comandos **break** e **continue**, que permitem a quebra ou continuidade da repetição.

Vídeo

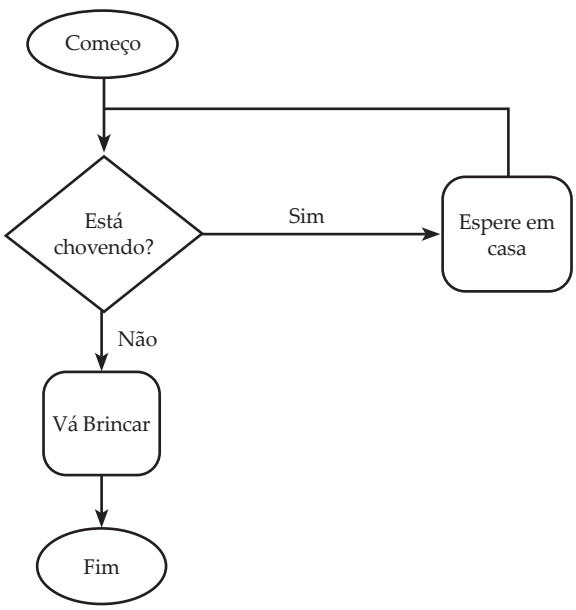
4.1 Laços

Até agora, esta obra abordou instruções em sequência, que são executadas uma de cada vez até chegar ao fim do programa. Isso funciona bem para programas simples, mas muitas tarefas que são realizadas pelos programas são repetitivas, e para isso deve-se utilizar os laços. A maioria das linguagens chama essas sequências de repetição pelo termo em inglês de **loop**.

Esta seção tem como finalidade apresentar ao leitor o conceito de laços, de forma a continuar a criação da base para o entendimento de códigos mais complexos. Uma das vantagens dos laços é que a tarefa de repetir instruções não fica mais a cargo do programador, e sim do programa, que passa a repetir instruções de acordo com o código.

Seguindo o exemplo do capítulo anterior, em que tínhamos uma criança que se estivesse chovendo ficaria em casa; e se estiver chovendo a criança espera em casa, mas continua a checar se está chovendo para ir brincar, repetindo a instrução <espere em casa enquanto está chovendo> for verdadeiro (**True**), representado pela seta que repete a ação. Pode-se ver esse fluxograma na Figura 1.

Figura 1 – Um laço simples em que uma criança checa se está chovendo ou não para poder brincar.



Fonte: Elaborada pelo autor.

Esse fluxo pode ser visto em código Python na Figura 2, em que se introduz o comando **while**, que teria tradução de *enquanto*.

Código 1 – Código do fluxograma da Figura 1.

```
1. while estaChovendo:
2.         espereEmCasa()
```

```
3.
4. vaBrincar()
```

Fonte: Elaborada pelo autor.

No caso do Código 1, enquanto a condição "estaChovendo" for verdadeira, (**True**) a função "espereEmCasa" é executada repetidas vezes. Quando o teste feito pelo **while** da condição "estaChovendo" for falso (**False**), o bloco endentado não é mais executado, e o código segue sua execução chamando a função "vaBrincar".

Pode-se verificar que o código que realiza o **while** é parecido com o código que realiza o **if**. O comando **while** primeiro checa se a condição "expressão" é verdadeira, para depois executar as instruções que estão endentadas "instrução1". Depois que essas instruções são executadas, o código repete o teste da condição e a execução das instruções até que a condição "expressão" retorne **False**. O código Python para **while** tem a seguinte estrutura.

Código 2 – Forma geral do comando while.

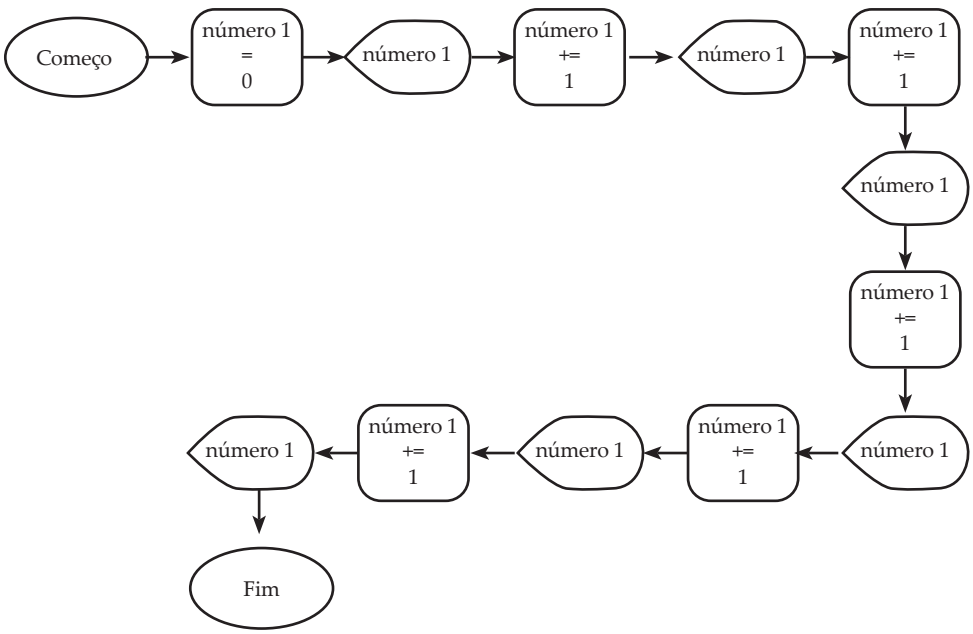
```
1. while <expressão>:
2.     <instrução1>
```

Fonte: Elaborada pelo autor.

4.1.1 Comparando o código com e sem uma estrutura de laço

Pode-se ter um código em que uma variável de valor inicial 0 é incrementada em 1 e imprime o seu valor até chegar ao valor de 5. Esse código sem uso de um laço ficaria conforme o fluxo da Figura 2 e do Código 3.

Figura 2 – Sequência de somas e imprime.



Fonte: Elaborada pelo autor.

Código 3 – Código do fluxograma da Figura 2.

```
1. numero1 = 0
2. print(numero1)
3. numero1 += 1
4. print(numero1)
5. numero1 += 1
6. print(numero1)
7. numero1 += 1
8. print(numero1)
9. numero1 += 1
10. print(numero1)
11. numero1 += 1
12. print(numero1)
```

Fonte: Elaborada pelo autor.

O resultado da execução do Código 3 pode ser visto no Código 4.

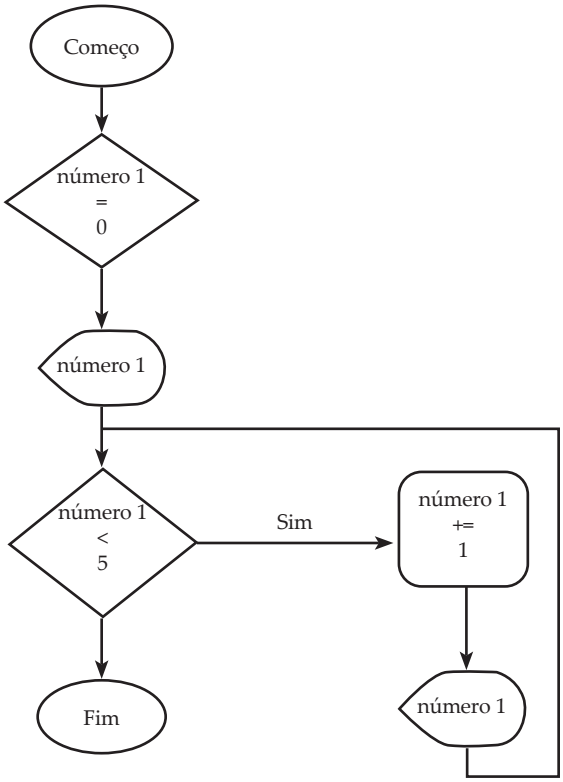
Código 4 – Resultado da execução do Código 3.

```
1. /Library/Frameworks/Pyton.framework/Version/3.6/bin/python3.6 "/Users/
BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap2/code/olaMundo.py"
2. 0
3. 1
4. 2
5. 3
6. 4
7. 5
8. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

Pode-se ver que nesse código temos as instruções das linhas 3 e 4 repetidas por 5 vezes. Se em vez de incrementar a variável numero1 e imprimir por 5 vezes, o programador quisesse fazer isso por 100 vezes, isso seria impraticável, pois ele teria que ter um programa de mais de 200 linhas. Para isso, deve-se usar um laço. Dessa forma, o leitor pode refatorar o código como mostra a Figura 3 e o Código 5. Refatorar é o processo de modificar um programa de forma a melhorar a estrutura interna de seu código, sem alterar o seu funcionamento e seu resultado, que nesse caso seria o mesmo do Código 4.

Figura 3 – Sequência de somas e imprime.



Fonte: Elaborada pelo autor.

Código 5 – Código do fluxograma da Figura 3.

```
1. numero1 = 0
2. print(numero1)
3.
4. while numero1 < 5:
5.         numero1 += 1
6.         print(numero1)
```

Fonte: Elaborada pelo autor.

Pode-se ver que o Código 5 tem bem menos linhas que no código sem laço (6 linhas contra 12 linhas) e caso o leitor mudar para imprimir até o valor 100, somente teria que mudar a condição, ficando com o mesmo número de linhas.

4.1.2 **Else** com o comando **while**

Como o comando **while** realiza uma condição, ele também pode ter um comando **else** (como o do **if**), em que somente é executado caso a condição do **while** falhe. Poderia mudar o Código 5 para ele imprimir "numero1 maior ou igual a 5" caso a condição do **while** falhe, usando o comando **else** como pode ser visto no Código 6.

Código 6 – Código 5 modificado para ter um else caso a condição falhe.

```
1. numero1 = 0
2. print(numero1)
3.
4. while numero1 < 5:
5.         numero1 += 1
6.         print(numero1)
7. else:
8.         print("numero1 maior ou igual a 5")
```

Fonte: Elaborado pelo autor.

O **else** também pode ser usado juntamente com o comando **for**, uma outra estrutura de repetição que será vista na próxima seção.

4.2 Um tipo especial de laço com o for

Vídeo

O comando **while** repete um laço de código enquanto uma condição for **True**, mas se o programador quisesse que o código repetisse um bloco de instruções um certo número de vezes, ele teria que declarar uma variável, verificar se essa variável ultrapassou um certo valor, e incrementar essa variável, como feito no Código 5. Para facilitar esse processo, existe um outro comando de repetição chamado de **for**, que teria tradução de **para**. O comando **for** trabalha com uma sequência de algum tipo, como uma **string**, lista, ou dicionários que serão vistos mais adiante.

Como exemplo poderia ter uma **string** com conteúdo texto, e se quer imprimir cada caractere dessa **string**, como mostra o Código 7.

Código 7 – Comando for para imprimir letras de uma string.

```
1. for letra in "texto":
2.     print(letra)
```

Fonte: Elaborado pelo autor.

O resultado da execução desse código pode ser visto na Figura 5. Pode-se verificar que o comando **for** declara a variável letra, e coloca sequencialmente cada caractere da **string**

"texto" (["t","e","x","t","o"]), para em seguida executar a instrução endentada ao comando, que nesse caso é um **print** que imprime o conteúdo da variável letra na tela.

Código 8 – Resultado da execução do Código 7.

```
1. /Library/Frameworks/Python.framework/Version/3.6/bin/python3.6"
2. t
3. e
4. x
5. t
6. o
7. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

Pode-se verificar que, apesar de ter um funcionamento de repetição parecido com o **while**, o **for** trabalha com sequências em vez de condições. A condição dele está implícita no comando, verificando somente se a sequência a ser usada já chegou ao fim. O comando **for** primeiro cria a sequência "sequencia" e coloca o primeiro item dela na variável "elemento" e checa se a "sequencia" está vazia ou se chegou ao fim dela, para somente depois executar as instruções que estão endentadas "instrução1". Depois que essas instruções são executadas, o código repete o processo, só que dessa vez ela seleciona o segundo item da sequência, até que não haja mais itens na sequência. O código Python para **for** tem a estrutura apresentada no Código 9.

Código 9 – Forma geral do comando for.

```
1. for <elemento> in <sequencia>:
2.     <instrução1>
```

Fonte: Elaborado pelo autor.

Pode-se verificar que o comando **for** pode ser representado pelo comando **while**, mas para isso ele precisa de mais linhas, para declarar a variável e para incrementá-la, como mostra o Código 10.

Código 10 – Forma geral do comando while.

```
1. <inicialização da variavel>
2. while <verificação se existem elementos na variavel>:
3.       <instrução1>
4.       <incremento da variavel>
```

Fonte: Elaborado pelo autor.

4.2.1 **For** com o comando **range**

O comando **for** pode ser combinado com uma função **range** (*intervalo*), que cria uma sequência de valores de 0 ao número passado como argumento, com decréscimo de um – que na verdade é uma lista de números entre 0-(final-1).

Caso o programador quisesse refatorar o Código 5 para usar somente um comando **for**, com um **range** de 6 (que irá retornar uma sequência de 0 até 5: [0,1,2,3,4,5]), teria bem menos linhas, como mostra o Código 11.

Código 11 – Código 5 refatorado para usar o for.

```
1. for numero1 in range(6):
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

Nesse caso, pode-se ver que o código original sequencial de 12 linhas ficou em somente 2 linhas. O comando **for** é bem poderoso, ele realiza a declaração da variável, verifica a condição de ela estar dentro do **range** e muda o valor da variável "numero1" para o próximo valor do **range**.

O comando **range** pode também receber argumentos que indicam o começo do intervalo. Por exemplo, caso queira uma sequência de 5 até 10 ([5,6,7,8,9,10]), deve chamar a função **range** com dois parâmetros **range** (5,11), em que o primeiro parâmetro é o começo do intervalo (5) e o segundo parâmetro é o final do intervalo (11). O **for** com esse intervalo pode ser visto no Código 12.

Código 12 – Código for com range de 5 a 11.

```
1. for numero1 in range(5,11):
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

O **range** também pode receber um terceiro parâmetro, que diz o incremento que o número inicial deve receber até chegar ao número final. Como exemplo, se fizer um **range** de 2 até 11, com incremento de 2, terá como resultado o intervalo [2,4,6,8,10], como mostra o Código 13.

Código 13 – Código for com range de 2 a 11, com incremento de 2.

```
1. for numero1 in range(2,11,2):
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

Também poderia se realizar a contagem regressiva com o **range**. Como exemplo, pode-se colocar como número inicial o 10 e realizar decréscimo até 0, colocando como terceiro argumento o número -1, dessa forma ele terá como sequência o intervalo [10,9,8,7,6,5,4,3,2,1]. O Código 14 mostra o uso desse **range**.

Código 14 – Código for com range de 10 a 0, com decremento de 1.

```
1. for numero1 in range(10,0,-1):
2.     print(numero1)
```

Fonte: Elaborado pelo autor.

Vídeo

4.3 Usando break e continue para modificar a execução do laço

Até o momento vimos como realizar uma repetição até que a condição inicial do laço falhe. Com o uso dos comandos **break** e **continue**, o programador pode alterar esse fluxo de execução dos testes de condição, fazendo com que o código saia da estrutura de repetição (do código endentado a ele), ou apenas pule as instruções seguintes e passe para a próxima interação do fluxo.

Como exemplo temos o **break**, que faz com que ele saia de uma estrutura **while** ou **for** quando executado. Podemos ter um contador de 5 até 11, como no Código 12, mas caso o "numero1" seja igual a 7, realiza o **break**, como mostra o Código 15.

Código 15 – Código for com break.

```
1. for numero1 in range(5,11):
2.     if numero1 == 7:
3.         break
4.     print(numero1)
5. else:
6.     print("else")
```

Fonte: Elaborado pelo autor.

Nesse caso, o código executado tem o resultado do Código 16. Pode-se verificar que o código somente imprime 5 e 6, pois quando a variável "numero1" tem valor 7, ele retorna **True** da condição da linha 2 e executa o **break** da linha 3, fazendo com que pare a execução do laço de repetição. Um detalhe nesse caso é que como o **while** terminou a execução devido a um **break**, e não da condição inicial falhar, ele não executa o bloco endentado pelo **else** da linha 6.

Código 16 – Resultado da execução do Código 15.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. 5
3. 6
4. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

Uma forma comum de usar o **break** com o **while** é quando a condição de parada depende de algum cálculo que é executado dentro do código endentado da estrutura de repetição. Nesse caso teríamos um **loop** com condição sempre verdadeira (1 ou **True**), e a saída do **loop** daria pela condição do **break**. Um erro de lógica comum nessa abordagem é realizar um laço infinito, que o programa executa sem parada. Pode-se ver um código com uma variável "numero1" incrementada em 3 até chegar ao valor de 100, como no Código 17.

Código 17 – Código while com break.

```
1. while 1:
2.         numero1 += 3
3.         if numero1 >= 100:
4.            break
```

Fonte: Elaborado pelo autor.

Já o comando **continue**, quando executado endentado em um comando de repetição, pula a execução das instruções remanescentes e vai para a próxima iteração do código. Como exemplo podemos ter um contador de 5 até 11, como no Código 12, mas caso o "numero1" seja igual a 7, realiza o **continue**, como mostra o Código 18.

Código 18 – Código for com continue.

```
5. for numero1 in range(5,11):
6.      if numero1 == 7:
7.         continue
8.      print(numero1)
```

Fonte: Elaborado pelo autor.

Nesse caso, o código executado tem o resultado do Código 16. Pode-se verificar que o código imprime todos os números do **range** menos o 7, pois quando a variável "numero1" tem valor 7, ele retorna **True** da condição da linha 2 e executa o **continue** da linha 3, fazendo com que passe para a próxima iteração do laço de repetição, sem executar o comando **print** da linha 4.

Código 19 – Resultado da execução do Código 18.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. 5
3. 6
4. 8
5. 9
6. 10
7. Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

As estruturas de repetição também podem ser combinadas com condições, operadores lógicos e relacionais e até mesmo ter laços dentro de laços, permitindo ao programador uma gama de ferramentas para criar suas estruturas de código.

Ampliando seus conhecimentos

Programação imperativa

(SILVA, 2011, p. 30-34)

O paradigma imperativo apoia-se na base teórica proporcionada pela Máquina de Turing, tendo como principal lastro tecnológico, a arquitetura de von Neumann. Essa arquitetura leva os programas a terem como recurso central de armazenamento de informação valores armazenados em memória, em forma de estruturas de dados, ou seja, agrupamento de variáveis. As instruções do programa costumam também ser organizadas em posições lógicas contíguas de memória, o que pode tornar mais eficiente o processamento (PELEGRINI, 2009).

Nesse paradigma, a ideia central é o conceito de estado de um programa, materializado na configuração da memória do programa e dos seus dados, e que é alterado durante a execução do mesmo através de sucessivas modificações dos valores das variáveis, impostas pelas instruções do programa sobre o seu estado.

Essa função é desempenhada, principalmente, pelos comandos de atribuição, que alteraram o estado de um programa através da substituição de um valor, contido em uma posição de memória, por algum outro valor.

Quando uma linguagem é capaz de fornecer recursos adequados que permitam a implementação de qualquer algoritmo que possa ser projetado, essa linguagem se diz Turing-Completa. Dessa forma, uma linguagem de programação imperativa que disponibilize variáveis e valores inteiros, as operações aritméticas básicas, comandos de atribuição, comandos condicionais e iterativos é considerada Turing-completa (SEBESTA, 2010).

Além dos comandos de atribuição, as linguagens de programação imperativas costumam disponibilizar ao programador: declarações de variáveis, expressões, comandos condicionais, comandos iterativos e abstrações procedimentais.

A abstração procedimental dá ao programador a possibilidade de atentar para as relações existentes entre um procedimento e a operação que ele

realiza (em particular, entre uma função e o cálculo que ela executa) sem preocupações acerca da maneira como essas operações são realizadas.

O refinamento gradual de abstrações é uma maneira sistemática muito empregada no desenvolvimento de programas e de muitas outras modalidades de sistemas computacionais (técnica dos refinamentos sucessivos). Usando abstração procedimental, um programador pode particionar a lógica de uma função, idealizada de forma macroscópica, em um grupo de funções mutuamente dependentes, mais simples e/ou mais específicas.

Com o fim de simplificar o desenvolvimento de algoritmos complexos, as linguagens imperativas modernas oferecem ao programador suporte a matrizes e estruturas de registro, além de bibliotecas extensíveis de funções, que, facilitando o reaproveitamento de partes já desenvolvidas de programas, evitam que operações comuns necessitem ser reprogramadas, como é o caso de operações de entrada e saída de dados, gerenciamento de memória, manipulação de cadeias, estruturas de dados clássicas, etc.

São exemplos de linguagens imperativas, entre inúmeras outras, as linguagens FORTRAN e C.

[...]

Atividades

1. Escreva um programa usando o comando **for** que imprime todos os números pares menores que 1000 em ordem crescente.

2. Agora escreva o programa da questão 1 usando o comando **while**.

3. Escreva um programa usando o comando **for** que imprime todos os números ímpares menores que 1.000 em ordem decrescente.

4. Agora escreva o programa da questão 1 usando o comando **while**.

5. Qual a diferença entre as instruções **range**(10), **range**(0,10) e **range**(0,10,1)?

6. Imprima o código a seguir:

```
1. for i in range(3):
2.     print("i:",i)
```

```
3.      if i == 2:
4.          continue
5.      for j in range(i+1):
6.          print("j:",j)
7.          if j > 1:
8.              break
```

Referências

MUELLER, J. P. **Começando a programar em Python para leigos**. Stalin, 2016.

PYTHON SOFTWARE FOUNDATION. **Documentation**. Disponível em: <http://www.python.org>. Acesso em: 4 jun. 2017.

SILVA, S. R. B. **Software adaptativo: método de projeto, representação gráfica e implementação de linguagem de programação**. Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de mestre em Engenharia Elétrica. São Paulo, 2011.

JET BRAINS SRO. **Getting started**. Disponível em: <https://www.jetbrains.com/pycharm/documentation/>. Acesso em: 4 jun. 2017.

Resolução

1.

```
1. for i in range(0,1000,2):
2.     print(i)
```

2.

```
1. numero = 0
2. while numero < 1000:
3.       print(numero)
4.       numero += 2
```

3.

```
1. for i in range(999,0,-2):
2.     print(i)
```

4.

```
1. numero = 999
2. while numero > 0:
```

```
3.          print(numero)
4.          numero -= 2
```

5. Nenhuma.

6. Irá imprimir:

i: 0

j: 0

i: 1

j: 0

j: 1

i: 2

5
Entrada e saída de dados

Neste capítulo pretende-se introduzir o entrada e saída de dados. Apesar desta obra já ter apresentado a saída de dados com o comando **print**, será visto este comando com todos os seus detalhes, para melhor formatar a saída ao usuário. A maioria dos programas necessita de uma interação do usuário de forma a criar interatividade. Sendo assim, neste capítulo será mostrado também como trabalhar com o **input** do usuário, para ter interação com o usuário e ter como parte dos programas o cálculo a partir da entrada de dados. O capítulo finalizará apresentando os comentários, estruturas auxiliares ao código que ajudam no seu entendimento, que são ignorados pelo interpretador, isto é, ele não é executado, mas ajuda no entendimento do código.

Vídeo

5.1 Saída de dados

Nesta obra, já nos primeiros capítulos, foi visto e utilizado o comando **print**, mas até agora ele não foi explicado. Dessa forma, esta seção pretende mostrar todo o poder da função **print**.

O **print** tem como principal funcionalidade imprimir em forma textual os argumentos passados para ele. O comando pode possuir diversos argumentos, como mostra o Código 1.

Código 1 – A função print com seus argumentos.

```
1. print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Fonte: Elaborado pelo autor.

Como será visto no próximo capítulo, que é focado em funções, o **print** é uma função e tem argumentos (que são valores ou variáveis que são passadas dentro dos parênteses). Ele tem como argumento um ou mais valores, de forma a imprimir esses valores usando o parâmetro "sep = ' '" que diz para colocar entre os valores esse separador (que por padrão é inicializado como um espaço ' ') e depois de imprimir todos os valores ele usa o parâmetro "end = '\n'" para colocar um caractere de nova linha no final da impressão. Os outros valores dizem que ele deve imprimir para o arquivo sys.stdout (file=sys.stdout), que é o console, e que ele não deve fazer **flush** (flush=false). O **flush** diz para ele mandar o **buffer** contendo o texto para o arquivo. Normalmente o interpretador somente manda o texto para o arquivo quando ele recebe o comando de nova linha, mas se o **flush** tiver valor **True**, ele irá mandar diretamente para a saída o texto, mesmo sem ter uma nova linha.

O programador poderia comandar o computador a imprimir duas variáveis e dentre elas colocar um texto para facilitar a visualização. Dessa forma, o programador passaria as variáveis e **strings** como mostra o Código 2.

Código 2 – Imprimindo duas variáveis com strings auxiliares para facilitar a visualização.

```
1. a = 1
2. b = 2
3. print("a=",a,"e b=",b)
```

Fonte: Elaborado pelo autor.

Dessa forma o programa tem como saída uma **string** de uma linha, como mostra o Código 3.

Código 3 – Saída do Código 2.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. a= 1 e b= 2
3. Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

5.1.1 Usando o argumento **sep** para modificar a saída

O programador poderia modificar o argumento separador para modificar a saída. No Código 3 há a modificação do mesmo Código 2, de forma a ter o argumento sep='\t'. Dessa forma, em vez de usar um espaço para separar os elementos, usaria uma tabulação.

Código 4 – Modificando o Código 3 para usar um separador de tabulação.

```
1. a = 1
2. b = 2
3. print("a=",a,"e b=",b,sep='\t')
```

Fonte: Elaborado pelo autor.

Sendo assim, o programa tem como saída a **string** de uma linha, com uma tabulação para separar os argumentos do **print**, como mostra o Código 5.

Código 5 – Saída do Código 4.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. a=   1 e b=      2
3. Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

5.1.2 Usando o argumento **end** do comando para modificar a saída

Também poderia se modificar o argumento final para modificar a saída. Normalmente ele tem como valor o '\n' que coloca uma nova linha depois do comando. Dessa forma, poderia se modificar o programa do Código 2 para ele realizar a mesma tarefa, só que dessa vez usando dois **prints**, um para imprimir os dados da variável **a** e outro para os dados da variável **b**, como parâmetro **end** do primeiro **print**, coloca-se um espaço vazio (end=' ') em vez do caractere de nova linha (end='\n'), como mostra o Código 6.

Código 6 – Modificando o Código 2 para usar um separador de final de print.

```
1. a = 1
2. b = 2
3. print("a=",a,end=' ')
4. print("e b=",b)
```

Fonte: Elaborado pelo autor.

Dessa forma, o programa tem como saída a **string**, como mostra o Código 7.

Código 7 – Saída do Código 6.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. a=  1 e b=    2
3. a= 1 e b=  2
4. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

5.1.3 Combinando operadores com o **print**

Poderia também usar o comando **print** com operadores. No caso pode-se colocar o operador **soma** para concatenar duas **strings**, ou até mesmo realizar uma operação matemática nos números. Também poderia se utilizar da multiplicação para imprimir várias vezes o mesmo conteúdo. Exemplos dessas operações podem ser visualizadas no Código 8, que mostra na linha 3 como imprimir **strings** concatenadas, na linha 4 como imprimir o resultado de uma operação matemática e na linha 5 como imprimir uma **string** múltiplas vezes.

Código 8 – Exemplo de operações.

```
1. a=1
2. b=2
3. print("a" + "b")
4. print(a + b)
5. print(5 * "a")
```

Fonte: Elaborado pelo autor.

Esse programa tem como saída as **strings**, como mostra o Código 9.

Código 9 – Saída do Código 8.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. ab
3. 3
4. aaaaa
5. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

5.1.4 Formatando strings com o operador %

Existe no Python um operador especial %, que permite ao programador formatar a saída. Ele tem dois argumentos: uma **string** formatada, com o operador % no meio da **string**, e um valor. Um exemplo de uso do operador % com **print** pode ser visto no Código 10.

Código 10 – Mostra como usar o operador % com um número inteiro.

```
1. print("a=%d" % 5)
```

Fonte: Elaborado pelo autor.

O Código 10 coloca como parâmetro à esquerda do operador % uma **string** com valor "a tem valor de %d". Nessa **string** existe o %d, em que o % indica o local que será substituído pelo valor do parâmetro e coloca à direita do operador %, que no caso é 5, e o **d** indica que este valor será um inteiro. Dessa forma, o programa tem como saída a **string**, como mostra o Código 11.

Código 11 – Saída do Código 10.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. a=5
3. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

Para imprimir outros tipos de valor, o % também pode usar outros caracteres, como o **s,** para **strings** e o **f** para números com ponto flutuante, como mostra o Código 12.

Código 12 – Mostra como usar o operador % de diversas formas.

```
1. print("a=%s" % "a")
2. print("a=%f" % 1.5324)
3. print("%s=%f" % ("a",1.5324))
```

Fonte: Elaborado pelo autor.

No Código 12, pode-se ver que na linha 1 o primeiro argumento tem o %s, ele indica que o segundo argumento do operador será uma **string**, que no caso é o "a". Na linha 2, pode-se ver que o primeiro argumento tem um %f, que indica que o segundo argumento do operador será um número de ponto flutuante, neste caso o valor 1.5324. Na linha 3 existem dois % no primeiro argumento do operador %, dessa forma, o segundo argumento tem dois valores passados dentro de um parêntese. No primeiro argumento, o primeiro % tem um s, mostrando que ele espera uma **string**, que no caso é o primeiro valor do segundo argumento, a **string** "a", e no segundo % tem o .1f que diz que deve colocar um número flutuante, mas somente na primeira casa decimal dele (pelo número 1). Dessa forma pode-se formatar o número de casas decimais para aparecer na tela. Dessa forma o programa tem a saída como mostra o Código 13.

Código 13 – Saída do Código 12.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. a=a
3. a=1.532400
4. a=1.5
5. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

Verificar com o autor se a cor está correta.

5.1.5 Imprimindo em um arquivo

O comando **print** é bem poderoso, ele também pode ser usado para imprimir a saída para um arquivo de texto. Dessa forma, deve-se primeiramente abrir esse arquivo com permissão de escrita, escrever no arquivo e depois fechá-lo. O Código 14 mostra a abertura e escrita de um arquivo.

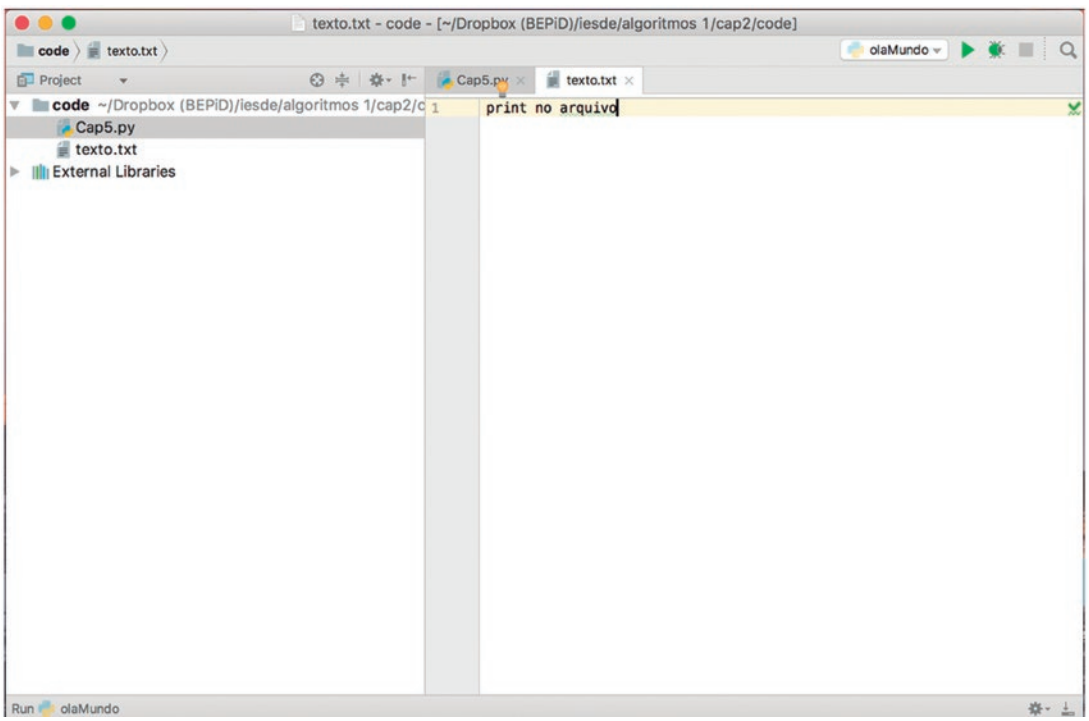
Código 14 – Mostra como usar o print para imprimir em um arquivo.

```
1. arquivo = open("arquivo.txt","w")
2. print("print no arquivo",file=arquivo)
3. arquivo.close()
```

Fonte: Elaborado pelo autor.

No Código 14 observa-se que a linha 1 guarda o resultado da chamada de uma função **open** em uma variável de nome "arquivo". Essa função **open** é responsável por realizar abertura de arquivos. Ela tem dois argumentos, primeiro o nome do arquivo a ser aberto, no caso "arquivo.txt" (que cria também o arquivo caso seja necessário), e o segundo argumento o modo que o arquivo deve ser aberto, no caso escrita "w", mas poderia ser também de leitura com o "r" ou de anexar conteúdo extra ao arquivo com o "a" de **append**. Após realizar ações com o arquivo, ele deve ser fechado. Para isso, chama-se a variável com o **close** seguido de um ponto, como mostra a linha 3. Como saída pode-se observar a criação de um arquivo "arquivo.txt" no diretório do código Python, com o conteúdo "print no arquivo", como mostra a Figura 1.

Figura 1 – Criando e imprimindo um arquivo, como mostra o Código 14.



Fonte: Elaborada pelo autor.

Vídeo

5.2 Entrada de dados

Muitas poucas aplicações existem sem ter dados externos, que podem vir de diversos lugares, como dados da internet, de outros programas, mas principalmente vindas do

usuário. Na verdade, a maioria das aplicações interage com o usuário, pois os programas e os computadores são concebidos para auxiliar as pessoas em suas diversas necessidades.

Para interagir com o usuário, é necessário obter essa entrada de dados do usuário. Para esse fim, esta obra se utilizará da função Python chamada **input**, que espera uma entrada de dados do usuário, feita pelo teclado (que somente considera fim da entrada quando o usuário pressionar a tecla enter) no console de dados (mesmo local onde se imprime a saída de dados com o comando **print**, visto na seção anterior).

O comando **input** pode ser usado sem parâmetros, ou pode ter como parâmetro uma **string**, de forma a indicar ao usuário a entrada que ele deve colocar. O comando **input** pode ser visto no Código 15.

Código 15 – A função input e seu argumento opcional.

```
1. input(value1)
```

Fonte: Elaborado pelo autor.

Por padrão, a função **input** tem como retorno um valor do tipo **string**, caso se queira no programa que este valor seja de outro tipo, cabe ao programador converter entre os tipos.

5.2.1 Requisitando o nome do usuário

Como exemplo, o programador pode ser requisitado a escrever um programa que pergunta a seu usuário seu nome, e com isso o complementa. Um código pode ser visto no Código 16.

Código 16 – Mostra como usar o input para requerer e imprimir o nome do usuário.

```
1. nome = input("Digite seu nome:")
2. print("Olá,",nome)
```

Fonte: Elaborado pelo autor.

No Código 16, tem-se a linha 1, que é a função **input**, com argumento "Digite seu nome:", e seu retorno guardado na variável de nome "nome". Esse comando irá imprimir o argumento ao usuário (que é "Digite seu nome:"), e irá esperar o usuário entrar com o nome e pressionar a tecla enter para guardar o valor digitado na variável **nome**. Na linha 2, o programa usa o comando **print** para imprimir a **string** "Olá,", e o valor armazenado na variável **nome**. A saída da execução desse programa, logo após a inserção do nome requisitado, pode ser vista no Código 17.

Código 17 – Saída do Código 16.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. Digite seu nome:Mark
3. Olá, Mark
4. Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

5.2.2 Criando um programa para somar dois números inteiros

O programador poderia ser requisitado para criar um programa que pede ao usuário para digitar dois números inteiros, somar esses números e apresentar o resultado. Um código realizando essa tarefa pode ser visto no Código 18.

Código 18 – Mostra como usar o print para imprimir em um arquivo.

```
1. numero1 = int(input("Digite o primeiro número:"))
2. numero2 = int(input("Digite o segundo número:"))
3. print(numero1,"+",numero2,"=",numero1+numero2)
```

Fonte: Elaborado pelo autor.

Caso o usuário digite qualquer coisa diferente de um inteiro, como uma **string** ou um **float**, o programa realizará um erro de execução fatal e terminará a sua execução, como mostra o Código 19. Uma das formas de evitar isso é usando **try/catch**, que é um comando mais avançado para evitar e contornar erros fatais.

Código 19 – Erro fatal ao tentar converter uma string para inteiro.

```
1. Digite o primeiro numero:a
2. Traceback (most recente call last):
3.   File "/Users/BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap2/code/olaMundo.py",
       line 24, in <module> numero1 = inte(input("Digite o primeiro numero:")0
4. ValueError: invalid literal for int() with base 10: 'a'
5. Process finished with exit code 1
```

Fonte: Elaborado pelo autor.

Vídeo

5.3 Comentando seu código

Comentários são textos ou linhas de texto que são ignoradas pelo interpretador/compilador. Dessa forma, pode-se colocar comentários no meio do código, de forma a facilitar o entendimento no futuro pelo programador, ou por algum terceiro que tenha necessidade de ler o código. A ideia dos comentários é prover informação extra ou explicações sobre alguma parte do código. Comentários também podem servir como lembretes, de parte de código que foram esquecidas de implementar, ou de algum *bug* que deve ser resolvido no futuro.

O Python utiliza o caractere "#" para determinar que o resto da linha deve ser um comentário. Um exemplo pode ser visto no Código 20.

Código 20 – Comentário.

```
1. #Isso é um comentário
```

Fonte: Elaborado pelo autor.

Como comentários não são executados, caso se execute um programa com essa linha, ele não terá nada como saída.

Um comentário pode ser colocado antes de uma linha de código, durante (na mesma linha do código) ou em uma linha posterior. O Código 21 mostra exemplos desse tipo de comentário.

Código 21 – Mostra diferentes tipos de comentários.

```
1. #começo do programa
2. print(“Olá,”) #imprime Olá
3. #fim do programa
```

Fonte: Elaborado pelo autor.

5.3.1 Comentários em múltiplas linhas

Pode-se querer colocar textos grandes para os comentários, de forma a explicar o funcionamento do código. Uma das formas de realizar isso seria a cada linha colocar um #, como mostra o Código 22.

Código 22 – Comentários em múltiplas linhas.

```
1. #linha 1
2. #linha 2
3. #linha 3
```

Fonte: Elaborado pelo autor.

Esse processo pode ser ruim, ou trabalhoso, requerendo a cada nova linha um #. Outra forma de realizar isso é usando três aspas simples (‘’’) que comenta todas as linhas até encontrar novamente três aspas simples (‘’’). Um exemplo pode ser visto no Código 23.

Código 23 – Comentários em múltiplas linhas usando três aspas.

```
1. ‘’’’’linha 1
2. linha 2
3. linha 3’’’
```

Fonte: Elaborado pelo autor.

Os comentários em múltiplas linhas podem ser bastantes úteis quando se quer colocar algum código como comentário, de forma que ele não seja executado, como por exemplo quando o código tem algum erro e o programador quer testar outra parte do código, ou quer que o código seja executado sem essa parte afetar a execução do projeto.

Ampliando seus conhecimentos

Engenharia de software para software livre

(BROD, 2017, p. 2-3)

Python

A linguagem de programação Python foi lançada para a comunidade em 1991, o que significa que ela é contemporânea do kernel Linux. Seus desenvolvedores, porém, procuraram, desde o princípio, documentar o estilo, a cultura e as ferramentas de desenvolvimento. A maior fonte de referência para este documento foi o portal oficial da linguagem.

A cultura Python tem muito de humor e leveza. O próprio nome da linguagem deriva do grupo de humor britânico Monty Python e o criador da linguagem, Guido van Rossum, é chamado de Benevolente Ditador Vitalício. Em 1999, Tim Peters, junto com Guido, publicaram na lista de discussão comp.lang.python os princípios de projeto da linguagem pedindo, na mesma publicação, que os mesmos não fossem levados tão a sério (os autores não recomendam que a lista de princípios seja usada como tatuagem, por exemplo). Ainda assim, tais princípios ilustram bem a cultura e o estilo de desenvolvimento do Python:

- belo é melhor que feio;
- explícito é melhor que implícito;
- simples é melhor que complexo;
- complexo é melhor que complicado;
- plano é melhor que aninhado;
- esparso é melhor que denso;
- legibilidade conta;
- casos especiais não são especiais o suficiente para violar as regras;
- ainda que a praticidade vença a pureza;

- erros nunca devem passar silenciosamente;
- a não ser que sejam explicitamente silenciados;
- em caso de ambiguidade, resista à tentação de adivinhar;
- deve haver uma - e apenas uma - maneira óbvia de fazer algo;
- mesmo que tal maneira não seja tão óbvia à primeira vista, a não ser que você seja holandês;
- agora é melhor do que nunca;
- embora nunca seja frequentemente
- melhor do que exatamente agora;
- se a implementação é difícil de explicar, a ideia é ruim;
- se a implementação é fácil de explicar, talvez a ideia seja boa;
- espaços de nomes (namespaces) são uma ideia fantástica - vamos fazer mais deles!

Estes princípios, associados à forma de programação orientada a objetos da linguagem (que exige uma identação formal em sua sintaxe), levam a um código limpo e legível.

As ferramentas usadas no desenvolvimento da linguagem são descritas por Brett Cannon em "Guido, Some Guys, and a Mailing List: How Python is Developed". Para o registro e controle de problemas é usada a ferramenta RoundUp, que também é usada para a submissão de sugestões de código e solicitação de novas funcionalidades; para o controle de versões, o Subversion; listas de discussões distintas são usadas para a comunicação dos grupos de desenvolvimento, gestão de problemas e para anúncios diversos. Sugestões de melhorias para a linguagem passam por um processo de análise e aprovação mais formal, iniciado pela submissão de um PEP (Python Enhancement Proposal – Proposta de Melhoria do Python), que além de passar pela equipe de desenvolvimento, deve ter o aval do mantenedor da linguagem (o Benevolente Ditador Vitalício).

[...]

Atividades

1. Escreva um programa que pergunta os dados básicos de uma pessoa (nome, cidade e ano de nascimento). A partir disso o programa imprime esses dados, colocando uma nova linha entre esses dados e a idade da pessoa.

2. Faça um programa que peça ao usuário a temperatura em Celsius e retorne a temperatura em Fahrenheit.

3. Escreva um programa que coloque a seguinte saída na tela.

ALUNO(A)	NOTA
Mark	9.0
Marcio	DEZ
Maria	4.5
João	7.0

4. Escreva um programa que peça para o usuário entrar com um número, que imprime um quadrado de X de acordo com o número inserido, como o exemplo.

   Caso o usuário inserir 1, a saída fica:

   X

   Caso o usuário inserir 2, a saída fica:

   XX

   XX

   Caso o usuário inserir 3, a saída fica:

   XXX

   XXX

   XXX

   E assim por diante.

5. Escreva um programa que receba 2 números e os coloque em ordem crescente.

6. Escreva um programa que receba caracteres até o usuário inserir um caractere "*". Então ele coloca na saída todos os caracteres inseridos.

7. Escreva um programa que receba 2 números e imprima em um arquivo se o primeiro número é múltiplo do segundo ou não.

Referências

BROD, C. A. **Engenharia de software para software livre**. Programa de pós graduação – Instituto de Informática da Universidade Federal do Rio Grande do Sul. Disponível em: <http://www.brod.com.br/files/engsoftlivre.pdf>. Acesso em: 7 jul. 2017.

Resolução

1.

```
1. nome = input(“Digite o seu nome:”)
2. cidade = input(“Digite a sua cidade:”)
3. ano = int(input(“Digite o seu ano de nascimento:”))
4.
5. print(nome)
6. print(cidade)
7. print(2017-ano)
```

2.

```
1. celsius = int(input(“Digite a temperatura em Celsius que vc gostaria de converter:”))
2.
3. fahrenheit = 9.0/5.0 * celsius + 32
4.
5. print(“Temperatura:”, celsius, “Celsius = “, fahrenheit, “ F”)
```

3.

ALUNO(A)	NOTA
Mark	9.0
Marcio	DEZ
Maria	4.5
João	7.0

```
1. print(“ALUNO (A)”,”NOTA”,sep=’\t’)
2. print(9*”=”,5*”=”,sep=’\t’)
3. print(“Mark”,”9.0”,sep=’\t\t’)
4. print(“Marcio”,”DEZ”,sep=’\t\t’)
5. print(“Maria”,”4.5”,sep=’\t\t’)
6. print(“João”,”7.0”,sep=’\t\t’)
```

4.

```
1. tamanho=int(input("Digite o tamanho do quadrado:"))
2.
3. for i in range(tamanho):
4. print(tamanho * "X")
```

5.

```
1. numero1 = int(input("Digite o primeiro numero:"))
2. numero2 = int(input("Digite o segundo numero:"))
3.
4. if numero1 > numero2:
5.     print(numero2,numero1)
6. else:
7.     print(numero1,numero2)
```

6.

```
1. character = ""
2. imprimir = ""
3.
4. while character != "*":
5.         character = input("Digite um caracter (digite * para terminar):")
6.         imprimir += character
7.
8. print(imprimir)
```

7.

```
1. numero1 = float(input("Digite o primeiro numero:"))
2. numero2 = float(input("Digite o segundo numero:"))
3.
4. arquivo = open("texto.txt","w")
5.
6. if 0 == numero1%numero2:
7.       print("primeiro numero é multiplo do segundo ",file=arquivo)
8. else:
9.       print("primeiro numero não é multiplo do segundo ", file=arquivo)
10.
11. arquivo.close()
```

6
Listas

Neste capítulo será introduzido o conceito de listas, que podem ser usadas para colocar valores dentro de uma sequência. Já foi visto no Capítulo 4 o **range**, que é também um tipo de sequência. Uma lista tem o conceito semelhante às listas reais, aquelas que são usadas no dia a dia, com elementos e uma sequência entre esses elementos. Listas em Python são extremamente importantes e permitem a criação de programa mais complexos. Este capítulo se concentra em exaurir o assunto, apresentando as principais características e métodos das listas na seção 1. Na seção 2 apresenta-se o uso de listas como dois tipos específicos de estruturas de dados, a **pilha** e a **fila**. E finalmente na seção 3 apresenta-se as compreensões de listas, que é uma forma sucinta e fácil de criar listas combinando-as com o comando **for**.

Vídeo

6.1 Listas

Esta seção se concentrará em mostrar o tipo básico **list** do Python, que é uma sequência indexada por um número, ou seja, uma lista. A **list** pode ser considerada como uma **string**, só que em vez de caracteres, ela pode ser colocada com diversos tipos de elementos, até em uma outra **list**. O Código 1 mostra a criação de duas listas, a primeira com uma variável de nome "listaVazia" e sem nenhum elemento, com o nome, e a segunda com uma variável de nome "lista", com conteúdo de uma lista com um **int**, um **float**, um **booleano** e uma **string**.

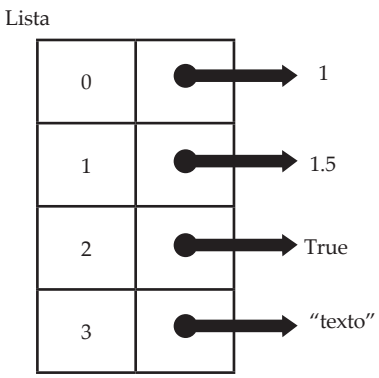
Código 1 – A função print com seus argumentos.

```
1. listaVazia = []
2. lista = [1,1.5,True,"texto"]
```

Fonte: Elaborado pelo autor.

As **lists** são colocadas na memória e referenciadas por um número, chamado **index** (que tem tradução de *índice*) e começa sempre no 0. Dessa forma, a **list** de nome *lista* do Código 1 na memória pode ser ilustrada pela Figura 1.

Figura 1 – A list e sua ilustração na memória.



Fonte: Elaborada pelo autor.

6.1.1 Acessando elementos de uma **list**

As **lists** usam o operador [] com o índice para pegar o valor do **index** passado entre o colchete. Dessa forma, caso se queira acessar o segundo item de uma **list**, deve-se passar como índice o valor 1, como mostra o exemplo do Código 2.

Código 2 – Acessando o segundo elemento da list.

```
1. lista = [1,1.5,True,"texto",[0,1]]
2. lista[1] = 3.0
3. print(lista[1])
```

Fonte: Elaborado pelo autor.

No Código 2 pode-se observar na linha 2 que o valor do segundo elemento é modificado (ele era 1.5 e tornou-se 3.0). E na linha 3 imprime o valor desse elemento, que será um texto com valor "3.0". Também pode-se observar que um dos elementos dessa **list** é uma outra **list**.

O programador pode copiar somente uma parte do conteúdo de uma lista (que em inglês se chama *slice*, que teria tradução de *fatiar*). Nesse caso, usa-se o mesmo padrão do **range**, podendo passar 2 ou 3 argumentos, como mostra o exemplo do Código 3.

Código 3 – Acessando sublistas usando o slice.

```
1. lista = [0,1,2,3,4,5,6,7,8]
2. lista4 = lista[0:4]
3. listaSliceLen = lista[4:len(lista)]
4. listaPares = lista[0:len(lista):2]
5. listaCopiada = lista[0:len(lista)]
6. listaCopy = lista.copy()
```

Fonte: Elaborado pelo autor.

No Código 3 pode-se verificar que a linha 2 contém todos os valores do índice 0 até o índice 4 da lista criada na linha 1, nesse caso ele retornará uma **list** com [0,1,2,3] que será armazenada na variável "lista4". Na linha 3 utiliza-se do mesmo **slice** com dois parâmetros, mas como primeiro parâmetro coloca-se o número 4, sendo assim o **slice** começará no índice 4, e irá até o final por passar como segundo parâmetro **len**(lista). A função **len** retorna o tamanho da lista passada como parâmetro, que no caso irá retornar 8, fazendo com que o retorno do **slice** seja uma **list** com valor [4,5,6,7,8]. E na linha utiliza-se o **slice** com 3 parâmetros, o primeiro indicando onde deve começar, o segundo onde deve terminar o **slice**, e o terceiro com o passo que ele deve fazer o **slice**, da mesma forma que o **range**. Sendo assim tem como parâmetros 0, **len**(lista) e 2, realizando um **slice** do começo até o fim da lista, pegando os valores dos índices somando o passo que é 2, retornando uma **list** [0, 2, 4, 6, 8] que é guardada na variável "listaPares". As linhas 5 e 6 realizam a mesma operação, na linha 5 é realizado um **slice** de todo o conteúdo da lista, copiando para a variável "listaCopiada", já na linha 6 é chamado o método **copy**, que copia todo o conteúdo para a variável "listaCopy".

Caso se tente acessar um **index** da **list** que não exista ou que seja maior que ela, um erro fatal em tempo de execução irá ocorrer.

6.1.2 Selecionando elementos do índice de uma **list** a partir de um valor

Também é possível pelo valor de um dos elementos requerer o índice dele na **list** usando o comando **index**. Nele é passado como parâmetro o valor da variável e o retorno dele é o índice desse valor na **list**. Um exemplo desse comando pode ser visto no Código 4.

Código 4 – Pegando o index de um elemento.

```
1. lista = [0,1,2,3,4,5,6,7,8]
```

```
2. lista.index(3)
```

Fonte: Elaborado pelo autor.

Nesse Código 4, na linha 2 irá pedir a lista que retornou o índice do elemento que tiver valor 3, que no caso é o elemento de índice 3, retornando o número 3.

Caso não exista objeto com esse valor, o programa retornará um erro do tipo "ValueError".

6.1.3 Inserindo novos valores

Existem diversas formas de inserir valores em uma **list**. Usando o comando **append**, que coloca o novo elemento no final da lista, usando o comando **insert**, que insere o novo elemento em um índice específico da lista, ou até mesmo concatenando com uma outra lista usando o operador +=. No Código 5 pode-se ver exemplos dessas adições.

Código 5 – Inserindo novos valores.

```
1. lista = [0,1]
2. lista.append(3)
3. lista.insert(0,4)
4. lista += [5]
```

Fonte: Elaborado pelo autor.

Pode-se verificar que ele cria uma **list** de nome **list** na linha 1 com os elementos [0,1]. Na linha 2 um outro elemento é adicionado ao final da **list** com o método **append**, resultando em uma **list** com os elementos [0,1,3]. Na linha 3 um elemento de valor 4 é inserido no índice 0, fazendo com que os outros elementos com índice maior que 0 tenham um índice acrescido de 1, resultando em uma **list** com os elementos [4,0,1,3]. Já na linha 4 a **list** é concatenada com outra **list**, que contém um elemento de valor 5, resultando em uma lista com elementos [4,0,1,3,5].

6.1.4 Removendo valores

Assim como a inserção, existem diversas formas de remover um valor de uma lista. Para isso, pode-se utilizar o método **remove**, que recebe como argumento o valor que deverá ser removido e retira da lista o primeiro elemento com valor igual ao do argumento. Um exemplo pode ser visto no Código 6.

Código 6 – Removendo um valor da lista.

```
1. lista = [0,1,0,2,3,0]
2. lista.remove(0)
```

Fonte: Elaborado pelo autor.

Nesse caso, na linha 2 será retirado o primeiro elemento da **list**, pois ele tem o valor igual ao do argumento (0), ficando como resultado a **list** com os valores [1,0,2,3,0]. Um detalhe desse comando é que, caso não haja um elemento com o valor igual ao do argumento, um erro ocorre.

Também pode-se utilizar o comando **pop**, que retira o elemento do índice igual ao do argumento. Caso nenhum argumento seja provido, ele retira o último elemento da lista. Um detalhe desse comando é que ele retorna o elemento a ser removido, que pode ser guardado em uma variável. Um exemplo desse método pode ser visto no Código 7.

Código 7 – Removendo elemento da lista.

```
1. lista = [0,1,0,2,3,0]
2. lista.pop(2)
3. ultimoElemento = lista.pop()
```

Fonte: Elaborado pelo autor.

No Código 7, na linha 1 é criada uma **list** com o nome de lista e elementos [0,1,0,2,3,0]. Na linha 2 o elemento de índice 2 é removido, com o método **pop** e argumento 2, que resulta na **list** com elementos [0,1,2,3,0]. Já na linha 3 o último elemento é removido com o método **pop** sem argumentos, resultando na **list** com elementos [0,1,2,3] e uma variável de nome "ultimoElemento", guardando o elemento que foi removido, que tem valor 0.

Para remover um elemento também pode-se utilizar do comando **del**, que deleta um elemento específico da memória. Nesse caso o elemento a ser deletado é passado em seguida do comando, como mostra o Código 8.

Código 8 – Removendo um elemento da lista.

```
1. lista = [0,1,0,2,3,0]
2. del lista[1]
```

Fonte: Elaborado pelo autor.

No Código 8, na linha 2, o comando **del** é chamado para deletar o elemento de índice 1 da **list**, tendo como resultado a lista com os elementos [0,0,2,3,0].

Também pode-se querer remover todos os elementos da **list** de uma só vez, resultando em uma **list** vazia []. Para isso existe o método **clear**, como mostra o Código 9.

Código 9 – Removendo todos os elementos da lista.

```
1. lista = [0,1,0,2,3,0]
2. lista.clear()
```

Fonte: Elaborado pelo autor.

6.1.5 Ordenamento de list

A **list** tem também outras funcionalidades que podem auxiliar o programador a desenvolver seus programas, como o método **sort**, que pode ser utilizado para ordenar uma **list**. Ele pode ser customizado por seus parâmetros, de forma a reverter a ordem dos elementos ou ter uma chave específica para realizar a ordenação. Um exemplo de seu uso pode ser visto no Código 10.

Código 10 – Ordenando uma list.

```
1. lista = [4,10,0,1,0,2,3,0]
```

```
2. lista.sort()
```

Fonte: Elaborado pelo autor.

No Código 10, na linha 2, o método **sort** é chamado de forma a ordenar os elementos da **list**, assim a **list** fica com os elementos [0,0,0,1,2,3,4,10].

6.1.6 Revertendo a ordem dos elementos de uma **list** com o método **reverse**

Pode ser necessário ter os elementos de uma lista revertida, isso é, colocados em ordem contrária, de trás para frente. Para isso é necessário usar o método **reverse**, como mostra o Código 11.

Código 11 – Revertendo os elementos de uma list.

```
1. lista = [4,10,0,1,0,2,3,0]
2. lista.reverse()
```

Fonte: Elaborado pelo autor.

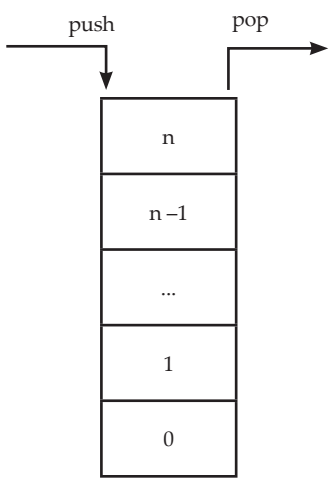
No Código 11, na linha 2, o método **reverse** é chamado de forma a colocar os elementos da **list** de trás para frente, dessa forma a **list** fica com os elementos [0,3,2,0,1,0,10,4].

Vídeo

6.2 Usando listas com o stack ou queue

Uma lista pode ter outras funcionalidades, além de guardar itens em uma ordem específica. Uma dessas funcionalidades é ser uma **pilha**, ou em inglês *stack*. Uma **pilha** é um tipo especial de lista em que todas as operações de inserção e remoção de elementos ocorre em sua extremidade, que é chamado de *topo*. A operação de inserção é chamada de **empilhar**, ou *push* pelo seu nome em inglês, e a operação de remover é chamada de **desempilhar**, ou *pop*, pelo seu nome em inglês. Uma ilustração dessa estrutura de dados pode ser vista na Figura 2.

Figura 2 – Uma pilha.



Fonte: Elaborada pelo autor.

Pilhas e listas são chamadas de *estruturas de dados*, pois são estruturas que auxiliam os programadores a organizarem os dados de seus programas. A pilha é chamada de uma estrutura do tipo lifo (*last-in first-out*). Uma pilha pode ser vista como uma lista com restrições, sendo assim para se colocar um novo item, somente se usa a operação **append**, que coloca o item como último da lista, e para retirar um item se utiliza do **pop** sem argumentos, que retira o último elemento, que será o último elemento que foi colocado. Um exemplo de uso de uma pilha pode ser visto no Código 12.

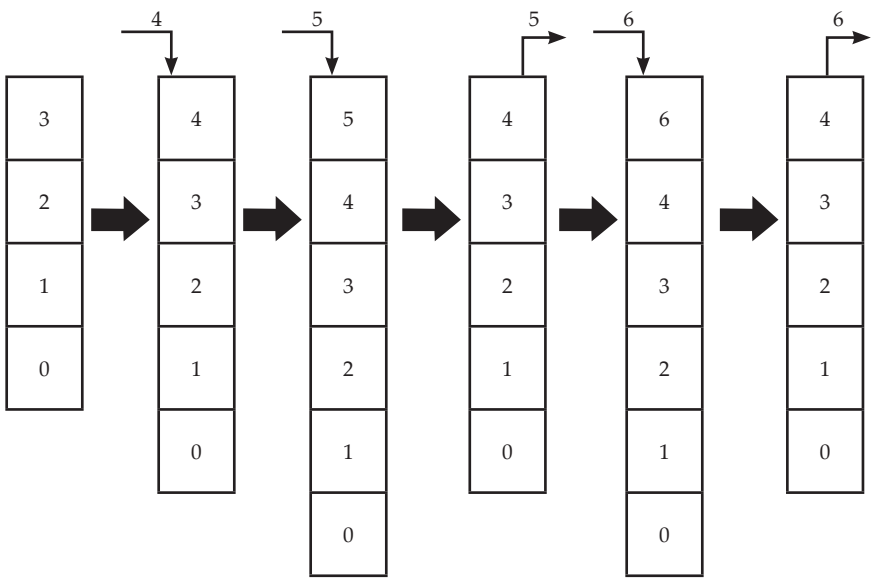
Código 12 – Usando uma pilha.

```
1. pilha = [0,1,2,3]
2. pilha.append(4)
3. pilha.append(5)
4. pilha.pop()
5. pilha.append(6)
6. pilha.pop()
```

Fonte: Elaborado pelo autor.

No Código 12 primeiramente a pilha é criada com os elementos [0,1,2,3] na linha 1. Na linha 2 o elemento com valor 4 é inserido no final, ficando com os elementos [0,1,2,3,4] na pilha. Na linha 3 um novo elemento com valor 5 é inserido no final, ficando com os elementos [0,1,2,3,4,5] na pilha. Na linha 4 o elemento do final é removido, ficando com os elementos [0,1,2,3,4]. Na linha 5 um elemento de valor 6 é inserido no final, ficando com os elementos [0,1,2,3,4,6]. E na linha 6 o último elemento é removido, ficando a **list** com os elementos [0,1,2,3,4]. Uma ilustração dessas operações pode ser vista na Figura 3.

Figura 3 – Ilustração das operações do Código 12.

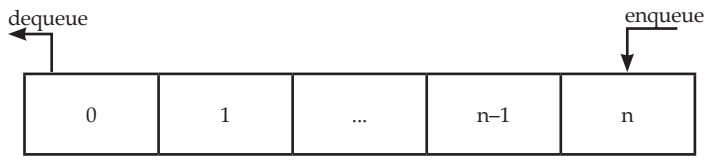


Fonte: Elaborada pelo autor.

6.2.1 A estruturas de dados **fila**

Outra estrutura de dados que também é bem importante é a fila, ou pelo seu nome inglês *queue*. Uma **fila** é um tipo especial de lista em que o primeiro elemento a ser inserido é o primeiro elemento a ser retirado, ou seja, sempre se adiciona itens ao fim e retira-se elementos do início. A operação de inserção é chamada de enqueue, e a operação de remover é chamada de dequeue. Uma ilustração dessa estrutura de dados pode ser vista na Figura 4.

Figura 4 – Ilustração de uma fila.



Fonte: Elaborada pelo autor.

A fila é chamada de uma estrutura do tipo **fifo** (*first-in first-out*). Uma **fila** pode ser implementada como lista com restrições, sendo assim para se colocar um novo item, somente se usa a operação **append**, que coloca o item como último da lista, e para retirar um item se utiliza do **pop** com argumento 0, retira o primeiro elemento, que será o primeiro elemento que foi colocado. Mas retirar o primeiro elemento de uma lista é uma operação custosa, por isso o Python tem uma implementação específica para a fila chamada **deque**. Um exemplo de uso de uma **pilha** pode ser visto no Código 13.

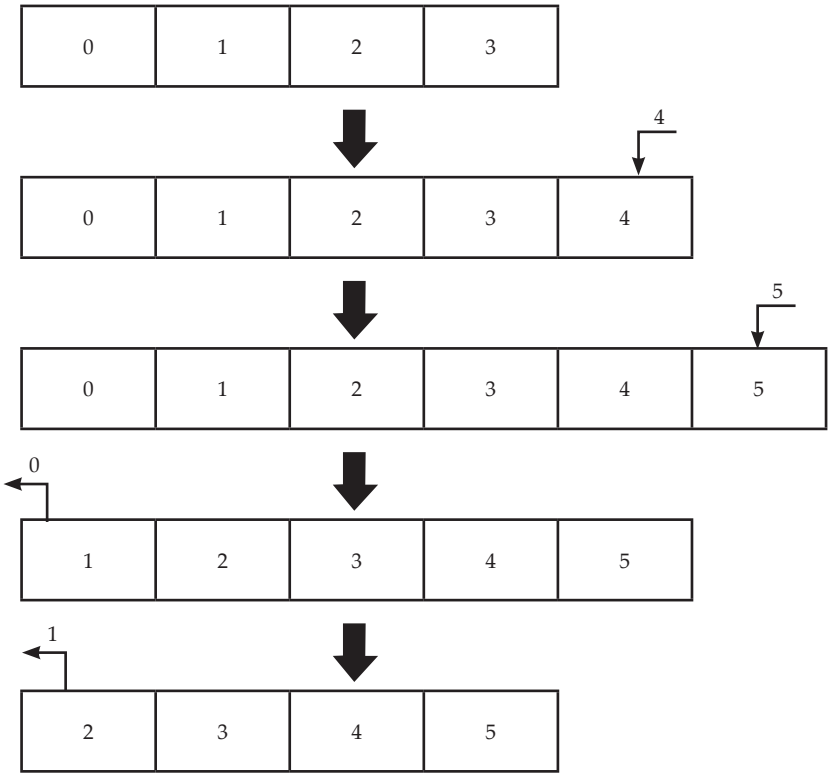
Código 13 – Usando uma fila.

```
1. from collections import deque
2. pilha = deque([0,1,2,3])
3. pilha.append(4)
4. pilha.append(5)
5. pilha.popleft()
6. pilha.popleft()
```

Fonte: Elaborado pelo autor.

No Código 13 primeiramente é importado o deque das **collections** (isso irá pegar a implementação do deque de outro arquivo, para se poder utilizar no código) na linha 1. Na linha 2 a fila é criada com os elementos [0,1,2,3]. Na linha 3 o elemento com valor 4 é inserido no final, ficando com os elementos [0,1,2,3,4] na fila. Na linha 4 um novo elemento com valor 5 é inserido no final da fila, ficando com os elementos [0,1,2,3,4,5]. Na linha 5 o elemento inicial é removido, com o método **popleft**, que é semelhante ao **pop**(0), ficando a fila com os elementos [1,2,3,4,5]. E na linha 6 o primeiro elemento é removido, ficando com a fila com os elementos [2,3,4,5]. Uma ilustração dessas operações pode ser vista na Figura 5.

Figura 5 – Ilustração das operações do Código 13.



Fonte: Elaborada pelo autor.

Vídeo

6.3 Compreensão de listas

Em Python, existe a compreensão de lista, que ajuda a criar lista com códigos mais sucintos e sem perda de legibilidade. Essa funcionalidade, apesar de muito poderosa, não é comum em outras linguagens de programação.

Para melhor apresentar esse conceito será apresentado um exemplo. A ideia de compreensão de listas é gerar uma **list** a partir de um código. No exemplo irá se criar uma lista com os valores da tabela de multiplicação de 2, no caso poderia se utilizar de um **for**, como mostra o Código 14.

Código 14 – Criando elementos de uma list a partir de um for.

```
1. lista = []
2. for i in range(10):
3.     lista.append(2*i)
```

Fonte: Elaborado pelo autor.

No Código 14 tem-se como resultado a lista com os elementos [0,2,4,6,8,10,12,14,16,18,20]. Mas poderia ter o mesmo resultado usando uma compreensão de lista, que faz a mesma operação em somente uma linha, em vez de 3, como mostra o Código 15.

Código 15 – Criando elementos de uma list a partir de uma compreensão de lista.

```
1. lista = [i*2 for i in range(11)]
```

Fonte: Elaborado pelo autor.

No Código 15 tem-se o mesmo resultado, o mesmo código feito em 3 linhas, tirando somente o comando do **append**. Dessa forma uma lista de compreensão consiste de parênteses contendo uma expressão seguido de um **for** com outros **for** e **ifs**. Um exemplo que cria uma **list** com os elementos iguais de duas listas com dois **for** e um **if** pode ser visto no Código 16.

Código 16 – Uma compreensão de lista com mais de um for e if.

```
1. lista = [x for x in [1,2,3,5] for y in [3,1,4,8] if x == y]
```

Fonte: Elaborado pelo autor.

Esse Código 16 pode ser desmembrado ficando com o Código 17.

Código 17 – Código 16 sem compreensão de lista.

```
1. lista = []
2. for x in [1,2,3,5]:
3.     for y in [3,1,4,8]:
4.         if x == y:
5.             lista.append(x)
```

Fonte: Elaborado pelo autor.

Pode-se observar que os Código 16 e Código 17 possuem os mesmos **for** e **if**.

6.3.1 Compreensões de listas aninhadas

Assim como pode-se ter listas aninhadas, que é uma lista de linhas, pode-se também ter compreensões aninhadas.

Como exemplo poderia ter uma matriz e querer a sua transposta. Para isso poderia se realizar o processo que mostra o Código 18.

Código 18 – Transpondo uma matriz.

```
1. matriz = [[1, 2, 3, 4],
2.           [5, 6, 7, 8],
3.           [9, 10, 11, 12]]
4. transposta = []
5. for i in range(4):
6.     linhaTransposta = []
7.     for linha in matriz:
8.         linhaTransposta.append(linha[i])
```

```
9.      transposta.append(linhaTransposta)
```

Fonte: Elaborado pelo autor.

No Código 18 tem-se nas linhas 1, 2 e 3 a criação da matriz e na linha 4 a criação da **list** chamada *transposta*, que irá guardar a matriz transposta. Na linha 5 tem um **for** que irá realizar o trabalho de criar as linhas transpostas em colunas, utilizando de uma **list** auxiliar e um **for** que irá fazer o processo de popular, a nova linha com as colunas da matriz original. Esse código tem como resultado uma **list** transposta com os elementos [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]. Este **for** que realiza população da nova linha transposta (linhas 6, 7, 8 e 9) poderia ser realizado por uma compreensão de listas, como mostra o Código 19.

Código 19 – Transpondo uma matriz usando uma compreensão de lista.

```
1. matriz = [[1, 2, 3, 4],
2.           [5, 6, 7, 8],
3.           [9, 10, 11, 12]]
4. transposta = []
5. for i in range(4):
6.     transposta.append([row[i] for linha in matriz])
```

Fonte: Elaborado pelo autor.

No código pode-se ver que é equivalente ao Código 18, e que ele poderia ter uma outra compreensão de lista, de forma a substituir as linhas 4, 5 e 6, como mostra o Código 20.

Código 20 – Transpondo uma matriz usando duas compreensões de lista.

```
1. matriz = [[1, 2, 3, 4],
2.           [5, 6, 7, 8],
3.           [9, 10, 11, 12]]
4. transposta = [[linha[i] for linha in matriz] for i in range(4)]
```

Fonte: Elaborado pelo autor.

Ampliando seus conhecimentos

Documentação de software

(DE SOUZA, et al. 2007)

Documentação de software pode ser definida como um artefato cuja finalidade seja comunicar a informação sobre o sistema de software ao qual ele pertence [5]. Entretanto, é necessário distinguir entre modelos, documentos, código fonte e documentação. Segundo Ambler [1], do ponto de vista da modelagem ágil, um documento é qualquer artefato externo ao código fonte cujo propósito seja transmitir informação de uma maneira

persistente. Modelo é uma abstração que descreve um ou mais aspectos de um problema ou de uma solução potencial para resolver um problema. Alguns modelos podem se tornar documentos, ou incluídos como parte deles, ou simplesmente serem descartados quando cumprirem seu papel. Código fonte é uma seqüência de instruções, incluindo os comentários que descrevem estas instruções, para um sistema de computador. O termo documentação inclui documentos e comentários de código fonte.

A documentação tem fundamental importância para a Engenharia de Software. Vários estudos tem sido realizados para minimizar os problemas em torno da documentação de software: documentação desatualizada e de baixa qualidade, processos de documentação dispendiosos e caros, documentação em abundância e sem propósito, dificuldade de acesso, entre outros. Alguns pesquisadores propuseram o uso de hipertexto para facilitar o acesso a documentação. Tilley e Müller [17] propuseram combinar documentação e código fonte de uma maneira fácil e eficiente, utilizando ferramenta de hipertexto. Freeman e Munro [6] propuseram um sistema interativo em hipertexto com gráficos e texto. Rajlich [13] propôs a adoção de uma estratégia de redocumentação incremental e oportunística na qual a compreensão obtida na manutenção é armazenada, por meio de anotações, em um sistema de hipertexto baseado na Web. Tilley e Huang [18] definiram um modelo de maturidade para documentação de software com foco no produto (DMM – Documentation Maturity Model), baseado no CMM (Capability Maturity Model) do SEI (Software Engeneering Institute), para estabelecer parâmetros de avaliação da qualidade da documentação. Ouchi [9] definiu uma estrutura de dados úteis para subsidiar um sistema de documentação voltado para manutenção de software e ressaltou a importância de métodos padronizados de documentação. Medina [10] descreveu os aspectos importantes para uma boa documentação e organização interna dos programas, como por exemplo, a utilização de comentários, identação dos comandos, padronização de nomes de variáveis e espaçamento para enfatizar a estrutura lógica, no sentido de auxiliar o entendimento do programa. O HCi Journal [7] ressaltou que antes do início de cada projeto, se faz necessário um planejamento da documentação a ser utilizada, pois todo desenvolvimento tem características peculiares. Tilley [16] propôs um método flexível de identificação, documentação, representação e apresentação dos aspectos estruturais da arquitetura do software por meio da engenharia reversa chamada de "documenting-in-the-large". A norma ANSI/ANS 10.3-1995 recomendou

a divisão da documentação em quatro categorias: resumo, informação da aplicação, definição das funcionalidades e informação do programa [11]. Lethbridge et al. [8] indicou a necessidade de empenhar na produção de um simples e poderoso formato e ferramenta de documentação, ao invés de forçar os engenheiros de software realizar um trabalho caro e ineficaz. Cioch et al. [4] propuseram uma abordagem de documentação que considera o tipo de informação necessária para cada estágio de aprendizagem (recém- chegado, aprendiz, interno e experiente).

Pode-se notar que os estudos sobre documentação de software visam solucionar problemas de falta de atualização, dificuldade de acesso, falta de qualidade, desorganização, documentação desnecessária e indicam para soluções simples, com o apoio de ferramentas e que sirvam a um propósito.

Atividades

1. Escreva um programa que recebe como entrada do usuário uma lista de 5 números inteiros e cria uma lista com os mesmos números multiplicados por 5.

2. Escreva um programa que recebe como entrada do usuário uma lista de 10 números inteiros e somente coloca na lista dos números não duplicados com algum número já colocado pelo usuário.

3. Escreva um programa que recebe como entrada do usuário uma lista de 10 números e retorna o maior número da lista.

4. Escreva um programa que dada duas listas ele retorna uma lista com os itens que não existem nas duas.

5. Escreva um programa que usa uma compreensão de lista para criar uma **list** com todos os números pares menores que 100.

6. Escreva um programa que usa uma compreensão de lista para criar uma **list** com todos os números primos menores que 50.

Referências

DE SOUZA, Sérgio Cozzetti Bertoldi, et al. **Documentação essencial para manutenção de software II**. IV Workshop de Manutenção de Software Moderna (WMSWM), Porto de Galinhas, PE. 2007.

Resolução

1.

```
1. lista = []
2. for i in range(5):
3.      lista.append( 5* int(input("Entre com um numero:")))
4. print(lista)
```

2.

```
1. lista = []
2. for i in range(10):
3.      numero = int(input("Entre com um numero:"))
4.      existe = False
5.      for num in lista:
6.           if numero == num:
7.              existe = True
8.              break
9.           if not existe:
10.             lista.append( numero)
11. print("Lista ",lista)
```

3.

```
1. lista = []
2. for i in range(10):
3.      numero = int(input("Entre com um numero:"))
4.      lista.append( numero)
5.
6. maiorNumero = lista[0]
7. for numero in lista:
8.     if numero > maiorNumero:
9.         maiorNumero = numero
10. print("Maior numero",maiorNumero)
```

4.

```
1. lista1 = [3,4,5,6,7]
2. lista2 = [0,1,2,3,4]
3. lista3 = []
```

```
4. for i in lista1:
5.     existe = False
6.     for j in lista2:
7.         if i == j:
8.             existe = True
9.             break
10.        if not existe:
11.            lista3.append(i)
12.for i in lista2:
13.     existe = False
14.     for j in lista1:
15.         if i == j:
16.             existe = True
17.             break
18.         if not existe:
19.             lista3.append(i)
20.print(lista3)
```

5.

```
1. lista = [x for x in range(100) if x%2 == 0]
```

6.

```
1. naoPrimos = [j for i in range(2, 8) for j in range( i*2, 50, i)]
2. primos = [x for x in range(2, 50) if x not in naoPrimos]
```

7
Funções

Neste capítulo será introduzido o conceito de funções. O leitor já pode usar funções do sistema, passando ou não atributos. Este capítulo proverá os meios para que o leitor possa criar suas próprias funções e ver as vantagens de fazer isso. Uma delas é o aproveitamento do código, que permite ao programador reutilizar sequências de código, e separar também as sequências de código, que o torna mais legível e portável. Este capítulo se concentra em apresentar as principais características e vantagens das funções na seção 1. Na seção 2 apresenta-se o uso de funções com argumentos e como utilizar os argumentos padrões. E finalmente na seção 3 apresenta-se como criar funções com **lambda**, como o escopo funciona e como se deve comentar as funções, de forma que o código possa ser utilizado por outros programadores.

Vídeo

7.1 Funções

Os códigos desenvolvidos até agora foram pequenos, mas caso seja necessário criar algo mais robusto, sem funções, será difícil, caso se queira repetir o código em outra parte do programa poderia se copiar as linhas, mas caso tenha de mudar algo nele, o programador teria que refazer em vários trechos do programa. Para esses casos o melhor é usar funções. Isso é colocado como reusabilidade do código. Dentre as vantagens de reutilizar o código, tem-se: redução do tempo de desenvolvimento, redução dos erros de programação, aumento da estabilidade do programa, compartilhamento de código, facilidade de entendimento do código e aumento da eficiência do programa.

Esta seção se concentrará em mostrar as funções. As funções são comuns em Python e em outras linguagens de programação. Elas podem ser definidas como agrupamentos de partes de códigos, que podem ser chamadas quando o programador precisar executar o trecho do código. Sendo assim, funções podem ser vistas como miniprogramas dentro do programa.

As funções possuem um nome, que é a forma que ela será chamada para executar o trecho de código definido na função. Algumas são incluídas com os módulos do Python, como o **print**. Um módulo é um arquivo que contém definições de funções e classes (que infelizmente está fora do escopo desta obra, mas você terá oportunidade de aprender futuramente durante o curso). Dessa forma, o programador pode criar e compartilhar suas funções dentro de um arquivo.

Para definir uma função, usa-se o comando **def**, seguindo do nome da função e dentro de parênteses, se coloca os argumentos, ou nada caso a função não tenha argumentos. Uma função tem o formato como mostra o Código 1.

Código 1 – A função print com seus argumentos.

```
1. def nome-da-função(lista-de-argumentos):
2.     corpo da função
```

Fonte: Elaborado pelo autor.

Um exemplo de uma definição de uma função pode ser visto no Código 2.

Código 2 – Exemplo de uma definição de função.

```
1. def PrimeiraFunc():
2.     print(“Esta é minha primeira função.”)
3.     print(“Ela é bem simples.”)
4.     print(“Somente imprime algumas coisas.”)
```

Fonte: Elaborado pelo autor.

No Código 2, pode-se ver a primeira função criada. Nesse código, na linha 1 tem-se a definição do nome da função, que é “primeiraFunc”. Na linha 2 até a 4 tem o corpo da função, em que se imprime 3 frases. Um exemplo de chamar a função primeira “primeiraFunc” 3 vezes pode ser vista no Código 3 e seu resultado no Código 4.

Código 3 – Chamando a mesma função definida no Código 2, por 3 vezes.

```
1. primeiraFunc()
2. primeiraFunc()
3. primeiraFunc()
```

Fonte: Elaborado pelo autor.

Código 4 – Saída do Código 3.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6
2. Esta é minha primeira função.
3. Ela é bem simples.
4. Somente imprime algumas coisas.
5. Esta é minha primeira função.
6. Ela é bem simples.
7. Somente imprime algumas coisas.
8. Esta é minha primeira função.
9. Ela é bem simples.
10. Somente imprime algumas coisas.
11. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

O mesmo Código 3 poderia ser feito sem uma função, como mostra o Código 5.

Código 5 – Código feito em 3 sem uso de funções.

```
1. print("Esta é minha primeira função.")
2. print("Ela é bem simples.")
3. print("Somente imprime algumas coisas.")
4. print("Esta é minha primeira função.")
5. print("Ela é bem simples.")
6. print("Somente imprime algumas coisas.")
7. print("Esta é minha primeira função.")
8. print("Ela é bem simples.")
9. print("Somente imprime algumas coisas.")
```

Fonte: Elaborado pelo autor.

No Código 5, pode-se ver que ele possui três vezes mais linhas do que o Código 3, mas possui o mesmo resultado da Figura 1. Como dito anteriormente, essa é apenas uma das vantagens de se utilizar as funções.

7.1.1 Argumentos de funções

Apesar do exemplo do Código 2 ser útil, pois ele elimina a necessidade de ter que digitar todo o corpo da função toda vez que se deseja imprimir as frases definidas, ele é limitado, pois somente deixa fazer uma coisa, e funções devem ser flexíveis de forma a permitir que o usuário realize mais de uma coisa. Se não, o programador pode ter que escrever diversas funções que variam de acordo com a corpo que ele realiza, em vez de variar de acordo com a funcionalidade. Dessa forma, o uso de argumentos provê maior flexibilidade nas funções.

O termo **argumento** já foi comentado nos capítulos anteriores, de forma a introduzir o assunto ao leitor. Um argumento seria um dado que é recebido pela função para utilizar em seu corpo. Como exemplo, temos o Código 6.

Código 6 – Exemplo de função com argumento.

```
1. def ola(nome):
2.     print("Ola, "+nome)
3.     print("Como você vai?")
```

Fonte: Elaborado pelo autor.

No Código 6, na linha 1, pode-se ver que além de definir o nome da função, que no caso é "Ola", coloca-se entre parênteses o argumento, que no caso é "nome". Esse argumento é usado na linha 2, para concatenar junto da **string** "ola, ". Um exemplo da função pode ser visto no Código 7.

Código 7 – Chamando a função do Código 6 com argumento "Mark".

```
1. ola("Mark")
```

Fonte: Elaborado pelo autor.

Caso o programador chame a função "Ola", sem argumentos, como mostra o Código 8, um erro em tempo de execução é ocasionado, pois a função tem o argumento **nome** como requerido, como mostra o Código 9.

Código 8 – Chamando a função do Código 6 com argumento "Mark".

```
1. ola()
```

Fonte: Elaborado pelo autor.

Código 9 – Erro ao chamar o Código 8.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6"/Users/
BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap2/code/Cap5.py"
2. Traceback (most recente call last):
3.   File "/Users/BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap2/code/Cap5.py",
      line 12, in <module> ola ()
4. TypeError: ola() missing 1 required positional argument: 'nome'
5. Process finished with exit code 1
```

Fonte: Elaborada pelo autor.

7.1.2 Retorno de uma função

Uma função também pode retornar um valor ou expressão quando ela terminar, usando o comando **return**. O formato da função com retorno pode ser visto no Código 10.

Código 10 – Formato de uma função com retorno.

```
1. def nome-da-função(lista-de-argumentos):
2.     corpo da função
3.     return expressão
```

Fonte: Elaborado pelo autor.

Um exemplo de função que recebe um argumento e retorna ele acrescido de 2 pode ser visto no Código 11.

Código 11 – Função acrescida de 2.

```
1. def adicionaDois(numero):
2.     numero += 2
3.     return numero
```

Fonte: Elaborado pelo autor.

No Código 11, na linha 1 se define uma função com nome de "adicionaDois", e que requer argumento de nome "número". Na linha 2 adiciona-se 2 ao argumento passado e na linha 3 retorna esse valor. Também pode-se colocar o **return** sem ter valor, como mostra o Código 10. Nesse caso tem a mesma funcionalidade que não ter **return**.

Código 12 – Função com return vazio.

```
1. def ola(nome):
2.     print("Ola, "+nome)
3.     print("Como você vai?")
4.     return
```

Fonte: Elaborado pelo autor.

Também pode-se utilizar o **return** dentro de um condicional para criar um retorno sem executar a sequência do código, como mostra o Código 13.

Código 13 – Função com return condicional.

```
1. def ola(nome):
2.     print("Ola, "+nome)
3.     if(nome != "Mark"):
4.         return
5.     print("Como você vai?")
```

Fonte: Elaborado pelo autor.

Vídeo

7.2 Funções com múltiplos argumentos

Uma função pode ter múltiplos argumentos. Dessa forma se separa a lista de argumentos por vírgulas, como mostra o exemplo do Código 14.

Código 14 – Função com dois argumentos.

```
1. def multiplica(numero1,numero2):
2.     return numero1 * numero2
```

Fonte: Elaborado pelo autor.

Para chamar a função do Código 14, o programador tem duas opções: colocar os argumentos ordenados, como foram definidos pela função, chamado de parâmetro posicional ou colocando o valor de cada variável, na ordem que preferir, chamado de parâmetro por *keyword*. Um exemplo dessas chamadas pode ser visto no Código 15.

Código 15 – Exemplo de formas de colocar os argumentos em uma chamada de função.

```
1. multiplica(3,4)
2. multiplica(numero2=4,numero1=3)
```

Fonte: Elaborado pelo autor.

No Código 15, na linha 1, a função **multiplica** é chamada, passando seus argumentos em ordem 3 e 4, dessa forma o "numero1" terá valor 3 e o "numero2" terá valor 4. Já na linha 2, tem um código que é equivalente ao da linha 1. Nesse caso ela coloca os valores pelo seu nome, pois a ordem dos argumentos não importa.

7.2.1 Argumentos com valor padrão

Até o momento, todas as funções que tinham argumento requereram que fosse passado um valor para elas. Mas as funções podem ter um valor padrão, caso não seja passado nenhum argumento. Dessa forma, erros de falta de valor não são ocasionados. Para se criar um argumento com valor padrão, somente deve-se prover um valor na definição do valor, como mostra o formato com dois argumentos no Código 16.

Código 16 – Formato de uma função com argumentos com valor padrão.

```
1. def nome-da-função(argumento1=valor1,argumento2=valor2):
2.     corpo da função
3.     return expressão
```

Fonte: Elaborado pelo autor.

Um exemplo de função com argumento padrão pode ser visto no Código 17.

Código 17 – Definindo e chamando uma função com dois argumentos com valor padrão.

```
1. def multiplica(numero1=3,numero2=3):
2.     return numero1 * numero2
3. multiplica()
```

Fonte: Elaborado pelo autor.

No Código 17, na linha 1 pode-se verificar que a função é definida com os valores padrão 3 e 3 para os dois argumentos. Dessa forma, a função pode ser chamada sem argumentos, como é feito na linha 3.

Se na ordem dos argumentos um for definido com valor padrão, todos os posteriores devem também ter um valor padrão ou irá ocorrer um erro de sintaxe.

7.2.2 Funções com número variável de elementos

As funções também possuem o poder de ter múltiplos argumentos, em que se passa uma sequência de argumentos que se quer utilizar na função. Para isso, o Python se utiliza do caractere * antes do nome da variável para indicar que ele terá uma lista de argumentos. Um exemplo de função pode ser visto no Código 18.

Código 18 – Definindo e chamando uma função com múltiplos argumentos.

```
1. def somaNumeros(*varArgs):
2.     soma = 0
3.     for arg in varArgs:
4.         soma += arg
5.     return soma
6. somaNumeros(3,1,2,3)
```

Fonte: Elaborado pelo autor.

No Código 18, tem-se o código que define e chama uma função, recebe vários argumentos e retorna a soma deles. Na linha 1, define-se a função que terá nome "somaNumeros", e o argumento "varArgs" precedido do caractere * para indicar que são múltiplos argumentos. Na linha 2, 3 e 4 realiza-se a soma dos argumentos de entrada e na linha 5 o retorno dessa soma. Na linha 6 tem-se a chamada dessa função, com quatro argumentos.

Vídeo

7.3 Definindo expressões com lambda

Em Python, existe a uma expressão **lambda**, que define um bloco de código que pode ser colocado em uma variável. O nome **lambda** vem do alfabeto grego, e é usado na matemática para indicar uma função anônima.

O **lambda** tem uma funcionalidade e sintaxe parecida com a da compreensão de lista, em que todo o código é praticamente definido em somente uma linha. Essa funcionalidade é muito poderosa, pois podem ser chamadas pela variável e até mesmo passadas como parâmetros de uma função. O **lambda** tem o formato apresentado no Código 19.

Código 19 – Formato de uma expressão lambda.

```
1. lambda lista-de-argumentos: expressão-a-ser-retornada
```

Fonte: Elaborado pelo autor.

Sendo assim, o **lambda** tem uma lista de argumentos, separadas por vírgulas, e retorna uma expressão. Um exemplo de **lambda** que multiplica dois números e depois é chamada pode ser visto no Código 20.

Código 20 – Criando e chamando uma expressão lambda.

```
1. f = lambda x,y: x*y
2. f(2,4)
```

Fonte: Elaborado pelo autor.

No Código 20, na linha 1, é criada uma função **lambda** que recebe como argumento x e y, e retorna a multiplicação entre os argumentos.

7.3.1 Expressões **lambda** com argumentos padrões

Assim como as funções, as expressões **lambda** também possuem a característica de poder ter argumentos com valor padrão, de forma a evitar erros e facilitar a vida dos programadores. Um exemplo pode ser visto no Código 21.

Código 21 – Criando e chamando uma expressão lambda com o segundo argumento com valor padrão.

```
1. f = lambda x,y=3: x+y
2. f(1)
```

Fonte: Elaborado pelo autor.

As expressões **lambda** também permitem a colocação de valores nos argumentos de forma explícita, como mostra o Código 22.

Código 22 – Criando e chamando uma expressão lambda com valores de argumentos explícitos.

```
1. f = lambda x,y,w: w*(x+y)
2. f(w=2,x=1,y=4)
```

Fonte: Elaborado pelo autor.

7.3.2 Escopo

Antes de uma variável ser utilizada, o Python deve achar o identificador com o nome da variável, e com isso determinar o valor associado com esse identificador. **Namespace** é quem guarda esses identificadores e os valores a qual eles estão associados. Essa busca pelos **namespaces** pode ser local, global e do sistema. As funções se utilizam de um escopo local, dessa forma seus argumentos são variáveis locais. Como exemplo tem-se o Código 23.

Código 23 – Escopo de funções.

```
1. def tres():
2.     x = 3
```

```
3. x = 2
4. tres()
5. print(x)
```

Fonte: Elaborado pelo autor.

Nessa função temos exemplificado o processo de escopo. Por exemplo, o leitor poderia achar que o código a seguir na linha 5 iria imprimir 3, mas não, ela imprime 2, pois a variável x, definida na função na linha 2, tem escopo local, sendo uma variável diferente da declarada na linha 3. Dessa forma, na linha 2 é criada uma nova variável de nome x, mas de escopo local, sendo assim os nomes de variáveis definidas dentro de funções não afetam os nomes definidos dentro do seu programa. Um outro exemplo pode ser visto no Código 24.

Código 24 – Outro exemplo de escopo de funções.

```
1. def somaTres(x):
2.     x += 3
3. x = 2
4. y = 3
5. tres(y)
6. print(x)
```

Fonte: Elaborado pelo autor.

No Código 24, apesar de a função usar como variável local de nome x, ele recebe como parâmetro a variável global y, dessa forma y tem seu valor modificado localmente, tornando 6 somente na cópia local, mantendo 3 na variável global. Já na linha 6, em que se imprime o valor de x, mantém-se o mesmo da linha 3.

7.3.3 Documentando as funções

O Python provê uma documentação especial para as funções chamado de **documentation strings**, ou **docstrings**, que é uma maneira conveniente de associar a documentação com as funções Python. É como um comentário normal, mas que pode ser chamado com o atributo __doc__ depois do nome de uma função. Um exemplo pode ser visto no Código 25.

Código 25 – Exemplo de docstring de funções.

```
1. def somaTres(x):
2. ‹››Soma tres a variavel passada como entrada. ‹››
3.         x += 3
4. print(somaTres.__doc__)
```

Fonte: Elaborado pelo autor.

Nesse exemplo, logo depois da definição da função, na linha dois, uma frase comentada é escrita, descrevendo como funciona a função. Quando essa função é chamada seguida do __doc__, como acontece na linha 4, ela retorna a **string** do comentário da linha 2.

Ampliando seus conhecimentos

A visão da função na metodologia de programação

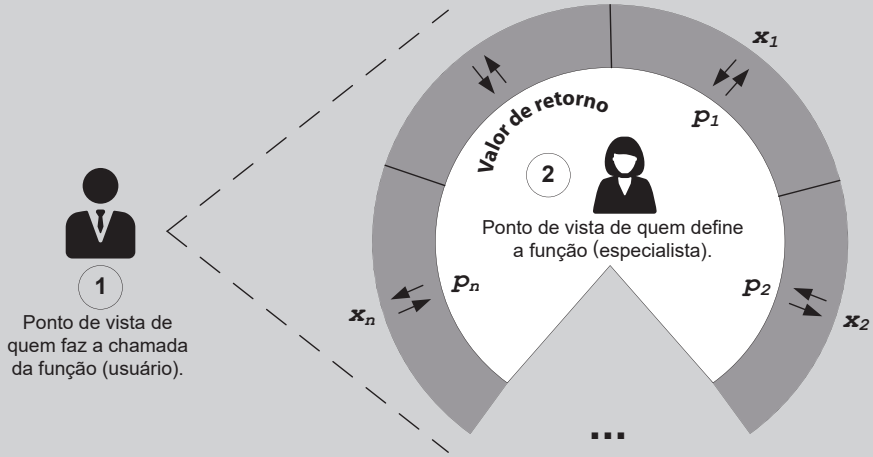
(BARROS, 2006)

A complexidade de um programa pode ser dominada através da Programação Estruturada, uma metodologia que se compõe de quatro princípios básicos:

- princípio da abstração é a concepção ou visão do programa separado de sua realidade. É a simplificação de fatos, descrevendo o que está sendo feito sem explicar como está sendo feito;
- princípio da formalidade possibilita analisar os programas de forma matemática. Fornece uma abordagem rigorosa e metódica. Possibilita a transmissão de ideias e instruções sem ambiguidades e permite que estas sejam automatizadas;
- princípio da divisão é a subdivisão organizacional de um programa em um conjunto de partes menores e independentes, mais fáceis de serem entendidas, resolvidas, manipuladas e testadas individualmente;
- princípio da hierarquia é a organização hierárquica que está relacionada com o princípio da divisão. A organização das partes em uma estrutura hierárquica do tipo árvore sempre aumenta a compreensibilidade.

Começando com uma função geral e abstrata do que o programa deve fazer, este é dividido em várias funções também abstratas. Qualquer uma destas funções pode, de forma hierárquica, ser dividida em mais funções abstratas e assim sucessivamente. A metodologia termina por chegar a um nível tal de formalidade no qual a função geral pode ser implementada no computador.

As funções aceitam entradas e saídas de forma a generalizar o processo aplicando-se a quaisquer dados de tipos determinados. As entradas são os argumentos e dados globais usados no procedimento ou função. As saídas são o valor de retorno (as funções podem ou não possuir valor de retorno), modificações feitas através de ponteiros e referências, bem como mudanças em dados globais (Figura 1).



As entradas e as saídas desempenham o papel da interface para as funções. O usuário só precisa entender a interface para fazer uso das funções. A sequência de funções é um detalhe oculto. Por isso, e de forma geral, consideramos uma função como um único elemento abstrato.

Tomemos como exemplo a função Ordena cuja entrada é uma lista de três parâmetros inteiros e cuja saída é a mesma lista em ordem crescente. Assim, se x=7, y=-2 e z=4 teremos, após executar o procedimento Ordena (x,y,z), x=-2, y=4 e z=7. Outro exemplo é a função Maior cuja entrada são dois parâmetros reais e cuja saída é o valor de retorno maior entre os dois. Assim Maior (3,4)=4, Maior(7,-2)=7, etc.

Atividades

1. Crie uma função que recebe uma temperatura de 0 a 100 em Celsius e retorna seu valor em Fahrenheit.

2. Um inteiro maior que 1 é primo se ele for divisível somente por 1 e por si mesmo. Sabendo disso, escreva uma função que recebe um número, e retorna **True** se ele for primo, e **False** se ele não for.

3. Um ano bissexto é um ano que é divisível por 4 e não é divisível por 100. Sabendo disso, escreva uma função que recebe um ano, e retorna **True** se ele for bissexto, e **False** se ele não for.

4. Faça uma função que recebe o preço de um produto e retorna com desconto de 9%.

5. Faça uma função que recebe uma lista de notas e retorna a média delas.

Referências

BARROS, E. A; PAMBOUKIAN, S.; ZAMBONI, L. C. Ensinando programação através de funções. WCCSETE-World Congress on Computer Science, Engineering and Technology Education. **Anais**... 2006. Disponível em: <http://meusite.mackenzie.br/edsonbarros/publicacoes/Artigo_658_WCCSETE2006.doc>. Acesso em: 8 jul. 2017.

Resolução

1.

```
1. def fahrenheit(celsius):
2.     return celsius * 1.8 + 32
```

2.

```
1. def primo(numero):
2.     contador = 0
3.     divisor = 1
4.     while divisor <= numero:
5.        if numero % divisor == 0:
6.           divisor += 1
7.           contador += 1
8.        else:
9.           divisor += 1
10.    if contador == 2:
11.       return True
12.    else:
13.       return False
```

3.

```
1. def bissexto(ano):
2.     if ano%4 == 0 and ano%100 != 0:
3.        return True
4.     else:
5.        return False
```

4.

```
1. def desconto(preco):
2. return preco * 0.91
```

5.

```
1. def media(notas):
2.     soma = 0
3.     for nota in notas:
4.         soma += nota
5.     return soma / len(notas)
```

8
Projeto completo

Neste capítulo será realizado um projeto completo, utilizando os conceitos apresentados nos capítulos anteriores. Dessa forma, o leitor poderá ver como trabalhar com o que foi aprendido. Este capítulo utilizará de condições, laços, listas e funções. A ideia é que o leitor ganhe experiência em realizar um projeto maior e conhecer como transformar uma ideia em algoritmo. Este capítulo irá propor a construção de um jogo da velha contra um computador. Dessa forma, apresenta na seção 1 os dados sobre o projeto. Na seção 2 são apresentadas todas as funções que serão usadas pelo jogo, e na seção 3 é apresentado o jogo completo.

Vídeo

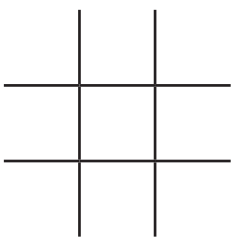
8.1 Introdução ao projeto

Até agora foram vistos somente pequenos trechos de código. Para um projeto um pouco maior, temos de organizar melhor o pensamento e o código desenvolvido, pensando melhor como serão as estruturas e as funcionalidades utilizadas nele.

8.1.1 Sobre o projeto a ser desenvolvido

O jogo da velha pode também ser conhecido como jogo do galo em Portugal e como Tic Tac Toe nos Estados Unidos. Ele é um jogo extremamente popular e bem simples de se aprender. O jogo é sempre com dois jogadores identificados por "X" ou círculo "O". O jogo é realizado sobre um tabuleiro, que também pode ser desenhado sobre um papel, e tem 9 posições, como mostra a Figura 1.

Figura 1 – Tabuleiro do jogo da velha.



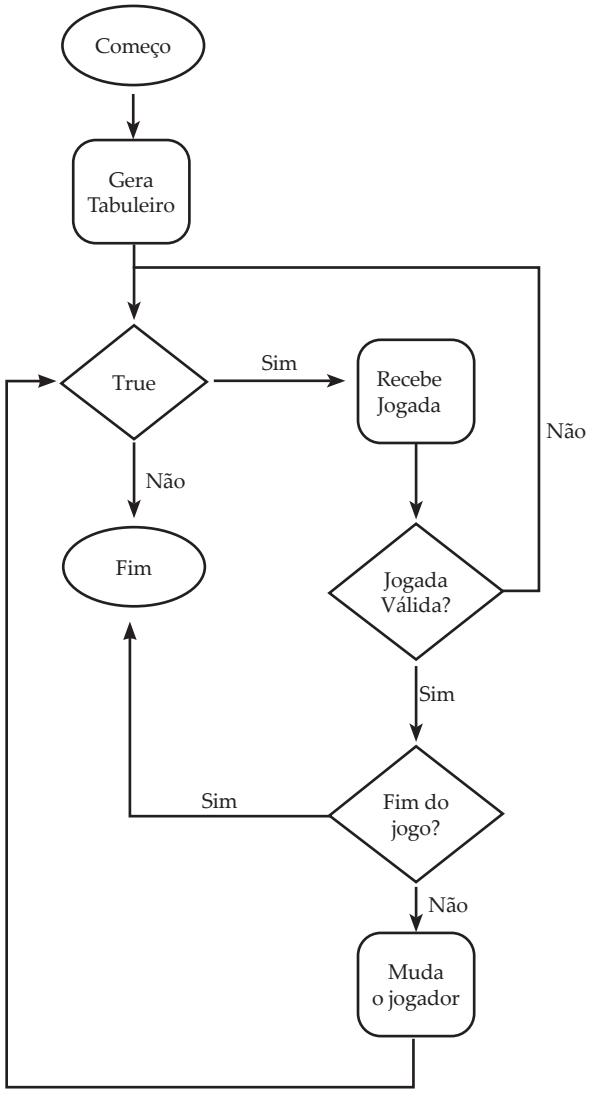
Fonte: Elaborada pelo autor.

O jogo é realizado em turnos, cada jogador coloca o símbolo (X/O) em um lugar vago do tabuleiro. O jogo termina quando um jogador fizer 3 símbolos em sequência, o jogador que fizer a sequência ganha, ou se não sobrarem espaços vazios, dá empate, ou como normalmente se diz "deu velha".

8.1.2 Lógica do jogo da velha

Antes de entrar no código, é necessário definir como será o fluxo do programa. Nesse caso, serão usadas as regras do jogo para definir o fluxo. Será necessário perguntar onde o jogador deseja colocar a pedra, verificar se a pedra pode ser colocada na posição desejada, e de acordo com essa nova posição deve-se verificar se o jogo acabou, isso é, deu empate ou se alguém ganhou, caso contrário, muda o jogador e refaz as ações. Esse fluxo pode ser visto na Figura 2.

Figura 2 – Fluxo do jogo da velha.



Fonte: Elaborada pelo autor.

Nesse fluxo, a primeira condição (o losango com "**True"**) é um *loop* infinito, isso é, ele roda para sempre. E a segunda condição (losango com "Fim do jogo") realiza um *break* que irá para o fim do programa caso tenha sido realizado o fim do jogo, isto é, tenham ocorrido 9 jogadas, ou alguém conseguiu formar 3 símbolos em sequência.

Vídeo

8.2 Implementando o projeto

Esta seção se concentrará em implementar o jogo da velha. Primeiramente precisa-se determinar como serão as estruturas de dados que irão guardar os valores que estão no tabuleiro. Existem diversas formas de realizar essa tarefa. No caso será utilizado uma **list** para guardar esse tabuleiro. Ele será de inteiros, em que 0 representa o espaço vazio, 1 representa o espaço ocupado por "X" e 2 representa o espaço ocupado por "O". Usar inteiros para isso facilitará verificar e também mostrar o tabuleiro ao jogador, como será visto adiante no capítulo. O exemplo de como os índices serão colocados de acordo com a posição pode ser visto na Figura 3.

Figura 3 – As posições do tabuleiro e seus índices na list.

0	1	2
3	4	5
6	7	8

Fonte: Elaborada pelo autor.

Dessa forma, o mapeamento do tabuleiro pode ser feito pelos índices de uma **list**. No Código 1 é criada uma função para gerar um tabuleiro com todas as posições vazias e armazenar esse tabuleiro em uma variável de nome "tabuleiro". Nesse caso será uma **list** com 9 posições com o valor 0 em todos seus elementos. O Código 1 mostra a função e sua chamada.

Código 1 – Criação do tabuleiro.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. tabuleiro = novoTabuleiro()
```

Fonte: Elaborado pelo autor.

No Código 1 se define na linha 1 a função "novoTabuleiro", que não recebe argumentos, e retorna na linha 2 uma **list** com todos os seus valores zerados. Esse retorno está dividido em 3 linhas de forma a melhorar a visualização de cada uma do tabuleiro. Na linha 6 uma variável de nome "tabuleiro" é inicializada com o retorno dessa função definida na linha 1.

8.2.1 Imprimindo o tabuleiro

De forma a facilitar a visualização do jogador, deve-se mostrar como está o tabuleiro. O ideal seria mostrar as posições vazias com a posição que o jogador quer colocar o seu símbolo, que seria o número do índice da **list**. Também, para facilitar a visualização do jogador,

deve-se imprimir em 3 colunas e 3 linhas o tabuleiro. Uma versão dessa função pode ser vista no Código 2.

Código 2 – Criando e imprimindo o tabuleiro.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.            print(" ",indice,sep="",end='')
10.    elif valor == 1:
11.           print(" X", end='')
12.    else:
13.           print(" O", end='')
14.
15.    if indice == 2 or indice == 5:
16.           print("\n", end='')
17.
18. print("\n")
19.
20. tabuleiro = novoTabuleiro()
21.
22. imprimirTabuleiro(tabuleiro)
```

Fonte: Elaborado pelo autor.

Esse Código 2 foi incrementado do Código 1, com adição da função "imprimirTabuleiro" nas linhas de 6 a 18 e a chamada para essa mesma função na linha 22. Na definição dessa função, pode-se ver que um **for** é utilizado na linha 7, usando a função **enumerate** (que retorna o índice e valor de cada elemento de uma **list**), para receber os índices e os valores (que estão separados por vírgulas). Dessa forma pode-se utilizar a variável **índice** para saber a posição e a variável **valor** para saber o valor. Caso o valor seja 0, imprime-se um espaço e o índice, modificando o separador e o final, para não ter separação nem final de linha, de forma a facilitar a formatação da apresentação dos dados ao jogador. Caso o valor seja 1, imprime-se um espaço e o caractere "X" para indicar que a posição está ocupada pelo símbolo do jogador que tem "X", como mostrado nas linhas 10 e 11. E nas linhas 12 e 13 imprime-se um espaço e "O" para indicar que essa posição está ocupada pelo símbolo do jogador que tem "O". O resultado desse código pode ser visto na Figura 4.

Código 3 – Resultado do Código 2.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6"/Users/
BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap8/projeto/velha.py"
2. 0 1 2
3. 3 4 5
4. 6 7 8
5. Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

Como a lista está vazia, imprimiu-se somente as posições que o jogador pode colocar seu símbolo. Um detalhe, que pode ser considerado ruim, é que o "0" usado para a posição pode confundido com o "O" que é utilizado para o símbolo. Dessa forma, será modificada a função para ter as posições de 1 até 9, e trabalha-se internamente com esse número decrescido de 1 para indicar o índice da **list**. Além disso, apesar de o Código 3 mostrar as posições, ela não tem o tabuleiro como o jogador está acostumado a ver (Figura 1). Dessa forma, utiliza-se os caracteres "-+|" para desenhar na tela um tabuleiro. Essas modificações podem ser vistas no Código 4.

Código 4 – Imprimindo o tabuleiro com ajustes.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.             print(" ",indice+1,sep="",end='')
10.        elif valor == 1:
11.            print(" X", end='')
12.        else:
13.            print(" O", end='')
14.
15.        if indice == 2 or indice == 5:
16.            print("\n---+---+---\n", end='')
17.        elif indice < 8:
18.            print(" |",end='')
19.    print("\n")
20.
```

```
21. tabuleiro = novoTabuleiro()
22.
23. imprimirTabuleiro(tabuleiro)
```

Fonte: Elaborado pelo autor.

As modificações do Código 3 no Código 4 podem ser vistas nas linhas 9, em que se imprime o índice acrescido de 1, para termos as posições mostradas de 1 até 9, e da linha 15 até 18, em que se imprime o tabuleiro com os caracteres. No Código 5 pode-se visualizar o resultado desse código.

Código 5 – Resultado do Código 4.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6"/Users/
BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap8/projeto/velha.py"
2.  1 | 2 | 3
    _ _|_ _|_ _
3.  4 | 5 | 6
    _ _|_ _|_ _
4.  7 | 8 | 9
5. Process finished with exit code 0
```

Fonte: Elaborada pelo autor.

8.2.2 Pegando a entrada do jogador

Antes de implementar o fluxo do jogo, é necessário desenvolver a função que irá pegar a entrada do usuário. Essa função deverá perguntar ao jogador que posição ele deseja colocar seu símbolo, e retornar, como mostra o Código 6.

Código 6 – Implementando a entrada do jogador.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.            print(" ",indice+1,sep="",end='')
10.    elif valor == 1:
11.           print(" X", end='')
12.    else:
13.           print(" O", end='')
14.
15.        if indice == 2 or indice == 5:
```

```
16.            print("\n---+---+---\n", end='')
17.         elif indice < 8:
18.            print(" |",end='')
19.     print("\n")
20.
21. def recebeJogada(jogador):
22.     jogada = int(input("Digite a posição a jogar 1-9 (jogador %s):" % jo-
gador))
23.     return jogada
24.
25. tabuleiro = novoTabuleiro()
26.
27. jogador = "X"
28.
29. imprimirTabuleiro(tabuleiro)
30. jogada = jogadaJogador(jogador)
```

Fonte: Elaborado pelo autor.

Nesse código temos adicionado a definição da função "recebeJogada", das linhas 21 a 23. Essa função recebe como parâmetro o argumento "jogador" que define o símbolo que está jogando no momento. Na linha 22 é requerida a entrada do jogador, informando a ele quais as posições aceitas, e qual o jogador que está na vez. Essa função retorna o número digitado pelo jogador na linha 23. Além disso, na linha 27 é criado uma variável **string** para guardar o jogador que está jogando no momento. Dessa forma o programa fica com a saída de acordo com o Código 6.

Código 7 – Resultado do Código 6.

```
1. /Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6"/Users/
BEPID/Dropbox (BEPiD)/iesde/algoritmos 1/cap8/projeto/velha.py"
2.  1 ¦ 2 ¦ 3
    _ _¦_ _¦_ _
3.  4 ¦ 5 ¦ 6
    _ _¦_ _¦_ _
4.  7 ¦ 8 ¦ 9
5.  Digite a posição a jogar 1-9 (jogador X):
```

Fonte: Elaborado pelo autor.

8.2.3 Melhorando a função de entrada do jogador com try/exception

Um dos problemas da função definida no Código 4 é que caso o jogador digite algo diferente de um número, ocorre um erro de exceção. Para evitar isso, deve-se utilizar um **try/**

exception, que consegue contornar erros no tempo de execução sem parar o programa. Esse comando tem a estrutura conforme o Código 8.

Código 8 – Comando try/exception.

```
1. try:
2.     Instruções_1
3. except TIPO:
4.     Instruções_2
```

Fonte: Elaborado pelo autor.

No Código 8, pode-se verificar o comando **try**, que funciona em conjunto com o comando **exception**. O comando **try** tem como tradução *tentar*, ou seja, ele tenta realizar as "instruções_1" e caso tenha uma exceção do tipo definido em **TIPO** (feita no comando **exception TIPO**:), ele irá rodar as "instruções_2". Esse comando poderia ser também colocado sem ter um **TIPO** definido, sendo assim qualquer tipo de exceção entraria no bloco definido pelo comando **exception**. Um exemplo desse comando verificando a conversão de uma **string** em **int** e capturando o erro pode ser visto no Código 9.

Código 9 – Exemplo de comando try/exception para converter uma string em int.

```
1. try:
2.     numero = int("aaa")
3. except ValueError:
4.     print("Erro ao converter")
```

Fonte: Elaborado pelo autor.

No Código 9, na linha 2 é tentado converter a **string** "aaa" em um inteiro e coloca essa conversão na variável de nome "**numero**", como "aaa" não pode ser convertida em um **int**, ele levantará uma exceção rodando o código definido na linha 4, imprimindo "Erro ao converter".

Sendo assim, fica melhor no jogo da velha evitar possíveis erros que o jogador possa fazer, caso a entrada dele seja diferente de um **int**. Dessa forma, colocará a entrada do usuário dentro de um bloco **try/exception**, como mostra o Código 10.

Código 10 – Entrada do usuário com try/exception.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.             print(" ",indice+1,sep="",end='')
10.        elif valor == 1:
```

```
11.             print(“ X”, end=’’)
12.         else:
13.             print(“ O”, end=’’)
14.
15.         if indice == 2 or indice == 5:
16.             print(“\n---+---+---\n”, end=’’)
17.         elif indice < 8:
18.             print(“ |”,end=’’)
19.     print(“\n”)
20.
21. def recebeJogada(jogador):
22.     try:
23.     jogada = int(input(“Digite a posição a jogar 1-9 (jogador %s):” % jo-
gador))
24.     return jogada
25.   except ValueError:
26.     print(“Entrada invalida”)
27.     return -1
28.
29. tabuleiro = novoTabuleiro()
30.
31. jogador = “X”
32.
33.
34. imprimirTabuleiro(tabuleiro)
35. jogada = recebeJogada(jogador)
```

Fonte: Elaborado pelo autor.

8.2.4 Verificação da entrada do usuário

Apesar de o Código 10 já estar prevendo eventuais erros que o usuário possa cometer na sua entrada, ainda é necessário evitar outros erros, como tentar colocar o símbolo em uma posição que já existe um símbolo, ou caso o usuário digite um número diferente das posições válidas, isso é, um número menor que 1 ou maior que 9. Dessa forma será criado uma função “posicaoValida”, como mostra o Código 11.

Código 11 – Código do jogo da velha com a verificação de jogadas.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.            print(“ “,indice+1,sep=””,end=’’)
10.        elif valor == 1:
11.           print(“ X”, end=’’)
12.        else:
13.           print(“ O”, end=’’)
14.
15.        if indice == 2 or indice == 5:
16.           print(“\n---+---+---\n”, end=’’)
17.        elif indice < 8:
18.           print(“ |”,end=’’)
19.    print(“\n”)
20.
21. def recebeJogada(jogador):
22.    try:
23.        jogada = int(input(“Digite a posição a jogar 1-9 (joga-
dor %s):” % jogador))
24.        return jogada
25.    except ValueError:
26.        print(“Entrada invalida”)
27.        return -1
28.
29.
30. def posicaoValida(jogada,tabuleiro):
31.    if jogada < 1 or jogada > 9:
```

```
32.         print("Posição invalida")
33.         return False
34.     if tabuleiro[jogada-1] != 0:
35.         print("Posição ocupada")
36.         return False
37.     return True
38.
39. tabuleiro = novoTabuleiro()
40.
41. jogador = "X"
42.
43.
44. imprimirTabuleiro(tabuleiro)
45. jogada = recebeJogada(jogador)
46. verificaJogada(jogada,tabuleiro)
```

Fonte: Elaborado pelo autor.

No Código 11, nas linhas 30 a 37 a função "posicaoValida" é definida, e recebe dois argumentos: a posição da jogada e o tabuleiro. Nessa função, primeiramente na linha 31, verifica se a posição da jogada é menor que 1 ou maior que 9, caso seja, imprime uma mensagem que essa é uma posição inválida e retorna **False**. A seguir, na linha 33, verifica se a posição que está tentando colocar o símbolo está ocupada, caso esteja, imprime uma mensagem de erro falando que a posição está ocupada e retorna **False**. Caso alguma dessas opções seja verdadeira, retorna **True**.

8.2.5 Mudando de jogador

Após verificar se a entrada do jogador é verdadeira, é necessário colocar a nova posição com o símbolo do jogador, e mudar o seu turno, como mostra o Código 12.

Código 12 – Jogo da velha com implementação da mudança de jogador.

```
1.  def novoTabuleiro():
2.          return [0,0,0,
3.                  0,0,0,
4.                  0,0,0]
5.
```

```
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.            print(“ “,indice+1,sep=””,end=’’)
10.    elif valor == 1:
11.           print(“ X”, end=’’)
12.    else:
13.           print(“ O”, end=’’)
14.
15.    if indice == 2 or indice == 5:
16.           print(“\n---+---+---\n”, end=’’)
17.    elif indice < 8:
18.           print(“ |”,end=’’)
19.    print(“\n”)
20.
21.def recebeJogada(jogador):
22.    try:
23.        jogada = int(input(“Digite a posição a jogar 1-9 (jogador %s):” % jogador))
24.        return jogada
25.    except ValueError:
26.        print(“Entrada invalida”)
27.        return -1
28.
29.
30.def posicaoValida(jogada,tabuleiro):
31.    if jogada < 1 or jogada > 9:
32.       print(“Posição invalida”)
33.       return False
34.    if tabuleiro[jogada-1] != 0:
35.       print(“Posição ocupada”)
36.       return False
```

```
37.     return True
38.
39. def mudaJogador(jogador,jogada,tabuleiro):
40.     if jogador == "X":
41.         tabuleiro[jogada-1] = 1
42.         return "O"
43.     else:
44.         tabuleiro[jogada - 1] = 2
45.         return "X"
46.
47. tabuleiro = novoTabuleiro()
48.
49. jogador = "X"
50.
51.
52. imprimirTabuleiro(tabuleiro)
53. jogada = recebeJogada(jogador)
54. posicaoValida(jogada,tabuleiro)
55. jogador = mudaJogador(jogador,jogada,tabuleiro)
```

Fonte: Elaborado pelo autor.

No Código 12, nas linhas 39 até 45, define-se a função "mudaJogador", que recebe como argumentos o jogador atual, a jogada com a posição escolhida e a **list** com o tabuleiro. Essa função tem um **if** para verificar o jogador que está jogando e coloca no tabuleiro o símbolo de acordo com a posição (linha 41 ou 44) e retorna o próximo jogador a realizar a jogada (linha 42 ou 45).

8.2.6 Verificando se o tabuleiro tem algum ganhador

Agora, falta implementar as regras para determinar o fim de jogo, seja pelo número de jogadas, seja pelo fato de algum jogador ter conseguido colocar três símbolos em sequência. Dessa forma, será necessário criar uma função que verifica se o número de jogadas é maior que 9, ou se algum símbolo está colocado em uma sequência de 3. Para isso deve-se verificar se existe algum símbolo com 3 em linhas, que pode ser superior, do meio ou inferior, ou 3 em colunas, esquerda, do meio ou direita, ou 3 em diagonais, diagonal da esquerda para direita, diagonal da direita para esquerda.

Código 13 – Jogo da velha com verificação do final.

```
1.  def novoTabuleiro():
2.  return [0,0,0,
```

```
3.          0,0,0,
4.          0,0,0]
5.
6.  def imprimirTabuleiro(tabuleiro):
7.      for indice,valor in enumerate(tabuleiro):
8.          if valor == 0:
9.             print(“ “,indice+1,sep=””,end=’’)
10.     elif valor == 1:
11.            print(“ X”, end=’’)
12.     else:
13.            print(“ O”, end=’’)
14.
15.     if indice == 2 or indice == 5:
16.            print(“\n---+---+---\n”, end=’’)
17.     elif indice < 8:
18.           print(“ |”,end=’’)
19.     print(“\n”)
20.
21. def recebeJogada(jogador):
22.     try:
23.            jogada  =  int(input(“Digite  a  posição  a  jogar  1-9  (joga-
dor %s):” % jogador))
24.            return jogada
25.     except ValueError:
26.            print(“Entrada invalida”)
27.            return -1
28.
29.
30. def posicaoValida(jogada,tabuleiro):
31.     if jogada < 1 or jogada > 9:
32.        print(“Posição invalida”)
33.        return False
34.     if tabuleiro[jogada-1] != 0:
35.        print(“Posição ocupada”)
36.        return False
37.     return True
```

```
38.
39. def mudaJogador(jogador,jogada,tabuleiro):
40.     if jogador == "X":
41.        tabuleiro[jogada-1] = 1
42.        return "O"
43.
44.     else:
45.          tabuleiro[jogada - 1] = 2
46.          return "X"
47.
48. def verificaFimDeJogo(numJogadas,tabuleiro):
49.     #verifica linhas
50.     if tabuleiro[0] == tabuleiro[1] == tabuleiro[2]:#verifica primeira li-
nha
51.        if tabuleiro[0] == 1:
52.           print("Jogador X ganhou")
53.           return 1
54.        elif tabuleiro[0] == 2:
55.           print("Jogador O ganhou")
56.           return 2
57.     if tabuleiro[3] == tabuleiro[4] == tabuleiro[5]:#verifica segunda linha
58.        if tabuleiro[3] == 1:
59.           print("Jogador X ganhou")
60.           return 1
61.        elif tabuleiro[3] == 2:
62.           print("Jogador O ganhou")
63.           return 2
64.     if tabuleiro[6] == tabuleiro[7] == tabuleiro[8]:#verifica terceira linha
65.        if tabuleiro[6] == 1:
66.           print("Jogador X ganhou")
67.           return 1
68.     elif tabuleiro[6] == 2:
69.          print("Jogador O ganhou")
70.          return 2
71. #verifica colunas
72.     if tabuleiro[0] == tabuleiro[3] == tabuleiro[6]:#verifica primeira coluna
```

```
73.       if tabuleiro[0] == 1:
74.          print("Jogador X ganhou")
75.          return 1
76.       elif tabuleiro[0] == 2:
77.          print("Jogador O ganhou")
78.          return 2
79.    if tabuleiro[1] == tabuleiro[4] == tabuleiro[7]:#verifica segunda coluna
80.       if tabuleiro[1] == 1:
81.          print("Jogador X ganhou")
82.          return 1
83.       elif tabuleiro[1] == 2:
84.          print("Jogador O ganhou")
85.          return 2
86.    if tabuleiro[2] == tabuleiro[5] == tabuleiro[8]:#verifica terceira co-
luna
87.       if tabuleiro[2] == 1:
88.          print("Jogador X ganhou")
89.          return 1
90.       elif tabuleiro[2] == 2:
91.          print("Jogador O ganhou")
92.              return 2
93.#verifica diagonais
94.    if tabuleiro[0] == tabuleiro[4] == tabuleiro[8]:#verifica diago-
nal da esquerda para direita
95.       if tabuleiro[0] == 1:
96.print("Jogador X ganhou")
97.return 1
98.elif tabuleiro[0] == 2:
99.print("Jogador O ganhou")
100.       return 2
101.       if tabuleiro[2] == tabuleiro[4] == tabuleiro[6]:#verifica diago-
nal da direita para esquerda
102.          if tabuleiro[2] == 1:
103.              print("Jogador X ganhou")
104.              return 1
105.          elif tabuleiro[2] == 2:
```

```
106.                print("Jogador O ganhou")
107.                return 2
108.        if numJogadas >= 9:
109.                print("Deu Velha")
110.                return -1
111.           return 0
112.
113.        tabuleiro = novoTabuleiro()
114.
115.        jogador = "X"
116.        jogadas = 0
117.
118.
119.        imprimirTabuleiro(tabuleiro)
120.        jogada = recebeJogada(jogador)
121.        posicaoValida(jogada,tabuleiro)
122.        jogador = mudaJogador(jogador,jogada,tabuleiro)
123.        jogadas += 1
124.        verificaFimDeJogo(jogadas,tabuleiro)
```

Fonte: Elaborado pelo autor.

No Código 13, das linhas 48 até a linha 111 é definida a função que verifica o fim do jogo, se o jogador X conseguir fazer 3 em sequência, a função retorna 1, caso o jogador O conseguir fazer 3 em sequência retorna 2, caso tenha dado velha retorna -1 e caso não seja o fim do jogo se retorna 0. Primeiramente a função verifica se algum dos jogadores conseguiu fazer 3 símbolos em sequência de linha (linhas 50 a 70), se algum jogador conseguiu fazer 3 símbolos em sequência de coluna (linhas 72 até 92), e se algum jogador conseguiu fazer 3 símbolos em diagonal (linhas 94 até 107). Depois verifica-se se o jogo terminou, se estourou o número de jogadas (linha 108, 109 e 110), dando velha. E caso nenhuma das alternativas anteriores tenha dado **True**, retorna 0 para continuar o jogo (linha 111).

Vídeo

8.3 Colocando o fluxo do jogo

Tendo já as funcionalidades básicas para começo do jogo e para imprimir o tabuleiro, é necessário implementar o **loop** principal do jogo, de acordo com a Figura 2. Esse **loop** principal irá colocar toda a sequência das funções implementadas na seção anterior e

implementá-la em um **loop**, verificando se a entrada é válida, caso não seja ela faz um **continue**, ou se tem o fim do jogo, que realiza o **break** do **loop**. O código do programa completo com o **loop** pode ser visto no Código 14.

Código 14 – Jogo da velha completo.

```
1. def novoTabuleiro():
2.     return [0,0,0,
3.             0,0,0,
4.             0,0,0]
5.
6. def imprimirTabuleiro(tabuleiro):
7.     for indice,valor in enumerate(tabuleiro):
8.         if valor == 0:
9.             print(“ “,indice+1,sep=””,end=’’)
10.        elif valor == 1:
11.            print(“ X”, end=’’)
12.        else:
13.            print(“ O”, end=’’)
14.
15.        if indice == 2 or indice == 5:
16.            print(“\n---+---+---\n”, end=’’)
17.        elif indice < 8:
18.            print(“ |”,end=’’)
19.    print(“\n”)
20.
21.def recebeJogada(jogador):
22.    try:
23.        jogada = int(input(“Digite a posição a jogar 1-9 (jogador %s):” % jogador))
24.        return jogada
25.    except ValueError:
26.        print(“Entrada invalida”)
27.        return -1
28.
```

```
29.
30. def posicaoValida(jogada,tabuleiro):
31.     if jogada < 1 or jogada > 9:
32.         print("Posição invalida")
33.         return False
34.     if tabuleiro[jogada-1] != 0:
35.         print("Posição ocupada")
36.         return False
37.     return True
38.
39. def mudaJogador(jogador,jogada,tabuleiro):
40.     if jogador == "X":
41.         tabuleiro[jogada-1] = 1
42.         return "O"
43.
44. else:
45.         tabuleiro[jogada - 1] = 2
46.         return "X"
47.
48. def verificaFimDeJogo(numJogadas,tabuleiro):
49.     #verifica linhas
50.     if tabuleiro[0] == tabuleiro[1] == tabuleiro[2]:#verifica primeira linha
51.         if tabuleiro[0] == 1:
52.             print("Jogador X ganhou")
53.             return 1
54.         elif tabuleiro[0] == 2:
55.             print("Jogador O ganhou")
56.             return 2
57.     if tabuleiro[3] == tabuleiro[4] == tabuleiro[5]:#verifica segunda linha
58.         if tabuleiro[3] == 1:
59.             print("Jogador X ganhou")
60.             return 1
61.         elif tabuleiro[3] == 2:
62.             print("Jogador O ganhou")
63.             return 2
```

```
64.     if tabuleiro[6] == tabuleiro[7] == tabuleiro[8]:#verifica tercei-
ra linha
65.        if tabuleiro[6] == 1:
66.           print("Jogador X ganhou")
67.           return 1
68.        elif tabuleiro[6] == 2:
69.           print("Jogador O ganhou")
70.           return 2
71.     #verifica colunas
72.     if tabuleiro[0] == tabuleiro[3] == tabuleiro[6]:#verifica primei-
ra coluna
73.        if tabuleiro[0] == 1:
74.           print("Jogador X ganhou")
75.           return 1
76.        elif tabuleiro[0] == 2:
77.           print("Jogador O ganhou")
78.           return 2
79.     if tabuleiro[1] == tabuleiro[4] == tabuleiro[7]:#verifica segunda coluna
80.        if tabuleiro[1] == 1:
81.           print("Jogador X ganhou")
82.           return 1
83.        elif tabuleiro[1] == 2:
84.           print("Jogador O ganhou")
85.           return 2
86.     if tabuleiro[2] == tabuleiro[5] == tabuleiro[8]:#verifica tercei-
ra coluna
87.        if tabuleiro[2] == 1:
88.           print("Jogador X ganhou")
89.           return 1
90.        elif tabuleiro[2] == 2:
91.           print("Jogador O ganhou")
92.           return 2
93.        #verifica diagonais
94.     if tabuleiro[0] == tabuleiro[4] == tabuleiro[8]:#verifica diago-
nal da esquerda para direita
```

```
95.         if tabuleiro[0] == 1:
96.            print("Jogador X ganhou")
97.            return 1
98.         elif tabuleiro[0] == 2:
99.            print("Jogador O ganhou")
100.                return 2
101.        if tabuleiro[2] == tabuleiro[4] == tabuleiro[6]:#verifica diago-
nal da direita para esquerda
102.             if tabuleiro[2] == 1:
103.                print("Jogador X ganhou")
104.                return 1
105.             elif tabuleiro[2] == 2:
106.                print("Jogador O ganhou")
107.                return 2
108.         if numJogadas >= 9:
109.            print("Deu Velha")
110.            return -1
111.         return 0
112.
113. tabuleiro = novoTabuleiro()
114.
115. jogador = "X"
116. jogadas = 0
117.
118. while True:
119.        imprimirTabuleiro(tabuleiro)
120.        jogada = recebeJogada(jogador)
121.        if not posicaoValida(jogada,tabuleiro):
122.              continue
123.        jogador = mudaJogador(jogador,jogada,tabuleiro)
124.        jogadas += 1
125.        if verificaFimDeJogo(jogadas,tabuleiro) != 0:
126.                break
```

Fonte: Elaborado pelo autor.

8.3.1 Números randômicos

Até agora foram implementadas todas as regras para o jogo da velha, mas são necessários dois jogadores para jogar. Para desenvolver um jogador simulado pelo computador, é necessário usar números randômicos, que podem gerar as posições de um jogador virtual de forma aleatória. Um exemplo de uso de número randômico pode ser visto no Código 15.

Código 15 – Exemplo de número randômico.

```
1. from random import randint
2. randomico = randint(1,9)
```

Fonte: Elaborado pelo autor.

O Código 15 primeiramente faz o importe do módulo necessário para o número randômico, no caso importe a função **randint** do módulo **random**. Após isso, chama a função **randint**, que recebe dois argumentos, o primeiro o número mínimo que pode ser gerado pelo randômico e o segundo argumento com o número máximo que pode ser gerado pelo randômico. No caso da linha 2, a variável "randômico" será um número entre 1 e 9. A implementação desse jogo da velha de um jogador será realizada como exercício.

Ampliando seus conhecimentos

Engine de Inteligência Artificial

(CLUA; BITTENCOURT, 2005)

Entende-se por Inteligência Artificial (IA) para jogos, os programas que descreverão o comportamento de entidades não controladas pelo jogador, tipicamente os NPCs (*Non-Player Characters*).

Na maioria dos casos, o comportamento inteligente desses agentes computacionais é implementado através de máquinas de estados. Os algoritmos de máquinas de estados procuram resolver problemas formalizando diversos possíveis estados em que um elemento pode se encontrar (no caso de uma televisão, por exemplo, poderia-se ter estes estados: Desligada, Acesa e Stand-by). A transição de um estado para outro estará atrelado a algum evento que dispare esta mudança (no exemplo anterior, ao apertar a tecla <on> a TV muda do estado de desligada para Stand-by).

Desta maneira, a inteligência de um personagem de um jogo pode ser descrita por diversos estados em que o mesmo pode se encontrar (no caso de um jogo de ação, poderíamos descrever estes estados como espera, perseguição, ataque, fuga, morte, por exemplo). Este personagem deverá fazer

um monitoramento constante sobre acontecimentos que possam disparar a mudança de um estado para outro. Usando o exemplo de um jogo de ação, suponha- se que o fato do jogador aproximar-se mais do que 30m do NPC faça com que o mesmo passe do estado de espera para o estado de perseguição.Entretanto é perceptível que ocorreram grandes inovações do ponto de vista gráfico, mas aspectos inteligentes das entidades computacionais não foram sofisticados com a mesma velocidade. Muitas vezes um jogo sofisticado graficamente apresenta uma IA mediana. Isto ocorre principalmente pelo fato do custo computacional que os algoritmos inteligentes e de processamento gráfico possuem. Conforme Sewald [SEW 02], técnicas de IA tais como Redes Neuronais Artificiais e Algoritmos Genéticos são pouco utilizadas em games. [...]

Um exemplo de toolkit para o desenvolvimento de agentes inteligentes é o DirectIA [DIR 05]. Trata-se de uma solução comercial genérica que permite desenvolver agentes adaptativos que aprendem através do contato com o ambiente e com usuário. Oferece uma série de módulos de Emoção Artificial, planejamento, comunicação e Aprendizado de Máquinas.

É importante destacar que ferramentas de suporte para criação da IA de games ainda é uma área deficitária, logo aberta para investigações técnico-científicas.

Atividades

1. Com base no Código 13, implemente o jogo da velha de um jogador.

Referências

CLUA, Esteban Walter Gonzalez, BITTENCOURT, João Ricardo. **Desenvolvimento de jogos 3D: concepção, design e programação.** XXIV Jornadas de Atualização em Informática (JAI) Part of XXIV Congresso da Sociedade Brasileira de Computação. 2005.

Resolução

1.

```
1. from random import randint
2.
3.
4. def novoTabuleiro():
5.     return [0,0,0,
```

```
6.              0,0,0,
7.              0,0,0]
8.
9. def imprimirTabuleiro(tabuleiro):
10.    for indice,valor in enumerate(tabuleiro):
11.        if valor == 0:
12.          print(“ “,indice+1,sep=””,end=’’)
13.        elif valor == 1:
14.          print(“ X”, end=’’)
15.        else:
16.    print(“ 0”, end=’’)
17.
18.        if indice == 2 or indice == 5:
19.          print(“\n---+---+---\n”, end=’’)
20.        elif indice < 8:
21.            print(“ |”,end=’’)
22.    print(“\n”)
23.
24.def recebeJogada(jogador):
25.    try:
26.          jogada  =  int(input(“Digite  a  posição  a  jogar  1-9  (joga-
dor %s):” % jogador))
27.          return jogada
28.except ValueError:
29.          print(“Entrada invalida”)
30.          return -1
31.
32.
33.def posicaoValida(jogada,tabuleiro):
34.    if jogada < 1 or jogada > 9:
35.         print(“Posição invalida”)
36.         return False
37.    if tabuleiro[jogada-1] != 0:
38.         print(“Posição ocupada”)
39.         return False
40.    return True
```

```
41.
42. def mudaJogador(jogador,jogada,tabuleiro):
43.     if jogador == "X":
44.         tabuleiro[jogada-1] = 1
45. return "O"
46.
47.     else:
48.           tabuleiro[jogada - 1] = 2
49.           return "X"
50.
51. def verificaFimDeJogo(numJogadas,tabuleiro):
52.     #verifica linhas
53.     if tabuleiro[0] == tabuleiro[1] == tabuleiro[2]:#verifica primeira linha
54.         if tabuleiro[0] == 1:
55.            print("Jogador X ganhou")
56.            return 1
57.         elif tabuleiro[0] == 2:
58.            print("Jogador O ganhou")
59.            return 2
60.     if tabuleiro[3] == tabuleiro[4] == tabuleiro[5]:#verifica segunda linha
61.         if tabuleiro[3] == 1:
62.            print("Jogador X ganhou")
63.            return 1
64.         elif tabuleiro[3] == 2:
65.            print("Jogador O ganhou")
66.            return 2
67.     if tabuleiro[6] == tabuleiro[7] == tabuleiro[8]:#verifica terceira linha
68.         if tabuleiro[6] == 1:
69.            print("Jogador X ganhou")
70.            return 1
71.         elif tabuleiro[6] == 2:
72.            print("Jogador O ganhou")
73.            return 2
74. #verifica colunas
75.     if tabuleiro[0] == tabuleiro[3] == tabuleiro[6]:#verifica primeira coluna
76.         if tabuleiro[0] == 1:
```

```
77.           print("Jogador X ganhou")
78.           return 1
79.        elif tabuleiro[0] == 2:
80.           print("Jogador O ganhou")
81.           return 2
82.     if tabuleiro[1] == tabuleiro[4] == tabuleiro[7]:#verifica segunda coluna
83.        if tabuleiro[1] == 1:
84.           print("Jogador X ganhou")
85.           return 1
86.        elif tabuleiro[1] == 2:
87.           print("Jogador O ganhou")
88.           return 2
89.     if tabuleiro[2] == tabuleiro[5] == tabuleiro[8]:#verifica terceira coluna
90.        if tabuleiro[2] == 1:
91.           print("Jogador X ganhou")
92.           return 1
93.        elif tabuleiro[2] == 2:
94.           print("Jogador O ganhou")
95.           return 2
96.#verifica diagonais
97.     if tabuleiro[0] == tabuleiro[4] == tabuleiro[8]:#verifica diago-
nal da esquerda para direita
98.        if tabuleiro[0] == 1:
99.           print("Jogador X ganhou")
100.               return 1
101.           elif tabuleiro[0] == 2:
102.               print("Jogador O ganhou")
103.               return 2
104.     if tabuleiro[2] == tabuleiro[4] == tabuleiro[6]:#verifica diago-
nal da direita para esquerda
105.     if tabuleiro[2] == 1:
106.               print("Jogador X ganhou")
107.               return 1
108.     elif tabuleiro[2] == 2:
109.               print("Jogador O ganhou")
110.               return 2
```

```
111.        if numJogadas >= 9:
112.            print("Deu Velha")
113.            return -1
114.        return 0
115.
116.    def jogadaCPU(jogador):
117.        jogada = randint(1,9)
118.        while tabuleiro[jogada-1] != 0:
119.              jogada = randint(1, 9)
120.
121.        print("Digite a posi��o a jogar 1-9 (jogador %s):" % jogador,jogada)
122.
123.        return jogada
124.
125.    tabuleiro = novoTabuleiro()
126.
127.    jogador = "X"
128.    jogadas = 0
129.
130.    while True:
131.         imprimirTabuleiro(tabuleiro)
132.         if jogador == "X":
133.             jogada = recebeJogada(jogador)
134.         else:
135.             jogada = jogadaCPU(jogador)
136.
137.         if not posicaoValida(jogada,tabuleiro):
138.             continue
139.         jogador = mudaJogador(jogador,jogada,tabuleiro)
140.         jogadas += 1
141.         if verificaFimDeJogo(jogadas,tabuleiro) != 0:
142.             break
```

Código Logístico
57108

IESDE
Ad Maiora Semper!

Fundação Biblioteca Nacional
ISBN 978-85-387-6349-9
9 788538 763499