**Introdução - Linguagem de Programação**

Esta disciplina se propõe ao estudo do desenvolvimento de algoritmos e suas respectivas implementações na linguagem de programação Python, gerando programas executáveis em computadores. É uma das disciplinas fundamentais do curso e os conceitos estudados aqui servirão de base para todas as disciplinas relacionadas a programação. A disciplina está organizada em 4 unidades, nas quais serão abordados tópicos como: variáveis, tipos de dados, entrada e saída de dados, operadores aritméticos, relacionais e lógicos, estruturas de decisão, estruturas de repetição, funções e sequências.

As unidades são:

* Introdução, ambiente de desenvolvimento e estrutura básica de Python;
* Expressões relacionais e lógicas, estruturas de seleção e funções;
* Estruturas de repetição;
* Sequências (listas, tuplas e strings).

**Livros**

**Livro** - [Aprendendo com Python: Edição interativa](https://panda.ime.usp.br/pensepy/static/pensepy/#)

**Livro** - [Pense em Python (2ª ed.), de Allen B. Downey](https://penseallen.github.io/PensePython2e/)

**Parte 1 -** [**Introdução à disciplina e conceitos iniciais**](https://sites.google.com/faculdadeimpacta.com.br/lp-p1?authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=BTx-GIDSdvs)

**Resumo**

Nesta aula os objetivos são: (I) introduzir as diretrizes da disciplina, como conteúdo programático, metodologia de ensino, critérios de avaliação e bibliografia; (II) conceituar os termos “algoritmo”, “lógica de programação”, “linguagem de programação”, “código-fonte” e “programa”; (III) simular o processo para resolução de problemas computacionais sem o uso de computadores (computação desplugada).

**Motivação**

A vida das pessoas é constantemente alterada pela evolução das tecnologias, inclusive pela tecnologia computacional. Diversos aspectos da vida são impactados pelo progresso das descobertas relacionadas à área da Ciência da Computação. Você pode imaginar o quão diferente seria sua rotina sem a existência de notebooks e smartphones?

Brookshear (2008, p. 17) menciona que, como uma disciplina, “[a Ciência da Computação] busca construir uma base científica para diversos tópicos, tais como a construção e a programação de computadores, o processamento de informações, as soluções algorítmicas de problemas e o processo algorítmico propriamente dito.”. Podemos resolver problemas com a construção de algoritmos e, para automatizá-los, aplicaremos lógica de programação com instruções em linguagens de programação, criando códigos-fonte que se tornarão programas executados por computadores.

**Algoritmos**

Conforme descrito por Ziviani (1999, p. 1) “[algoritmo é] uma sequência de passos de ações executáveis para a obtenção de uma solução para um determinado tipo de problema”. Podemos imaginar um algoritmo como um procedimento que visa resolver um problema, formado por uma sequência ordenada e finita de passos passíveis de serem executados, assim como conceituado por Pereira (2010).

O termo algoritmo não está diretamente relacionado aos computadores, muito menos depende deles, pois podemos criar algoritmos que não serão, necessariamente, executados em um computador. Imagine, por exemplo, uma receita de bolo

Uma característica importantíssima sobre os algoritmos é que a construção pode variar de acordo com quem os elabora. Portanto, nem todos os algoritmos que objetivam resolver o mesmo problema terão a mesma sequência de passos, afinal dependerá, dentre outros fatores, da experiência, vocabulário, concisão e lógica do autor. Em contraste, a execução dos algoritmos, se forem bem formulados, independe de quem os executará, bastando que os passos sejam rigorosamente seguidos.

Genericamente, podemos dizer que lógica é uma parte da filosofia que trata das formas do pensamento em geral (dedução, indução, inferência, hipótese etc.) e das operações intelectuais que visam a determinação do que é verdadeiro ou falso. Faremos uso de lógica para escrever as instruções de nossos algoritmos, de modo que a execução de cada instrução faça sentido para atingir o objetivo final.

Segundo um dos mais importantes nomes da Computação, Knuth (1997), um algoritmo tem cinco características fundamentais:

**Finitude:** um algoritmo precisa terminar após um número finito de passos;

**Definição:** cada passo do algoritmo precisa ser precisamente definido, as ações devem ser especificadas clara e rigorosamente evitando ambiguidade;

**Entrada:** um algoritmo tem zero ou mais entradas, que são dados fornecidos ao algoritmo ao ser iniciado, ou dinamicamente conforme for executado;

**Saída:** um algoritmo tem zero ou mais saídas, que geralmente estão relacionadas com os dados fornecidos nas entradas;

**Eficácia:** é esperado que um algoritmo seja eficaz, no sentido que suas operações devem ser suficientemente básicas para que possam, a princípio, ser feitas precisamente e em um tempo finito por alguém com lápis e papel.

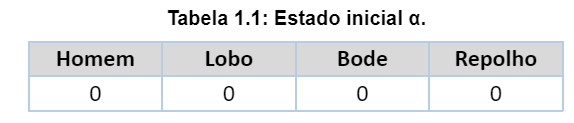
Nesta disciplina nossa atenção será direcionada para a lógica de programação, para que possamos estudar as estruturas básicas para a construção de algoritmos destinados a serem executados em computadores com uso de uma linguagem de programação.

**Abstração e representação**

Para resolver problemas computacionalmente são necessários pelo menos dois instrumentos: (I) uma representação que capte todos os aspectos relevantes do problema, descartando detalhes insignificantes para a sua resolução e; (II) um algoritmo que resolva o problema com base em ações aplicadas na representação construída.

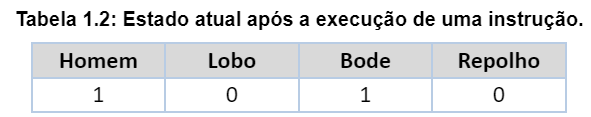
O que seria uma representação adequada para este problema? Uma vez que apenas os aspectos relevantes do problema precisam ser representados, todos os detalhes irrelevantes podem ser omitidos. Uma representação que deixa de fora detalhes do que está sendo representado é uma forma de abstração.

O uso de abstração é predominante na computação. Pense, a cor do barco é relevante? A largura do rio? O nome do homem? Não! A única informação relevante é onde está cada personagem/entidade desse contexto, a cada momento antes e após a execução de um passo/instrução da solução proposta para o problema. A localização coletiva de cada entidade, neste caso, refere-se ao estado do problema. Assim, o estado inicial do problema, que chamamos de α, pode ser representado conforme a Tabela 1.1.



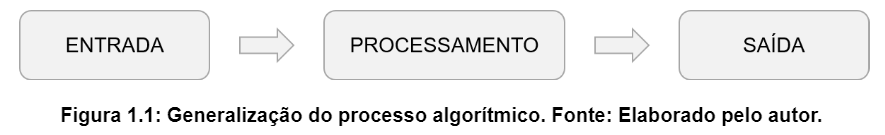
Note que nesta representação criamos uma tabela em que o nome das colunas são as entidades relevantes no contexto do problema, e os valores das colunas são indicações sobre os respectivos estados de cada entidade.

Arbitrariamente, definimos que colunas com 0 (zero) indicam que a respectiva entidade está no lado leste do rio. Já colunas com 1 (um) indicam que a entidade está no lado oeste do rio. Por exemplo, se a primeira instrução do algoritmo fosse “homem atravessa com o bode de leste para oeste”, a representação do novo estado do problema, gerado após a execução desta instrução, seria como ilustrado na Tabela 1.2. Neste novo estado apenas homem e bode estão no lado oeste e as localizações do lobo e do repolho não foram alteradas, pois a instrução executada não os impactou.



Com essa representação plenamente assimilada, podemos estabelecer que uma solução para este problema é a aplicação de um algoritmo que tenha como entrada o estado inicial α (Tabela 1.1), processe α com uma sequência de instruções logicamente ordenadas e que conduzam à saída do estado final β.

Portanto, podemos generalizar nossos algoritmos em três etapas: (I) receber entradas; (II) processá-las e; (III) gerar saídas que resolvam o problema de acordo com as entradas processadas.



É importante diferenciar a “representação” do “algoritmo” que será aplicado sobre ela. A tabela é apenas uma forma de representar o “mundo do problema” que queremos resolver, já o algoritmo é uma sequência de ações que serão aplicadas neste “mundo” de forma a deixá-lo no estado que desejamos, solucionando o problema.

**Linguagens de Programação**

Até agora discutimos sobre algoritmos de forma genérica, porém, o foco da disciplina é o estudo de algoritmos que serão executados em computadores, ou seja, serão transformados em programas. Thomas Cormen, conceitua esse tipo de algoritmo:

*Também podemos considerar um algoritmo como uma ferramenta para resolver um problema computacional bem especificado. O enunciado do problema específica em termos gerais a relação desejada entre entrada e saída. O algoritmo descreve um procedimento computacional específico para se conseguir essa relação entre entrada e saída. (CORMEN et al., 2012, p. 3).*

Para que os algoritmos possam ser executados automaticamente é necessário que sejam escritos em linguagens compreensíveis aos computadores, que são as linguagens de programação. A codificação, também chamada de implementação, é a fase em que o programador escreve seus algoritmos em uma linguagem de programação.

Como escrito por Lira (2019, p. 29), “Linguagens de programação são conjuntos de símbolos e regras de sintaxe que permitem a construção de instruções que descrevem, de forma não ambígua, ações que podem ser entendidas e executadas por meio de computadores”. Cada linguagem de programação define um padrão para a escrita de suas instruções, esses padrões têm motivações diversas, algumas linguagens favorecem a legibilidade para humanos, outras a concisão, ou o desempenho na execução etc.

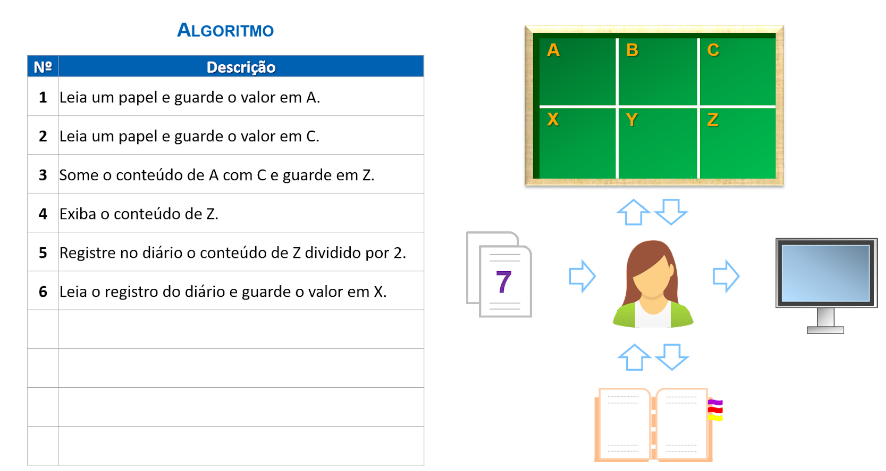
Em nossa disciplina utilizaremos a linguagem de programação Python 3.

**Computador Simplificado**

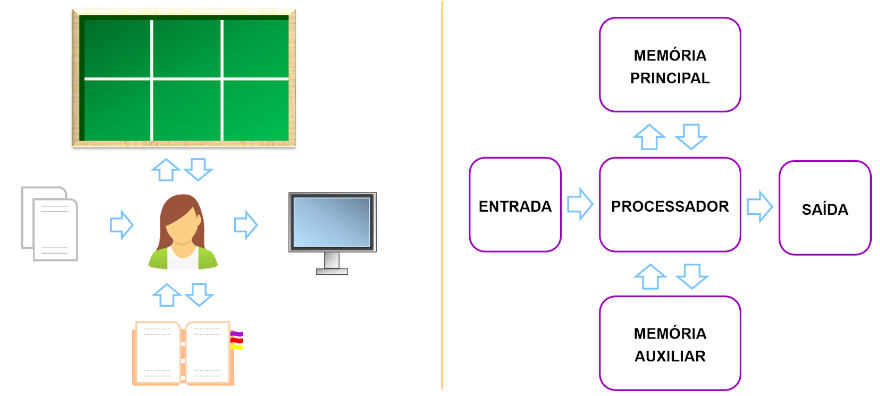
Podemos simular procedimentos computacionais sem a necessidade de um computador, aprendendo conceitos básicos e usando objetos do dia a dia que, em seguida, serão mapeados para correspondentes de um sistema computacional eletrônico. O Computador Simplificado (que abreviaremos como CS) se encaixa neste contexto!

À direita da Figura 1.2 está um algoritmo com seis instruções, à esquerda há uma representação simplificada de um sistema computacional sem referência direta aos componentes de um computador, apenas objetos do cotidiano com função equivalente.

No CS, temos: (I) ao centro, uma pessoa chamada Megan, responsável por executar as instruções do algoritmo; (II) à esquerda, papéis descartáveis, cada um representando uma entrada de dados; (III) à direita, uma tela para exibir dados para outras pessoas; (IV) acima, uma lousa em que cada parte tem um nome, em nosso caso uma única letra, e pode guardar dados temporariamente, afinal pode ser facilmente apagada; (V) abaixo, um diário para registrar dados, de modo a armazená-los por longos períodos. As setas ao redor de Megan indicam o fluxo em que os dados podem trafegar.



Com reconhecimento da importância de cada parte do CS, a Figura 1.7 ilustra a correspondência de seus itens em relação aos componentes de um computador real.



O mapeamento é dado assim: (a) Megan é o processador, pois é o componente que executa as instruções do algoritmo; (b) os papéis descartáveis são dados fornecidos como entrada, geralmente inseridos pelo usuário por meio de um teclado, mouse ou tela sensível ao toque; (c) a tela é a saída, que corresponde a um monitor ou impressora, por exemplo; (d) a lousa simboliza a memória principal, conhecida como Memória RAM, que serve para guardar dados temporariamente, pois ao cortar o fluxo de energia do computador os dados são apagados e; (e) o diário é a memória auxiliar, que serve para persistir os dados, ou seja, mantê-los registrados para que possam ser consultados futuramente, como HDDs, SSDs, cartões de memória, pendrives etc.

Você pode estar se perguntando: “qual seria a equivalência do algoritmo em um sistema computacional real?”. Resposta: ao código-fonte de um programa escrito em uma linguagem de programação. Aprender essa linguagem será um de nossos objetivos!

O que a maioria das escolas não ensinam. <https://youtu.be/nKIu9yen5nc>

**Parte 2 -** [**Ambiente de desenvolvimento, tipos de dados e variáveis**](https://sites.google.com/faculdadeimpacta.com.br/lp-p2?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=IVJ3ER-kmGg)

**Resumo**

Nesta aula os objetivos são: (I) preparar o ambiente computacional necessário para a disciplina; (II) acessar as duas ferramentas básicas utilizadas na disciplina para o desenvolvimento de programas em Python (interpretador interativo e editor de código fonte); (III) conceituar e utilizar tipos de dados primitivos, constantes e variáveis; (IV) conhecer os operadores aritméticos; (V) manipular o sistema online Python Tutor.

**Motivação**

Para criarmos nossos programas, teremos que usar um ambiente que permita escrever algoritmos em uma linguagem de programação e que possibilite a execução automática dessas instruções. Por isso, nesta aula faremos o download, instalação e execução do interpretador da linguagem Python e do IDLE, um software que possibilita o processo básico para criação e execução de programas em Python. Introduziremos conceitos básicos de Python para construir instruções simples e, por fim, manipularemos um sistema online que possibilita a visualização passo a passo do que ocorre quando nossos códigos-fonte são executados, potencialmente facilitando a aprendizagem.

**Download e instalação do Python no Windows**

Iniciaremos com o download do interpretador do Python juntamente com o IDLE, ambos são softwares gratuitos e de código aberto e serão instalados por meio do mesmo arquivo. Para isso basta acessar o site https://www.python.org, clicar na guia “Downloads” e, em seguida, no botão “Python 3.9.1” (atualmente a última versão estável disponível), como ilustrado na Figura 2.1.

**IDLE**

O IDLE (Ambiente Integrado de Desenvolvimento e Aprendizagem, tradução nossa) é o ambiente em que programaremos e foi instalado juntamente com o interpretador do Python no procedimento anterior. Este ambiente é programado na própria linguagem Python e funciona igualmente em outros sistemas operacionais. O IDLE possui dois módulos, o interpretador interativo e o editor de código-fonte, analisaremos as funcionalidades de cada um deles.

**Interpretador interativo**

Por padrão, ao clicar no atalho do IDLE no Menu Iniciar a tela exibida será do interpretador interativo, também conhecido como Shell.

Este módulo é usado principalmente para testar instruções de Python e executar programas construídos no editor. Repare na sequência de três sinais de maior (>>>) indicando que a Shell está pronta para receber novas instruções. A cada instrução, o programador deve teclar [ENTER] para executá-la e gerar o efeito correspondente.

A Shell é bastante útil e possui atalhos e recursos interessantes, podendo, inclusive, substituir uma calculadora tradicional, veremos posteriormente como fazer isso. Alguns dos atalhos que facilitam a codificação são:

* Para regressar a última instruções inserida, posicione o cursor na linha com >>> e pressione as teclas [ALT]+[P];
* Para avançar para a próxima instrução inserida, posicione o cursor na linha com >>> e pressione as teclas [ALT]+[N];
* Para interromper a execução de uma instrução ou programa pressione as teclas [CTRL]+[C].

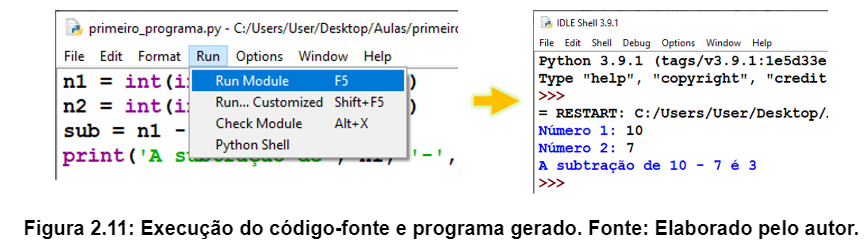
**Editor de código-fonte**

A Shell é muito útil para testar trechos de código, mas existem situações em que precisaremos construir um programa completo e guardar seu código-fonte para que possa ser editado e executado diversas vezes sem a necessidade de reescrevê-lo após encerrar o IDLE. Para isso existe o módulo editor de código-fonte. Para acessá-lo, clique no menu “File > New File” da janela da Shell.

Cada caractere é importante, portanto tome cuidado para não esquecer de nenhum ou trocá-lo por acidente. Caso tenha dúvida se em Python letras maiúsculas e minúsculas são interpretadas como caracteres diferentes, a resposta é SIM! Python é uma linguagem case-sensitive, isso quer dizer que n1 é completamente diferente de N1.

Repare que a barra de título da janela consta como “untitled”, indicando que o arquivo não foi salvo. Para executarmos o código-fonte e visualizar o programa funcionando, precisamos salvar este arquivo. Para isso, acesse o menu “File > Save”, defina o nome do arquivo e o local em que será salvo.

Após o salvamento do arquivo, a barra de título será alterada para corresponder ao nome do arquivo e local em que foi salvo. Para executar o código-fonte e visualizar o programa funcionando, vá até o menu “Run > Run Module”. O procedimento de execução e o resultado, supondo as entradas 10 e 7, estão ilustrados na Figura 2.11. Note que as saídas do programa estão em azul e as entradas do usuário em preto.



Observe que a execução do programa, mesmo que solicitada no editor, é feita na Shell. Caso queira fazer alguma alteração no código, basta selecionar novamente a janela do editor, que provavelmente foi sobreposta pela da Shell, fazer as modificações e executá-lo novamente, o código-fonte será automaticamente salvo.

**Conceitos básicos de Python**

Agora que temos conhecimento sobre o ambiente de programação que usaremos durante a disciplina, vamos aprender alguns conceitos básicos da linguagem Python.

**Constantes ou Literais**

Constantes são símbolos que representam valores e não podem ser alterados, são também chamadas de literais e geralmente são utilizadas em expressões. Exemplos de constantes em Python: 9, -541, 3.1415, 'estou aprendendo Python!', "Olá Megan", False e True.

O Python possui alguns tipos de constantes, e cada tipo deve ser usado de acordo com o contexto do problema. Os quatro tipos que usaremos de início são:

* **Números inteiros (int):** valores numéricos que não possuem ponto decimal.

Exemplos: 13, 123456789, -65, 0, 0b1011(base 2), 0o7(base 8), 0xF(base 16).

* **Números reais (float):** valores numéricos com ponto decimal ou escritos em notação científica. Note que é utilizado um ponto, não uma vírgula.

Exemplos: 2.7345, .25, -65.0, 0.0, 6.02e-23, 6.02E-23, 2e1.

* **Valores booleanos (bool):** também conhecidos como valores lógicos, existem apenas dois valores, um corresponde a falso e o outro a verdadeiro.

Exemplos: False e True. Note que somente a primeira letra é maiúscula e não há aspas como nas strings a seguir.

* **Textos (string):** são cadeias de caracteres delimitadas por apóstrofos ou aspas. São usadas para representar textos (letras, palavras ou frases).

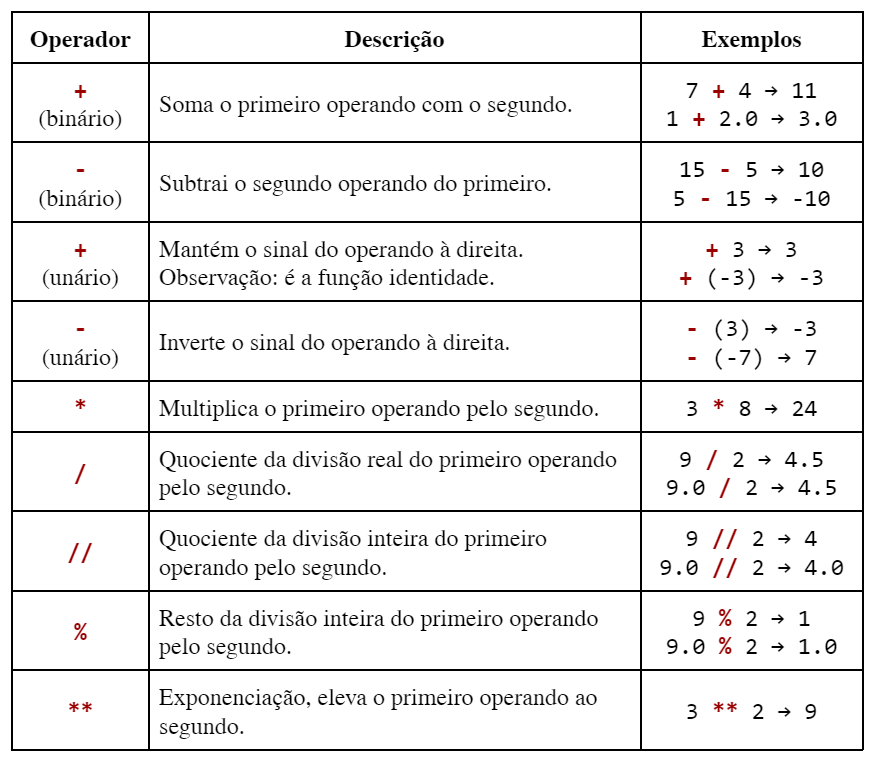
Exemplos: 'Olá!', '123', "Python é 10", "True", "3+5", 'X'.

**Operadores**

Em Python, assim como em outras linguagens de programação, operadores são símbolos pré-definidos que realizam uma operação sobre um ou mais operandos, produzindo um valor como resultado. O tipo do valor resultante dependerá do operador e dos operandos envolvidos. Quando um operador realiza uma operação entre 2 operandos, chamamos-o de binários, quando realiza uma operação com apenas 1 operando, chamamos-o de unários e, por fim, ternários quando envolvem 3 operandos.

**Operadores aritméticos**

Os operadores aritméticos, quando aplicados em operandos numéricos resultam valores numéricos, assim como na matemática.

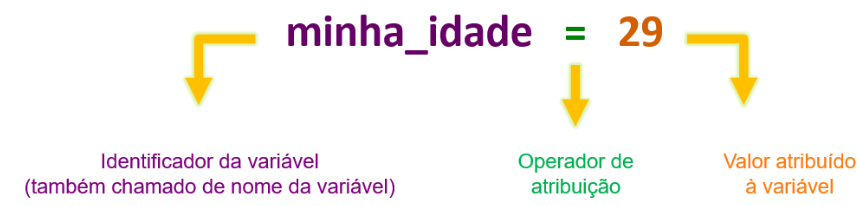


Note que, exceto pela divisão real, quando todos os operandos envolvidos na operação forem números inteiros o resultado será um número inteiro, porém, basta um operando real para que o resultado da operação resulte em um número real.

**Variáveis**

Uma variável é um espaço de memória associado a um identificador, ou seja, um nome, e serve para guardar valores que o programa poderá acessar e modificar. Toda variável possui um identificador único, de forma que possa ser referenciada pelo programador sem ambiguidade em qualquer parte do programa.

Em Python, uma variável é criada no momento em que um valor é atribuído a um identificador válido. A atribuição é feita colocando um identificador à esquerda de um sinal de igual e um valor à direita deste mesmo operador.



O conteúdo de uma variável pode “variar”, ou seja, uma mesma variável pode guardar valores diferentes em momentos diferentes de um programa. Lembre-se: uma variável só guarda um valor por vez, portanto a cada nova atribuição o valor atual será sobrescrito pelo novo.

Um destaque importante da linguagem Python é que o tipo do dado está relacionado ao valor atribuído e não a variável que recebeu esse valor, diferentemente de outras linguagens de programação como C, C++ e Java, dentre outras. Note também que uma mesma variável pode armazenar, inclusive, valores de tipos distintos.

**Identificadores**

Um identificador de uma variável, também referido como nome, é formado por uma sequência de um ou mais caracteres, de acordo com as seguintes regras:

Simplificadamente, pode conter apenas combinações de letras, dígitos e sublinhados (não pode conter símbolos especiais como &, ¨, %, $, #, @, !);

* Não pode iniciar com dígito;
* Não pode ser uma palavra reservada (abordaremos a seguir).

Recomenda-se criar identificadores concisos, porém descritivos:

* idade é melhor que i;
* tamanho\_nome é melhor que tamanho\_do\_nome\_da\_pessoa.

Evite abreviar exageradamente, escreva por extenso para melhorar a legibilidade:

* sobrenome é melhor que sbrnome;
* litros é melhor que ltrs;
* data\_criacao é melhor que dt\_cri.

Existe um guia de estilo para programação em Python, criado e mantido pela própria comunidade mundial da linguagem, com o intuito de fornecer recomendações para facilitar a leitura do código. Esse guia é conhecido como [PEP8](https://peps.python.org/pep-0008/) e embasou a forma como foram escritos os códigos deste material.

No decorrer da disciplina, apresentaremos as partes relevantes do guia para que a sua utilização ocorra de maneira natural, não sendo necessário decorá-lo integralmente. Por enquanto, temos mais uma recomendação relativa à criação de identificadores:

* Use apenas letras minúsculas e sem acentuação, separando as palavras com um sublinhado para melhorar a legibilidade.

Vale relembrar que o Python é uma linguagem case-sensitive, diferenciando letras maiúsculas de minúsculas, portanto o identificador meu\_nome não é o mesmo que Meu\_Nome ou MEU\_NOME.

**Palavras reservadas**

Python possui palavras reservadas, as palavras-chave, chamadas em inglês de keywords, e não podem ser usadas como identificadores, pois têm papel especial na linguagem. O Python 3.9.1 possui 36 keywords, porém a quantidade varia entre as versões, para saber quais são as de sua versão execute na Shell a Codificação 2.3.

>>> help('keywords')

Comentários

Podemos inserir comentários em nossos códigos-fonte, algo útil para ajudar tanto outros programadores que lerão nossos códigos quanto a nós mesmos para recordar a razão de determinadas instruções. Comentários são ignorados na execução do programa, portanto é algo para auxiliar humanos, não computadores. Linhas iniciadas com # são interpretadas como comentários.

**Parte 3 -** [**Funções embutidas, entrada e saída, operadores e expressões aritméticas**](https://sites.google.com/faculdadeimpacta.com.br/lp-p3?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=GdBOkB5hhc8)

**Resumo**

Nesta aula os objetivos são: (I) relembrar expressões; (II) conhecer os operadores de atribuição composta; (III) compreender o conceito de precedência e associatividade de operadores; (IV) entender como Python avalia expressões; (V) descobrir o que são funções e como usar funções integradas; (VI) exibir dados para o usuário; (VII) obter entradas do usuário; (VIII) conhecer tipos de erros comuns em programação.

**Motivação**

Como já aprendemos o que são expressões, relembraremos pontos importantes e conheceremos outros, principalmente para compreender a forma que Python avalia expressões. Poderemos até substituir nossas calculadoras pela Shell, que possui muitas facilidades! Abordaremos funções, um recurso de programação que possibilita, por exemplo, usar códigos de outros programadores em nossos programas. Conheceremos funções para entrada e saída de dados, permitindo programas mais personalizados. Por fim, conheceremos alguns tipos de erros comuns na criação de programas.

Expressões

Uma expressão é uma combinação de operandos com zero ou mais operadores, resultando em um valor, sendo que os operandos podem ser, por exemplo, constantes e variáveis. Para que uma expressão esteja correta, essa combinação precisa ser válida. Um único operando já é considerado uma expressão válida. Também podemos usar parênteses, como na matemática.

>>> 3.1415 # expressão válida, resulta em 3.1415

>>> (2 + 5) # expressão válida, resulta em 7

>>> 3.0 \* 2 # expressão válida, resulta em 6.0

>>> 10 / 0 # expressão válida, porém resulta em erro de aritmética (erro de divisão por zero)

>>> +123 # expressão válida, resulta em 123

>>> 'oi' # expressão válida, resulta em 'oi'

>>> 123+ # expressão inválida, falta um operando à direita

>>> 2 \* 3) # expressão inválida, parênteses desbalanceados

Quando uma instrução contém uma expressão, Python avaliará a expressão, ou seja, encontrará o valor resultante, caso seja válida. Em expressões inválidas, ocorrerão erros que podem variar de acordo com a razão de estarem inválidas.

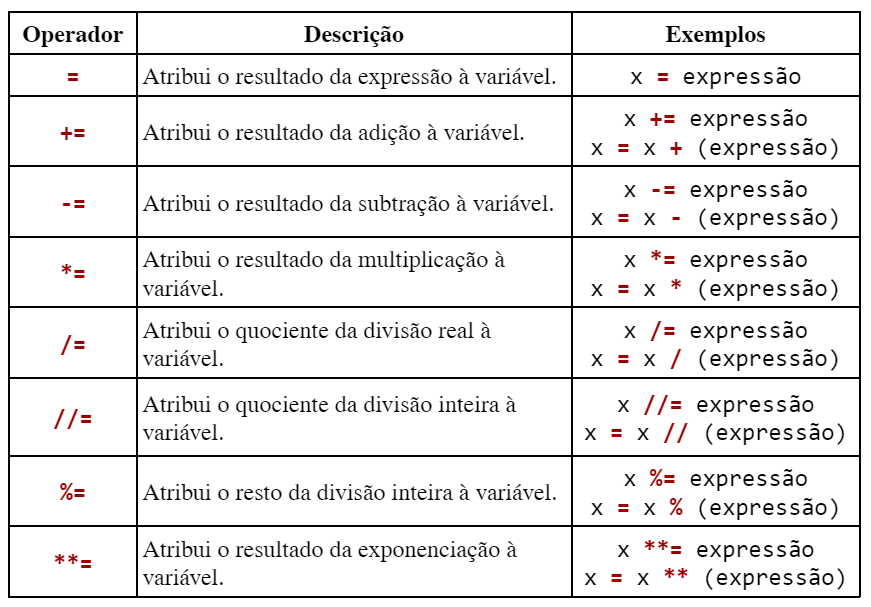
Conhecemos o operador de atribuição simples de Python (=), e entendemos que seu papel é atribuir valores de expressões às variáveis.

>>> x = 10 + 5 # atribui o valor 15 à variável x

É importante compreender que antes da atribuição, a expressão à direita de = será avaliada. Logo, a variável x não receberá a expressão 10 + 5, mas sim o valor resultante da avaliação de 10 + 5, que é 15. O mesmo ocorre em qualquer atribuição.

Em Python, há outros operadores de atribuição, chamados de operadores de atribuição composta, ou atribuição aumentada, usados para tornar algumas instruções mais concisas e, dependendo da maturidade do programador, mais legíveis. Também é possível substituir instruções com atribuição composta por atribuição simples combinada com outro operador.

Podemos dizer que estes operadores são um atalho de sintaxe para quando precisamos acessar o valor de uma variável, realizar uma operação com ele, e atribuir o resultado da operação à mesma variável.



>>> m = 2 # m recebe 2

>>> m \*\*= 5 # m recebe 32

>>> m //= 4 # m recebe 8

>>> m += 1 # m recebe 9

>>> m /= 2 # m recebe 4.5

>>> m %= 2 # m recebe 0.5

>>> m -= 0.2 # m recebe 0.3

>>> m \*= 5 # m recebe 1.5

###### **Codificação 3.3: Uma mesma variável recebendo diversas atribuições.**

>>> a = 10 # a recebe 10

>>> b = 10 # b recebe 10

>>> c = 10 # c recebe 10

>>> a \*= 2 + 3 # a recebe 50

>>> b = b \* 2 + 3 # b recebe 23

>>> c = c \* (2 + 3) # c recebe 50

###### **Codificação 3.4: Atribuição composta e atribuição simples equivalente.**

Talvez você tenha achado estranho que o segundo valor atribuído à variável a seja diferente do segundo valor atribuído à variável b, porém note que na terceira coluna da Tabela 3.1 está definido que a equivalência dos operadores de atribuição composta com a atribuição simples (=) só é atingida ao colocarmos a expressão, originalmente à direita, entre parênteses. Logo, a segunda atribuição à c corresponde à segunda de a.

É importante conhecer esse conjunto de operadores de atribuição, pois são frequentemente usados no mercado de trabalho, em fóruns e livros. Ao longo desta disciplina, usaremos principalmente os operadores =, += e -=.

Novamente, teste a Codificação 3.5 na Shell e verifique se compreendeu a razão dos valores atribuídos às variáveis.

>>> x = 10 # x recebe 10

>>> y = 10 # y recebe 10

>>> z = 10 # z recebe 10

>>> x += 2 \* 3 # x recebe 16

>>> y = y + 2 \* 3 # y recebe 16

>>> z = z + (2 \* 3) # z recebe 16

###### **Codificação 3.5: Atribuição composta e atribuição simples equivalente.**

Você pode ter ficado com dúvida sobre a razão das três últimas atribuições terem o mesmo resultado, mesmo não existindo parênteses na segunda atribuição para y. Isso ocorreu porque a precedência da multiplicação é naturalmente superior à da adição, como na matemática, tornando a ausência de parênteses irrelevante nesta instrução. Para entender plenamente como Python avalia expressões, é necessário estudarmos como essa linguagem de programação define a precedência e associatividade dos operadores.

**Precedência e associatividade de operadores**

Em Python, de modo semelhante à matemática, cada operador possui uma precedência, que é uma indicação de sua prioridade em relação aos outros operadores que estejam na mesma expressão.

Portanto, partes de uma expressão com operadores de maior precedência são avaliadas antes daquelas com operadores de menor precedência. Porém, a precedência natural de um operador pode ser artificialmente modificada com o uso de pares de parênteses, da mesma forma como fazemos na matemática.

Por exemplo, a expressão 4+1\*5 resulta em 9, afinal a multiplicação tem precedência sobre a adição e por isso será avaliada primeiro, mesmo que a adição esteja escrita antes na expressão. Lembre-se que a leitura de qualquer instrução é feita da esquerda para a direita, por isso apontamos que “a adição apareceu antes na expressão”.

Neste segundo exemplo, a expressão (4+1)\*5 resulta em 25, pois a precedência do operador de adição que está entre os parênteses foi alterada, aumentando sua prioridade. Em expressões com pares de parênteses aninhados, como (4+(1\*5))/2, a precedência é do mais aninhado para o menos aninhado. Uma conclusão natural, afinal o par de parênteses mais externo aumentou a prioridade da parte da expressão que está dentro dele e que, por sua vez, contém outro par de parênteses, aumentando a prioridade da parte da expressão dentro deste segundo par, e assim sucessivamente.

Porém, existem casos em que uma expressão pode conter operadores de mesma precedência, por exemplo, 8/4\*2. Repare que nesta expressão não podemos nos basear na precedência para decidir qual parte da expressão Python avaliará primeiro, afinal divisão e multiplicação têm a mesma precedência. No entanto, se a expressão contivesse parênteses, não haveria dúvidas, pois (8/4)\*2 resultaria 4.0, e 8/(4\*2) resultaria 1.0. Para resolver o impasse existe o conceito de associatividade.

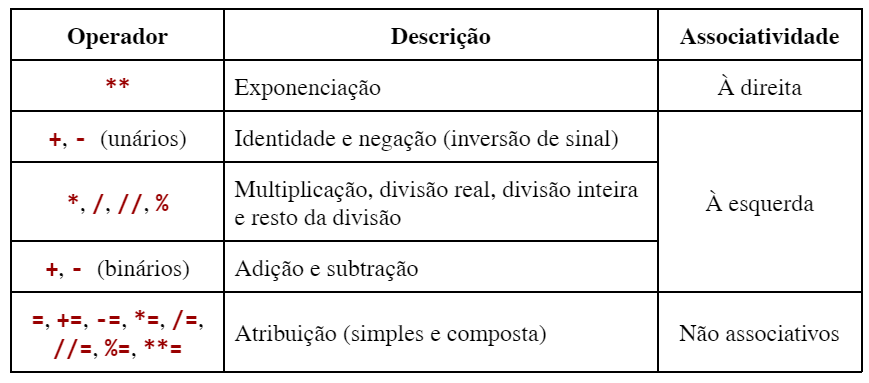
A associatividade é uma propriedade que define como operadores de mesma precedência devem ser agrupados, caso estejam na mesma expressão e sem parênteses que determinem a ordem da avaliação.

Por exemplo, na expressão 8/4\*2 o operando 4 está precedido e sucedido por operadores de mesma precedência, portanto poderia ser associado ao operador da esquerda, impondo que a avaliação começaria por 8/4, ou poderia ser associado ao operador da direita, impondo que a avaliação começaria por 4\*2. Com fundamento na regra de associatividade dos operadores de divisão real e multiplicação, poderemos concluir, sem ambiguidade, o resultado da expressão. Neste exemplo, o resultado será 4.0, porque esses dois operadores são associativos à esquerda.

Os operadores podem ser associativos à esquerda, associativos à direita ou não associativos. Veja a descrição de cada tipo:

* **Associativos à esquerda:** as operações são agrupadas da esquerda para a direita. Portanto, o operando será associado ao operador à sua esquerda;
* **Associativos à direita:** as operações são agrupadas da direita para a esquerda. Portanto, o operando será associado ao operador à sua direita;
* **Não associativos:** as operações não podem ser encadeadas, por uma de duas razões: (I) porque possuem comportamento indefinido, gerando um erro ou; (II) porque representam algo “especial”, uma interpretação diferenciada na linguagem.

Cada linguagem de programação possui suas próprias regras de precedência e associatividade de operadores, por isso é recomendado consultar o manual da linguagem para assegurar que as expressões coincidam com o desejado. A Tabela 3.2 possui uma relação simplificada dessas regras em relação aos operadores vistos até agora. Operadores na mesma linha possuem a mesma precedência e associatividade. A ordem de precedência está decrescente, logo o primeiro operador tem maior precedência e o último tem menor precedência.



**Avaliação de expressões**

De posse do conhecimento sobre precedência e associatividade de operadores e dos demais tópicos estudados até o momento, temos condições de compreender o procedimento de avaliação de expressões executado pelo Python. Para isso, analisaremos algumas expressões, avaliando-as de modo semelhante ao feito em Python.

>>> a = 5

>>> b = 4

>>> c = 9

>>> d = 7

>>> e = 1

>>> f = 2

>>> s = 10

>>> s += a + b \*\* (c - d) / e \* f

>>> s # qual o valor de s?

O par de parênteses prioriza a expressão interna, dando a ela maior precedência, porém, como a operação contém variáveis, primeiro devemos substituí-las por seus respectivos valores. Após ser avaliado, o par de parênteses é descartado:

s += ⇨ s += a + b \*\* 2 / e \* f

Precedência: s += a + 4 \*\*a + b \*\* (9 - 7) / e \* f 2 / e \* f ⇨ s += a + 16 / e \* f

Precedência e associativ.: s += a + 16 / 1 \* f ⇨ s += a + 16.0 \* f

Precedência: s += a + 16.0 \* 2 ⇨ s += a + 32.0

Precedência: s += 5 + 32.0 ⇨ s += 37.0

Único operador: s += 37.0

⇨ Atribuição composta, pode ser reescrita como: s = s + (37.0)

⇨ s = 10 + (37.0) ⇨ s = 10 + 37.0 ⇨ s = 47.0

>>> t = 1 + 2 - 3 + 4 - 5

>>> t # qual o valor de t?

Associatividade: t = 1 + 2 - 3 + 4 - 5 ⇨ t = 3 - 3 + 4 - 5

Associatividade: t = 3 - 3 + 4 - 5 ⇨ t = 0 + 4 - 5

Associatividade: t = 0 + 4 - 5 ⇨ t = 4 - 5

Associatividade: t = 4 - 5 ⇨ t = -1

Único operador: t = -1

>>> u = 2 \*\* 1 \*\* 3

>>> u # qual o valor de u?

Associatividade: u = 2 \*\* 1 \*\* 3 ⇨ u = 2 \*\* 1

Associatividade: u = 2 \*\* 1 ⇨ u = 2

Único operador: u = 2

Note que neste último exemplo foi necessário analisar a associatividade, que no caso do operador de exponenciação é à direita, conforme a Tabela 3.2. O operando 1, por estar entre dois operadores de exponenciação, foi associado àquele à sua direita.

>>> v = ((2 \*\* 1) \*\* 3)

>>> v # qual o valor de v?

Parênteses: v = ((2 \*\* 1) \*\* 3) ⇨ v = (2 \*\* 3)

Parênteses: v = (2 \*\* 3) ⇨ v = 8

Único operador: v = 8

>>> w = (5\*8) + ((9//7) % 2)

>>> w # qual o valor de w?

Parênteses: w = (5\*8) + ((9//7) % 2) ⇨ w = 40 + ((9//7) % 2)

Parênteses: w = 40 + ((9//7) % 2) ⇨ w = 40 + (1 % 2)

Parênteses: w = 40 + (1 % 2) ⇨ w = 40 + 1

Precedência: w = 40 + 1 ⇨ w = 41

Único operador: w = 41

Independente da precedência, a leitura de um código-fonte é feita de cima para baixo, ou seja, da primeira para a última linha de instrução, e cada instrução é lida da esquerda para a direita. Consequentemente, existindo dois pares de parênteses independentes, o mais a esquerda é avaliado primeiro, como visto na Codificação 3.10.

>>> a = b = c = d = 10 # quais os valores das variáveis?

Os operadores de atribuição são não associativos em Python, portanto o encadeamento deles pode gerar uma interpretação especial ou um erro. No caso do operador de atribuição simples (=), há uma interpretação especial, atribuindo o valor mais à direita a todas as variáveis à sua esquerda. Recomenda-se cautela em seu uso, pois pode levar a comportamentos inesperados.

>>> a = b = c = (d = 10) # quais os valores das variáveis?

A Codificação é inválida, pois em Python a atribuição não resulta em valor, logo não há o que atribuir à c. Note que o = entre parênteses está fora do encadeamento.

>>> x = 4

>>> y = 5

>>> x += y += 1 # quais os valores das variáveis?

A Codificação é inválida, pois operadores de atribuição composta são não associativos e quando encadeados geram erro.

**Funções**

Uma função é uma sequência de instruções que executa alguma tarefa específica e que tem um nome. Por exemplo, podemos imaginar uma função que calcula a raiz quadrada de um número não negativo qualquer, ou uma função que coleta dados inseridos pelo usuário, ou uma função que exibe dados na tela.

Podemos dividir as funções em três tipos, de acordo com sua origem:

* **Integradas:** disponibilizadas com a própria linguagem de programação e prontas para uso imediatamente, sem necessidade de instrução adicional;
* **Importadas:** criadas por outros programadores e disponibilizadas para serem incluídas no ambiente de programação, mediante instrução que faça a importação, e então usadas de modo semelhante às integradas;
* **Definidas:** criadas pelo próprio programador e disponíveis para serem utilizadas no código-fonte em que são definidas. É possível distribuí-las para serem importadas em outros códigos-fonte.

Por enquanto focaremos em funções integradas, em próximas aulas criaremos nossas próprias funções que até poderão ser distribuídas para outros programadores!

**Funções integradas**

Uma das vantagens em usar funções integradas, e também das importadas, é a abstração do seu algoritmo, isto é, não precisamos saber detalhes de como foram construídas. O essencial é saber os nomes das funções e o que elas fazem.

Para usar funções integradas, também conhecidas como built-in functions, é necessário chamá-las. Para chamar funções, escreve-se seu nome seguido por um par de parênteses. Ao chamar a função solicitamos sua execução.

>>> print() # chama a função print.

Além de seus nomes e parênteses, a maioria das funções exige um terceiro componente: argumentos. Argumentos são valores inseridos entre os parênteses da chamada e servem para passar dados à função. Os dados passados são necessários para que a função cumpra seu objetivo. Por exemplo, a função abs recebe como argumento um número e retorna o valor absoluto correspondente.

>>> abs(-29) # chama a função abs com o argumento -29.

29 # valor retornado pela função abs.

O valor retornado ou valor devolvido é, como a própria expressão indica, um valor que será dado como retorno pela função, ou seja, a resposta obtida ao chamar a função. Esse valor é o resultado da avaliação da função, algo semelhante ao que ocorre quando avaliamos expressões. Podemos associar os argumentos como as entradas da função e o valor de retorno como a saída da função.

Conceitualmente, nem todas as funções precisam retornar um valor, isso depende do objetivo da função, definido por quem a criou. Não entraremos em todos os detalhes neste momento, mas veremos em outras aulas que, na prática, o Python possui um tratamento especial para funções que não precisariam retornar um valor.

Atente para o fato de que o valor de retorno da função pode ser usado como um valor qualquer, podendo, por exemplo, ser atribuído à uma variável. Veja o exemplo:

>>> x = abs(-29) # atribui o valor de retorno de abs à x.

>>> x + 15

44

Também podemos usar a função em uma expressão com outros operandos, basta lembrar que a função será executada e imaginar que sua chamada será substituída pelo valor retornado, tornando-se um operando como outro qualquer.

>>> abs(-29) + 15 # soma 15 ao valor retornado por abs.

44

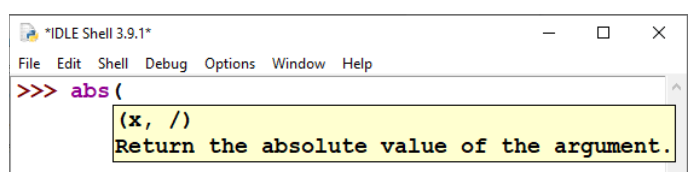
Porém, isso só faz sentido com funções que retornem valor como resposta e que esse valor seja compatível com a expressão onde a função está inserida. Por exemplo, se a função está em uma expressão aritmética, o valor de retorno deve ser um número.

Existem funções que requerem a passagem de mais de um argumento, neste caso eles deverão ser passados na ordem correta e separados por vírgulas. Veja, por exemplo, a função pow que retorna como resposta a potência de uma base elevada a um expoente, semelhante ao operador de exponenciação. Vamos refletir. Para que pow possa realizar a exponenciação, quais os dados necessários? Exatamente! A base e o expoente que se pretende calcular! O primeiro argumento é a base e o segundo é o expoente.

>>> pow(2, 5) # 2 é a base e 5 o expoente

32

Quem determina quais os argumentos e a ordem correta de passá-los à função é seu criador, que pode disponibilizar documentação, geralmente na internet. Também podemos usar o menu Help do IDLE, caso seja uma das funções que acompanham a instalação. A própria Shell e o editor podem ajudar, basta digitar o nome da função com a abertura de parênteses e aguardar alguns instantes.



**Saída de dados**

Você deve ter notado a função embutida print em algumas codificações, apesar de intuitiva por conta do efeito evidente em sua execução, aguardávamos a explicação inicial sobre funções para melhor compreensão. Esta função serve para exibir dados e será bastante usada durante a disciplina.

>>> print('Hello World!') # print com um argumento.

Hello World!

>>> print('2 + 2 =', 2+2) # print com dois argumentos.

2 + 2 = 4

>>> idade = 29

>>> print('Tenho', idade, 'anos') # print com três argumentos.

Tenho 29 anos

Talvez surja a indagação: “por que usar print se ao inserir uma expressão na Shell o resultado é exibido após a execução?”. A Shell do IDLE sempre exibe o resultado de uma expressão após avaliá-la. Porém, isso é uma característica desta ferramenta que foi construída com foco no aprendizado de Python, e não é garantido em outros ambientes, além da Shell ser inadequada para criação de programas completos.

Lembre-se que códigos-fonte sem print, quando executados, não exibem nada.

nome = 'Megan'

nome

idade = 34

idade

Não houve saída? Agora modifique o código-fonte para que fique conforme a Codificação 3.21 e execute-o novamente.

nome = 'Megan'

print(nome)

idade = 34

print(idade)

A função print tem uma característica interessante, pois possui quantidade indeterminada de argumentos, por isso pode ser chamada com zero ou mais argumentos, e o tipo dos argumentos também é arbitrário, podem ser números, strings etc. O Python se encarregará das conversões necessárias para que tudo seja exibido corretamente.

Por padrão, print exibe os valores de seus argumentos e uma quebra de linha '\n', fazendo com que o próximo print exiba seus dados na linha seguinte. Às vezes esse comportamento é indesejado.

print('boa ')

print('noite ')

print('vizinhança')

Se quisermos que apenas o último print quebre a linha, basta acrescentar como último argumento da função um end, seguido de um sinal = com um valor à direita que será exibido no lugar da quebra de linha.

print('boa ', end='')

print('noite ', end='')

print('vizinhança')

No exemplo acima, passamos uma string vazia ('') para o argumento nomeado end, cujo valor padrão é um caractere de quebra de linha ('\n'), não se preocupe em entender os detalhes agora, pois isso será estudado mais a frente, quando aprendermos a definir nossas próprias funções em Python.

A função print não foi projetada para retornar valor, por isso você não a verá como valor de atribuição para uma variável ou usada em uma expressão.

**Entrada de dados**

Para permitir que o usuário do programa forneça dados de entrada, usaremos a função embutida input que, quando executada, faz com que: (I) o programa aguarde um valor de entrada antes de prosseguir com as próximas instruções; (II) converte o valor lido para string e; (III) retorna-o como resposta.

>>> nome = input('Qual o seu nome? ')

Qual o seu nome? Megan

>>> print(nome)

Megan

A função input, pode receber no máximo um argumento, que deve ser uma string e será exibida para o usuário. Note que o retorno de input sempre é uma string e isso pode ser inadequado em algumas situações. Execute no editor a Codificação 3.25, que solicita o nome, o valor do empréstimo e a quantidade de parcelas, e exibe o total da dívida, que é o valor do empréstimo multiplicado pela quantidade de parcelas.

nome = input('Seu nome: ')

valor = input('Valor do empréstimo: ')

parcelas = input('Quantidade de parcelas: ')

print(nome, ', a dívida será de:', valor \* parcelas)

###### **Codificação 3.25: Programa sem conversões dos valores retornados por input.**

Ocorreu um erro, certo? A razão é que as variáveis valor e parcelas receberam strings, consequência do valor retornado por input, e na quarta linha há uma multiplicação entre essas variáveis, ou seja, entre strings, o que é uma expressão inválida. Para resolver esse problema, as entradas deverão ser convertidas para o formato correto, usando as funções integradas int e float.

As funções int e float recebem como argumento um número, ou uma string que represente adequadamente um número do tipo para qual se deseja converter, e devolvem como resposta o valor correspondente convertido para um número inteiro ou em ponto flutuante (real), respectivamente.

nome = input('Seu nome: ')

valor = float(input('Valor do empréstimo: '))

parcelas = int(input('Quantidade de parcelas: '))

print(nome, ', a dívida será de:', valor \* parcelas)

Note que na Codificação 3.26, tanto para a função int quanto para float foi passado como argumento a própria função input, que devolve como resposta uma string. Quando uma função é passada como argumento para outra, a execução inicia pela mais interna (argumento), para que então a mais externa possa ser executada. Vamos analisar a sequência de execução da segunda instrução (input cujo valor de retorno será convertido para float), sendo que o princípio é aplicável também à terceira (input cujo valor de retorno será convertido para int).

Inicialmente input será executada, pois é a função mais interna:

1. valor = float(input('Valor do empréstimo: '))
2. Supondo que o usuário dê a entrada 1000.00, a função input converterá esse valor para string e a retornará como resposta:

valor = float('1000.00')

1. Agora a função float converterá seu argumento, que neste caso é uma string, para o correspondente número real e retornará esse valor.

valor = float('1000.00') ⇨ valor = 1000.0

1. Por fim, o valor 1000.0, de fato um número real representado em ponto flutuante, será atribuído à variável.

valor = 1000.0

**Tipos de erros comuns**

Observamos que erros acontecem! E teremos que lidar com eles, afinal, faz parte da rotina de um programador. Para generalizá-los, classificaremos os erros em três tipos:

* **Erros de sintaxe:** ocorrem por falha na escrita das instruções da linguagem, violando regras e estruturas. Por exemplo, a instrução print 'olá', sem parênteses, é inválida em Python 3, mas é válida em Python 2. Esquecer de completar toda abertura de parênteses com um parêntese de fechamento ou usar um operador binário com apenas um operando são outros exemplos. Geralmente são erros de simples resolução, pois o ambiente de programação costuma indicar instruções com erros de sintaxe ao executar o código;
* **Erros em tempo de execução:** são erros que só aparecem quando o programa é executado, pois a sintaxe está correta. Também podem ser chamados de exceções, indicando que algo excepcional ocorreu, como uma violação do uso esperado do programa. Podem ocorrer, por exemplo, quando o usuário insere um valor inválido ou quando um recurso necessário para a execução do programa está ausente. O exemplo a seguir gera esse tipo de erro:

n = int(input('Número inteiro: ')) # Usuário digita ABC

Caso o usuário digite o que consta no comentário, uma exceção será lançada, indicando que não é possível converter o valor 'ABC' para inteiro.

* **Erros de lógica/semântica:** são erros relacionados ao algoritmo. Nesses casos, o código-fonte pode ser executado e o programa não gerará nenhuma mensagem de erro, mas o resultado não resolve o problema proposto. Algo como “escrever certo a coisa errada”, afinal o algoritmo não soluciona o problema. Estes costumam ser os erros mais difíceis de serem resolvidos.

Podemos resumir os três tipos de erros com a seguinte analogia: (1) em erros de sintaxe, o interpretador não entende o que deve fazer, como falar em um idioma desconhecido; (2) em erros de execução, o interpretador entende o que deve fazer, mas não consegue completar a tarefa, como tentar abrir uma porta com a chave errada; (3) em erros de semântica ou lógica, o interpretador entende o que deve fazer e consegue completar com sucesso todas as instruções, mas isso não resolve o problema especificado, como descarregar um caminhão de tijolos no endereço errado.

**Parte 4 -** [**Operadores e expressões relacionais e lógicas, estruturas de seleção simples e composta**](https://sites.google.com/faculdadeimpacta.com.br/lp-p4?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=y1-1huJ7v4g)

**Resumo**

Nesta aula os objetivos são: (I) apresentar as estruturas de controle de fluxo de execução; (II) conhecer as expressões relacionais e lógicas, assim como seus operadores; (III) entender a associatividade dos operadores relacionais; (IV) compreender o que são expressões equivalentes e complementares; (V) conceituar a avaliação de curto-circuito; (VI) construir estruturas de seleção simples e composta.

**Motivação**

Os primeiros computadores eletrônicos foram chamados de “cérebros eletrônicos”, causando a impressão de que poderiam pensar como humanos. Embora sejam máquinas extremamente complexas, computadores simplesmente executam aquilo que lhes é instruído, sem autonomia para tomar decisões por conta própria… ao menos por enquanto. Assim, não há mais inteligência em um computador do que há nas instruções que lhe são dadas, e sua vantagem está em conseguir executar uma sequência de instruções de modo mais confiável e rápido do que uma pessoa poderia fazer.

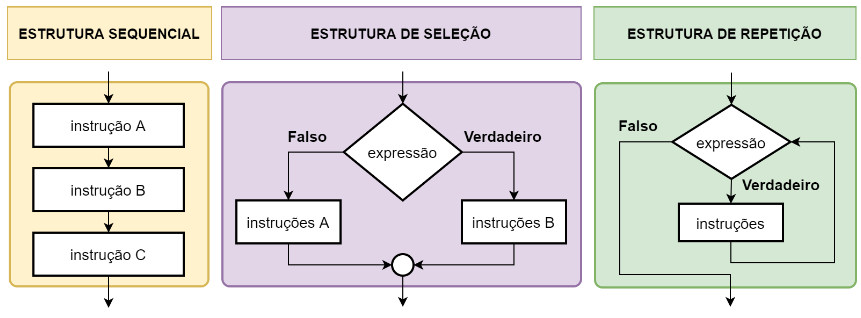
Com estruturas sequenciais de controle de fluxo, já conseguimos resolver diversos problemas simples, mas muitas vezes precisamos escolher se iremos ou não executar determinadas instruções com base em dados que só serão conhecidos em tempo de execução, muitas vezes dependentes de entradas fornecidas pelo usuário do programa. Para isso, veremos as estruturas de seleção, que permitirão que nossos programas reajam de acordo com decisões pré-determinadas e com base no que ocorrerá durante seu fluxo de execução.

**Estruturas de controle**

A ordem na qual as instruções de um programa são executadas pode ser chamada de fluxo de execução, sendo assim, uma estrutura de controle do fluxo de execução é uma forma de controlar em qual ordem as instruções serão executadas.

Fundamentalmente existem três tipos de estruturas para controlar o fluxo de execução de um programa, cuja representação pode ser vista na Figura 4.1. São elas:

* Estrutura sequencial;
* Estrutura de seleção ou condicional; e
* Estrutura de repetição ou iterativa, também chamada de laço.



Por enquanto, os problemas que lidamos requisitaram apenas soluções com estrutura de controle sequencial, em que a ordem que as instruções são executadas é idêntica a ordem que foram escritas. Agora, aprenderemos o segundo tipo de estrutura de controle, a estrutura de seleção ou estrutura condicional, na qual a execução de uma sequência de instruções ficará condicionada ao resultado da avaliação de uma expressão.

Com essa estrutura de controle, poderemos criar programas que decidam, em tempo de execução, quais instruções serão executadas com base em uma expressão que resulte em um valor lógico (também conhecido como valor booleano), ou seja, verdadeiro ou falso. Portanto, antes de tratarmos da estrutura de seleção em si, veremos como criar e avaliar expressões que resultam valores lógicos.

**Expressões relacionais e lógicas**

Anteriormente lidamos com expressões aritméticas, nas quais existem apenas operadores aritméticos e operandos numéricos, quando essas expressões são avaliadas o resultado é um número. Agora estudaremos expressões relacionais e lógicas, em que a avaliação resultará em um valor verdadeiro ou falso, representados em Python pelas constantes True e False, respectivamente.

Expressões relacionais são aquelas em que dois ou mais operandos são comparados para verificar se uma determinada relação entre eles é verdadeira ou falsa. A relação é indicada por meio de operadores relacionais (maior, menor, diferente, etc.) e o resultado é um valor booleano.

>>> idade = 23

>>> pode\_tirar\_cnh = idade >= 18 # Resulta True

A Idade é comparada com o inteiro 18, o resultado da comparação será True se o operando à esquerda for maior ou igual (>=) ao da direita, ou resultará False caso contrário. Note que este exemplo poderia ser aplicado à condição básica para um brasileiro, em condições normais, tentar obter sua carteira de motorista.

Já expressões lógicas são aquelas em que os operandos são valores booleanos e usamos os operadores lógicos (E, OU e NÃO), análogos aos da lógica matemática, para obter um resultado também booleano, indicando se a expressão como um todo é verdadeira ou falsa.

>>> aprovado\_detran = True

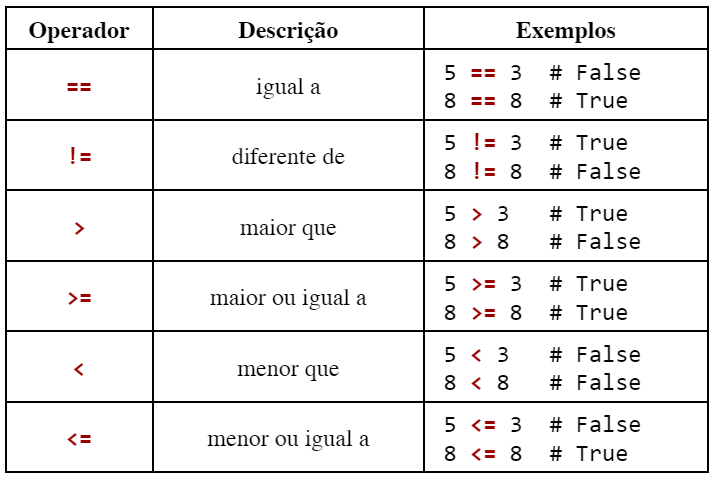
>>> pode\_dirigir = pode\_tirar\_cnh and aprovado\_detran

No Brasil, para estar habilitado a dirigir, não basta ter 18 anos, é preciso também ser aprovado pelo Detran, por isso, na Codificação 4.2, pode\_dirigir receberá True apenas se pode\_tirar\_cnh e (and) aprovado\_detran estiverem com o valor True.

**Operadores relacionais**

Os operadores relacionais em Python são mostrados na Tabela e, evidentemente, são todos binários, pois é necessário relacionar um operando a outro.

Esses operadores relacionais também estão presentes na matemática, mas vale lembrar que a interpretação em Python difere daquelas em equações e inequações matemáticas. Na matemática tais sinais são uma afirmação sobre a (in)equação, se dizemos que 2x + 2 = x+5, sabemos que ambos os lados da equação devem ter o mesmo resultado e podemos então calcular o valor de x.



Sabemos que em Python o sinal de igual simboliza o operador de atribuição simples (=), indicando que a variável à esquerda do operador receberá o valor resultante da avaliação da expressão à direita do operador, consequentemente a equação mostrada no parágrafo anterior é inválida em Python.

Já com os operadores relacionais, é como se fizéssemos uma pergunta em relação a ambos os operandos, como: “eles são iguais?”, “são diferentes?”, “o da esquerda é maior que o da direita?”, “o da esquerda é menor ou igual ao da direita?” e assim sucessivamente, obtendo sempre como resposta um valor verdadeiro ou falso, que, como já mencionado, são representados em Python por True e False.

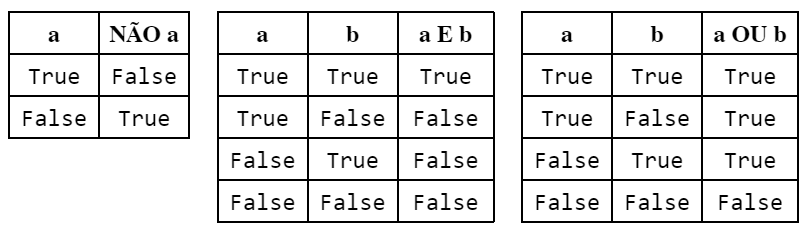
Em Python todos os operadores relacionais possuem a mesma precedência entre si e são não associativos, porém possuem uma interpretação especial quando encadeados, algo que veremos adiante.

**Operadores lógicos**

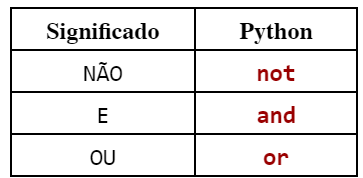
Os operadores lógicos permitem a combinação de comparações ou valores booleanos, possibilitando a construção de expressões mais complexas. Consideraremos três operadores lógicos:

1. NÃO: operador unário que inverte o valor do operando associado, ou seja, é a negação do operando. Negar True resulta em False e negar False resulta em True;
2. E: operador binário que resulta True apenas se ambos os operandos forem True, e False caso contrário. Portanto, basta um operando False para que o resultado da expressão resulte em False;
3. OU: operador binário que resulta True se pelo menos um dos operandos for True, e False caso contrário. Portanto, resulta em False apenas se ambos operandos forem False.

Podemos resumir o funcionamento dos operadores lógicos em uma tabela verdade para cada operador:

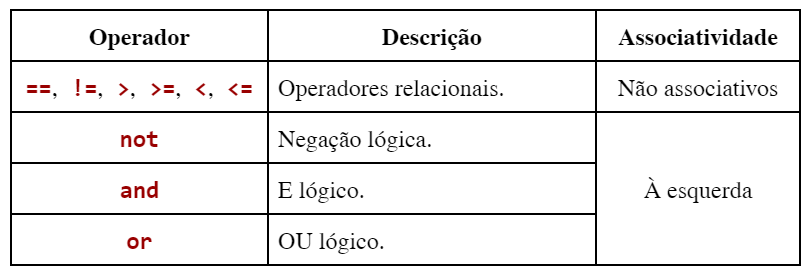


A Tabela contém as palavras reservadas em Python correspondentes a cada operador lógico mostrado até agora.



**Precedência e associatividade**

Operadores relacionais e lógicos em ordem decrescente de prioridade, juntamente com suas associatividades.



Precisamos entender o que ocorre ao digitarmos expressões com operadores relacionais encadeados, afinal são operadores não associativos.

>>> 3 < 10 < 23 # Resulta em True

Se houvesse associatividade dos operadores relacionais em Python, poderíamos ter uma entre duas possíveis interpretações:

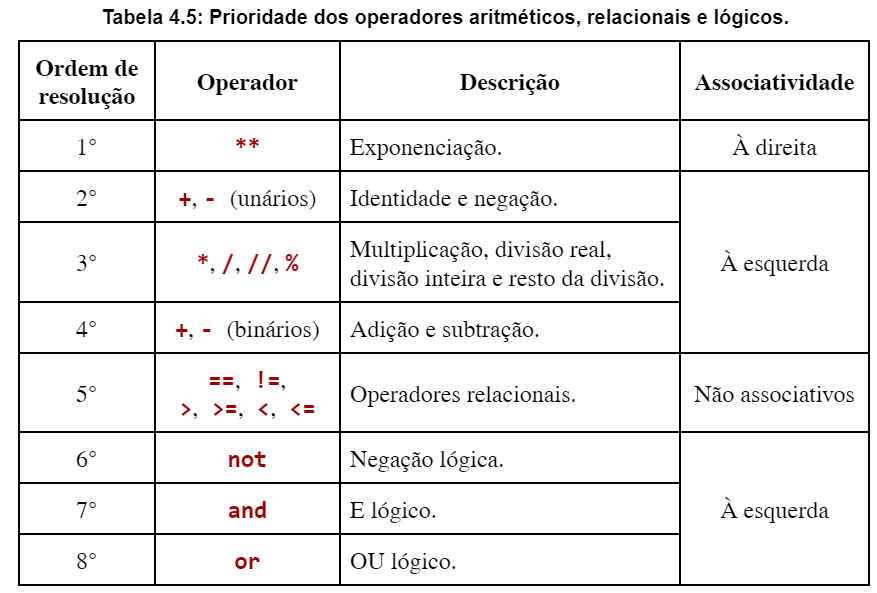
* Com associatividade à esquerda: 3 < 10 < 23 ⇨ True < 23 ⇨ ?
* Com associatividade à direita: 3 < 10 < 23 ⇨ 3 < True ⇨ ?

Podemos observar que a interpretação da comparação de um valor booleano com um valor numérico, ainda que possível em Python (teste na Shell a comparação de True e False com valores numéricos), não faz sentido neste contexto. Outra forma de interpretação seria “o número 10 está entre os números 3 e 23?”. Também poderíamos interpretá-la como de costume na matemática, “3 é menor que 10 que é menor que 23?”

Muitas linguagens de programação não permitem o encadeamento de operadores relacionais, evitando o problema mencionado. Porém Python, visando facilidade de leitura e escrita de código, permite o encadeamento, aproximando o código à notação matemática. Neste caso, o interpretador traduz a expressão como um encadeamento de expressões binárias, repetindo os operandos entre os operadores relacionais e unindo-os com and.

1. 3 < 10 < 23 ⇨ 3 < 10 and 10 < 23
2. 3 < 10 and 10 < 23 ⇨ True and 10 < 23
3. True and 10 < 23 ⇨ True and True
4. True and True ⇨ True

É comum que expressões relacionais e lógicas incluam também operadores aritméticos, então é importante saber a precedência destes operadores não apenas entre si, mas com relação aos demais tipos de operadores. A Tabela 4.5 mostra a ordem em que os operadores são resolvidos pelo Python, de acordo com sua precedência, sendo o operador de maior prioridade resolvido primeiro.



Dentre os operadores vistos até o momento, a atribuição, simples ou composta, tem a menor prioridade independentemente da instrução. Isso é necessário para que a expressão à direita do operador de atribuição possa ser completamente avaliada e o resultado atribuído à variável à esquerda.

**Expressões equivalentes e complementares**

Duas expressões são equivalentes quando possuem exatamente o mesmo significado, apesar de escritas de forma diferente. Por exemplo, na Codificação 4.1, fizemos a comparação da idade do usuário com o número 18, verificando se a idade era maior ou igual a 18, mas poderíamos verificar se 18 era menor ou igual à idade, e chegaríamos ao mesmo resultado em ambas as expressões, conforme a Codificação 4.4.

>>> idade = 23

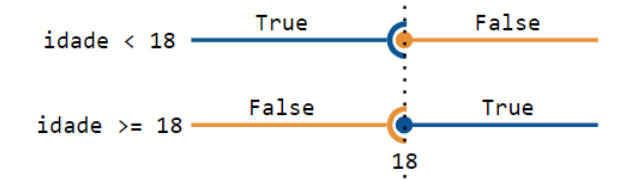
>>> idade >= 18

True

>>> 18 <= idade

True

Duas expressões são complementares quando uma é a negação da outra, ou seja, juntas cobrem todo o espaço possível para os valores envolvidos, sem sobreposição. Por exemplo, a expressão idade < 18 é complementar à idade >= 18, e vice-versa, como ilustrado na Figura 4.2.



Essas expressões, quando juntas, cobrem toda a reta dos números reais sem nenhuma sobreposição (observe que idade < 18 não inclui o número 18), isso significa que para cada possível valor de idade, se uma das expressões for verdadeira, a outra será obrigatoriamente falsa.

>>> 'Maria' != 'Megan' # Expressão A

>>> 'Maria' == 'Megan' # Expressão B (complementar à A)

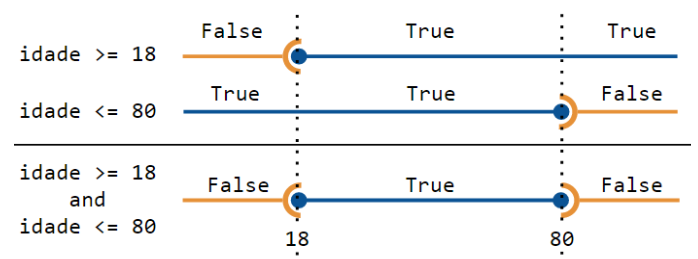
>>> not('Maria' == 'Megan') # Expressão C (complementar à B)

>>> not('Maria' != 'Megan') # Expressão D (complementar à A)

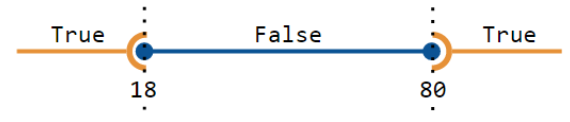
É comum o surgimento de dúvidas ao buscar expressões complementares de expressões que contenham, além de operadores relacionais, operadores lógicos. Veja na Codificação 4.6 um exemplo em que pessoas acima de 80 anos não possam obter carteira de motorista.

>>> pode\_tirar\_cnh = idade >= 18 and idade <= 80

Podemos reescrever esta expressão com uma expressão equivalente 18 <= idade and idade <= 80 ou também 18 <= idade <= 80, no entanto, prosseguiremos com a versão inicial, que inclui o operador lógico “E”, cuja representação na reta dos números reais:



Se quisermos encontrar a expressão complementar a esta, ou seja, a expressão que resulte True para aqueles que não podem obter carta de motorista (negação da expressão inicial), podemos começar negando os valores booleanos na figura e a partir dela, tentar chegar em uma expressão.



Agora, podemos escrever as duas expressões relacionais que resultarão True:

1. idade < 18
2. idade > 80

Observamos que as expressões não podem ser verdadeiras ao mesmo tempo, pois não existe nenhum número que seja simultaneamente menor que 18 e maior que 80, portanto basta que a primeira OU a segunda resulte em verdadeiro. Com isso, chegamos de maneira intuitiva à seguinte expressão: idade < 18 or idade > 80.

Podemos concluir que, ao negar a expressão inicial, obtemos sua expressão complementar e, mais importante, que o operador lógico “E” é substituído pelo “OU” e os operadores relacionais “maior ou igual” e “menor ou igual” são substituídos por, respectivamente, “menor” e “maior”. Algo que pode ser visto no seguinte resumo:

1. Expressão inicial: idade >= 18 and idade <= 80
2. Expressões complementares (em ordem crescente de simplificação):
3. not (idade >= 18 and idade <= 80)
4. not (idade >= 18) or not (idade <= 80)
5. idade < 18 or idade > 80

Se negarmos qualquer uma das expressões complementares, retornaremos à expressão inicial, fazendo as substituições inversas. Neste exemplo, qualquer uma das três expressões é considerada complementar à expressão inicial, no entanto, a última é a mais simples de ser lida por um humano, por ser mais direta usando menos operadores, e menos custosa de ser executada por um computador, por ter menos operações.

**Avaliação de curto-circuito**

Algumas linguagens de programação definem operadores lógicos com a característica de avaliação de curto-circuito, também chamada de avaliação mínima. Operadores lógicos com avaliação de curto-circuito estabelecem que seu segundo operando só será avaliado caso a avaliação do primeiro não seja suficiente para concluir o valor da expressão completa.

Em Python, os operadores lógicos and e or são operadores com avaliação de curto-circuito. Veja o comportamento de cada um deles:

1. **and:** o segundo operando só será avaliado caso o primeiro resulte em True, pois em uma expressão com and, o resultado só pode ser antecipado se o primeiro operando resultar em False, afinal neste caso não importaria o valor do segundo operando, a expressão sempre resultaria em False;
2. **or:** o segundo operando só será avaliado caso o primeiro resulte em False, pois em uma expressão com or, o resultado só pode ser antecipado se o primeiro operando resultar em True, afinal neste caso não importaria o valor do segundo operando, a expressão sempre resultaria em True.

Para exemplificar a consequência e utilidade da avaliação mínima, veja a Codificação 4.7 e a Codificação 4.8, em que essa característica é essencial para o correto funcionamento dos programas.

# Crie um programa que receba dois números naturais e exiba uma

# mensagem indicando se o primeiro é divisível pelo segundo.

a = int(input('Primeiro: '))

b = int(input('Segundo: '))

print(f'{a} é divisível por {b}: {b != 0 and a % b == 0}')

Note na Codificação acima que, sem a avaliação de curto-circuito, caso o usuário inserisse o valor zero para b, o programa geraria um erro pela tentativa de divisão por zero. Porém, a % b == 0 só será avaliada após b != 0 resultar True, logo o erro não ocorrerá. Como teste, inverta a ordem das sub-expressões e insira valor zero para b.

# Crie um programa que receba dois números naturais e exiba uma

# mensagem indicando se o primeiro não é divisível pelo segundo.

a = int(input('Primeiro: '))

b = int(input('Segundo: '))

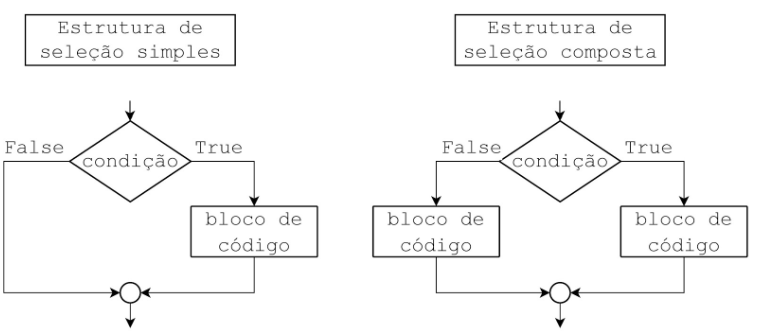
print(f'{a} não é divisível por {b}: {b == 0 or a % b != 0}')

Assim como na Codificação anterior, na Codificação acima, sem a avaliação de curto-circuito, se o usuário inserisse o valor zero para b, o programa geraria um erro. Porém, como a % b != 0 só será avaliada após b == 0 resultar False, o erro não ocorrerá. Novamente, inverta a ordem das sub-expressões e insira valor zero para b.

**Estruturas de seleção ou estruturas condicionais**

São as estruturas usadas para selecionar se um trecho de código será executado com base na avaliação de uma expressão, dizemos que a execução do trecho de código está condicionada ao resultado desta expressão, por isso a expressão de decisão também pode ser chamada de condição. Inicialmente, abordaremos dois tipos de estruturas de seleção.

1. **Seleção simples:** usada quando condicionamos um bloco de código a ser executado apenas quando a condição resultar True. Logo, quando a condição resultar em False o bloco condicionado será pulado/ignorado;
2. **Seleção composta:** usada quando há dois blocos de código condicionados, sendo que um deles será executado apenas quando a condição resultar True e o outro apenas quando a condição resultar False. Portanto, ao executar um bloco, seja qual for, o outro obrigatoriamente será pulado/ignorado.



**Estruturas de seleção simples**

As estruturas de seleção simples são escritas em Python com o uso do comando if, uma das palavras reservadas do Python.

if <condição>:

<bloco de código>

A definição de um bloco de código em Python é dada por dois fatores:

1. O bloco de código é introduzido por “dois pontos”;
2. Instruções pertencentes ao bloco estão no mesmo nível de indentação entre si e indentadas em 4 espaços relativamente à instrução que introduz o bloco.

A indentação é o espaçamento de uma instrução em relação à sua margem esquerda, recomenda-se o uso de caracteres de espaços e não caracteres de tabulação. Na maioria dos editores de código-fonte para Python, incluindo o IDLE, a configuração padrão insere automaticamente quatro espaços ao pressionarmos a tecla [TAB].

Python identifica que o bloco de código pertencente à estrutura de seleção terminou ao encontrar a primeira instrução, subsequente ao início do bloco, que regride em nível de indentação, ou seja, a primeira instrução deslocada de volta para a esquerda. Caso haja instruções com alguma indentação inesperada, incluindo um bloco vazio, isso é marcado como erro de sintaxe, impossibilitando a execução do código até a correção.

idade = int(input('Qual a sua idade? ')) # executado sempre

if idade >= 18:

# Bloco executado apenas quando a condição é verdadeira

print('Você pode ter uma CNH.')

print('Desejamos boa sorte!')

print('Fim') # executado sempre

**Estruturas de seleção composta**

As estruturas de seleção composta são escritas em Python de forma semelhante às estruturas de seleção simples, porém adicionando o comando else, como um complemento ao if.

if <condição>:

<bloco de código 1>

else:

<bloco de código 2>

Após o comando else não é preciso (nem permitido) colocar uma segunda condição, pois será considerada automaticamente a condição complementar àquela do if. Consequentemente, quando a condição do if resultar em True, o bloco 1 será executado e o bloco 2 ignorado e, quando a condição do if resultar em False, ocorrerá o oposto, pois o bloco 2 será executado e o bloco 1 ignorado.

Como else é um complemento do if, jamais deve ser escrito isoladamente. Repare que Python identifica o if correspondente ao else por meio da indentação, a regra é simples: cada else corresponde ao if mais próximo que o antecede no mesmo nível de indentação. Só pode existir um else por if.

Um erro frequente com iniciantes em programação é assumir que todo if deve ser complementado com um else. Não faz sentido! Quando não existem dois blocos de instruções mutuamente exclusivos (ao executar um bloco o outro deve necessariamente ser ignorado), usa-se seleção simples que, como já explicado, não tem else.

idade = int(input('Qual a sua idade? ')) # executado sempre

if idade >= 18:

# Bloco executado quando a condição resulta verdadeiro

print('Você pode ter uma CNH.')

print('Desejamos boa sorte!')

else:

# Bloco executado quando a condição resulta falso

print('Você ainda não completou 18 anos, ', end='')

print('portanto não pode ter uma CNH.')

print('Fim') # executado sempre

# Crie um programa que receba como entrada o valor de um produto

# e a quantidade comprada. O programa concederá 10% de desconto

# para compras com total maior ou igual à R$ 100,00.

valor = float(input('Valor: '))

quantidade = int(input('Quantidade: '))

total = valor \* quantidade

if total >= 100.0:

total = total \* 0.9

else:

total = total

print('Total:', total)

Na Codificação acima quando o valor é inferior a cem reais, não é preciso fazer nada especial, portanto o else é desnecessário, veja que seu bloco contém uma instrução inútil, que consome processamento e não altera o valor da variável.

**Parte 5 -** [**Estruturas de seleção aninhadas e encadeadas**](https://sites.google.com/faculdadeimpacta.com.br/lp-p5?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=wlJ0dQ-7S8w)

**Resumo**

Nesta aula os objetivos são: (I) avançar o estudo de strings; (II) introduzir o conceito de flags booleanas; (III) apresentar as estruturas de seleção aninhadas e encadeadas; (IV) entender o funcionamento do comando elif.

**Strings**

Já utilizamos strings em nossos programas, são usadas para representar cadeias de caracteres, ou seja, textos. Agora poderemos aprofundar um pouco mais sobre esse tipo de dados, aprendendo a concatenar, repetir, comparar e formatar strings.

**Concatenação**

Quando precisamos juntar duas ou mais strings, realizamos uma operação chamada de concatenação de strings. Em Python, essa operação é feita usando o operador +.

>>> 'Bom' + ' ' + 'dia' + '!'

'Bom dia!'

É importante notar que a operação de concatenação utiliza o mesmo operador que vimos anteriormente como adição ou identidade (+), porém tal definição se aplica apenas quando os operandos envolvidos na expressão são numéricos. Isso é comum em Python e também em outras linguagens de programação.

A concatenação não altera seus operandos, mas gera uma nova string que pode, inclusive, ser atribuída a uma variável.

>>> nome = 'Megan'

>>> idade = 34

>>> x = 'Olá ' + nome + '! Você tem ' + str(idade) + ' anos.'

>>> print(x)

>>> print(nome, idade + 5)

A função integrada str recebe como argumento um valor de qualquer tipo e retorna a representação do valor como uma string. Tivemos que usar essa função para converter o valor da variável idade, um inteiro, para string e assim poder fazer a concatenação com as demais strings. Para aprendizado, tente realizar a concatenação sem a conversão e veja o que acontece.

**Comparação**

Podemos realizar comparações entre strings com operadores relacionais, os mesmos usados para verificar relações entre números.

>>> 'maria' != 'MaRiA' # Exemplo 1

True

>>> 'maria' == 'MARIA' # Exemplo 2

False

>>> 'Meg' < 'Megan' # Exemplo 3

True

>>> 'beatriz' > 'bia' # Exemplo 4

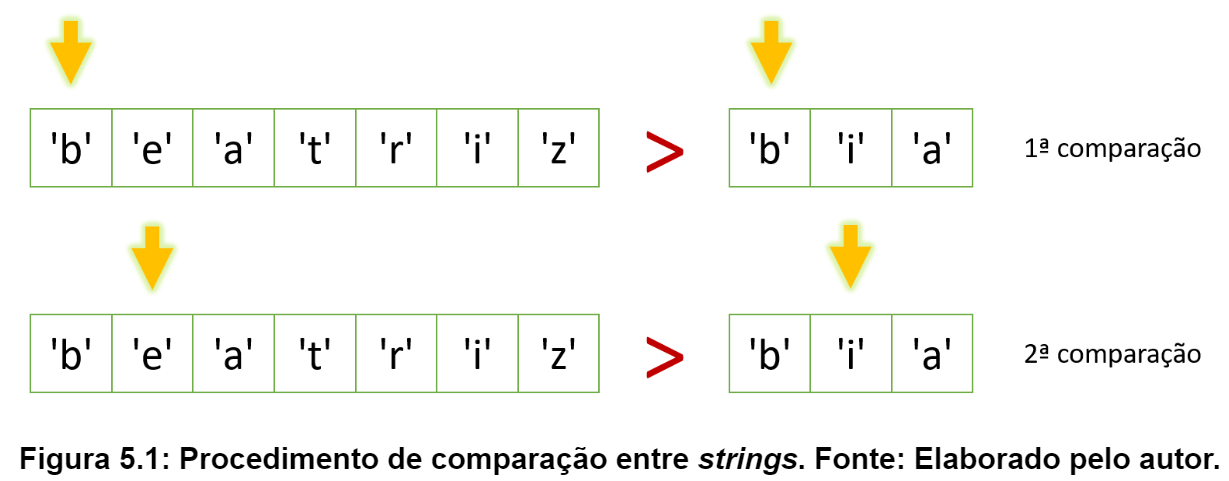
False

Talvez surjam dúvidas sobre o Exemplo 4, em que 'beatriz' > 'bia' resulta em False, neste caso, possivelmente o motivo da expressão do Exemplo 3 resultar em True, também não foi completamente entendido. Para melhor compreensão, precisamos de mais detalhes sobre como a comparação entre strings é realizada.

Em Python a comparação entre strings ocorre caractere a caractere, ou seja, compara-se o primeiro caractere da string à esquerda com o primeiro caractere da string à direita do operador. Enquanto as strings não acabarem (existirem caracteres em ambas as strings que ainda não foram comparados) e os caracteres comparados forem iguais, compara-se o próximo par. A Figura abaixo ilustra o procedimento com base no Exemplo 4.

Strings de tamanhos distintos, ou seja, com quantidades diferentes de caracteres, são consideradas diferentes entre si, assim como strings com caracteres diferentes.

Note que não é o tamanho da string que determina se ela é menor ou maior que outra, mas sim seu conteúdo. Portanto, 'beatriz' não é maior do que 'bia', pois o caractere 'e' de 'beatriz' não é maior do que 'i' de 'bia'.



Mas qual a razão para um caractere ser identificado como “menor” ou “maior” do que outro? Intuitivamente, podemos assumir a ordem alfabética, em que um caractere X é considerado menor do que um caractere Y se X antecede Y no alfabeto. Porém isso é apenas uma simplificação, que não é suficiente para o exemplo abaixo.

>>> 'oi!' < 'oi?'

True

Na Codificação acima, as strings têm o mesmo tamanho e conteúdo, exceto pelo último caractere. Como saber se '!' é menor do que '?' se não estão em nosso alfabeto? Usaremos a função integrada ord que recebe como argumento um caractere e devolve como resposta um número natural correspondente, conforme abaixo:

>>> ord('!')

33

>>> ord('?')

63

Quando caracteres são comparados, internamente ocorre a comparação entre números naturais que correspondem a esses caracteres. Logo, é fácil perceber que 33 é menor do que 63, justificando a resposta de 'oi!' < 'oi?'. A dúvida é “o que são esses números que correspondem aos caracteres?”. São códigos da tabela Unicode.

***Unicode*** é uma padronização que permite que computadores representem e manipulem caracteres de quase todos os sistemas de escrita existentes. Na tabela Unicode existem códigos associados a símbolos, sendo que cada código está mapeado para apenas um símbolo. Há milhares de códigos, basta consultá-los em: <https://unicode-table.com/pt/>.

Não nos aprofundaremos neste tópico, sendo suficiente saber que os códigos são sequenciais e que nosso alfabeto está disposto nesta tabela da forma como estamos habituados, consequentemente 'A' é menor que 'B', 'B' é menor que 'C' e assim sucessivamente. Também é importante saber que as letras maiúsculas são listadas nesta tabela antes das letras minúsculas, logo os códigos das maiúsculas são menores que os das minúsculas, justificando o resultado abaixo.

>>> 'ANA' < 'ana'

True

>>> print('A =', ord('A'), '| a =', ord('a'))

A = 65 | a = 97

Por fim, temos a função integrada chr que realiza o inverso da função ord, isto é, recebe como argumento um número natural representando um código Unicode e retorna como resposta o caractere correspondente. Veja o exemplo abaixo.

>>> print(chr(48), chr(49), chr(50), chr(51), chr(9733))

0 1 2 3 ★

**Formatação**

Às vezes é necessário formatar strings, estipulando, por exemplo, quantidade de colunas, alinhamento, quantidade de casas decimais para representar floats, etc. O Python possui algumas formas para criar strings formatadas, dentre as quais citamos:

* **Interpolação:** formatação utilizando o operador de interpolação (%), com sintaxe próxima ao estilo usado na função printf da Linguagem C. É compatível com todas as versões do Python, mas seu uso é desencorajado para novos projetos;
* **Método format:** método de strings em que os argumentos são formatados e inseridos nos marcadores identificados por pares de chaves. Aceito a partir do Python 2.6. É um avanço em relação à interpolação, com maior controle sobre a formatação, mas aumenta a verbosidade podendo reduzir a legibilidade do código. É recomendado para projetos Python que antecedem à versão 3.6;
* **f-strings ou “strings literais formatadas”:** é a abordagem mais recente para formatação de strings, basicamente uma string comum prefixada com a letra f ou F (antes da abertura de aspas/apóstrofos) e com valores ou expressões entre pares de chaves que serão formatados e inseridos na string. Compatível com as versões do Python a partir da 3.6, é a forma recomendada para novos projetos;
* **Template strings:** é uma forma especial de criação de strings e é recomendada para situações em que a formatação da string será fornecida pelo usuário final, pois seu funcionamento traz algumas proteções contra injeção de código malicioso. Não será abordada neste curso.

Nesta disciplina optamos pelas f-strings, pois possuem vantagens em legibilidade e desempenho, além de serem indicadas pela comunidade (Bader, 2018).

Por ser um tópico extenso, aprenderemos inicialmente a usar f-strings com as formatações mais recorrentes em nossos programas. Começaremos com um exemplo simples.

>>> mensagem = f'2 + 3 = {2 + 3}, entendeu?'

>>> mensagem

'2 + 3 = 5, entendeu?'

Note que a expressão entre o par de chaves é avaliada e o resultado inserido no mesmo ponto em que está escrita na string. A avaliação ocorre em tempo de execução, o que permite o uso de f-strings com variáveis e expressões.

>>> salario = float(input('Seu salário: '))

Seu salário: 1000.00

>>> f'20% a mais em R$ {salario} dará R$ {salario \* 1.2}'

'20% a mais em R$ 1000.0 dará R$ 1200.0'

Vimos que utilizar f-strings para formatação melhora a legibilidade do código quando comparado com a concatenação ou com a passagem de diversos argumentos para a função print. Entretanto, só isso não seria suficiente para problemas que exigem formatação mais elaborada, então veremos algumas possibilidades de manipulação de texto com o uso de f-strings.

No Codificação acima, seria útil exibir os valores monetários com duas casas decimais. Para especificar a formatação de um valor em uma f-string, o pós-fixamos com : e acrescentamos alguns especificadores.

Para formatar um dado do tipo float, podemos usar o seguinte padrão de formatação: f'{<valor>:<colunas>.<decimais>f}'. Onde:

* **colunas:** a quantidade mínima de colunas reservadas para o valor na string formatada, note que cada caractere do valor ocupa uma coluna, inclusive o ponto. Se omitido, assume-se a quantidade mínima para expressar o número;
* **decimais:** o número total de casas decimais que serão representadas na string, a última casa à direita é arredondada. Se omitido, o padrão será de seis casas;
* **f:** indica que a formatação será feita para o tipo float, podendo haver uma conversão automática entre tipos compatíveis, como o int.

Veja a Codificação abaixo, em seguida reescreva a Codificação acima, porém formatando os valores em reais com duas casas decimais.

>>> pi = 3.14159265

>>> f'{pi:f}' # sem especificação o padrão são 6 casas decimais

'3.141593'

>>> f'{pi:.3f}' # note o arredondamento na última casa decimal

'3.142'

>>> f'{pi:7.3f}' # 7 colunas com 3 reservadas para parte decimal

' 3.142'

Existem outros especificadores para formatação de float, como exibição em notação científica, e também para outros tipos, como int e string. A lista completa com as explicações e exemplos de uso pode ser vista na documentação oficial do Python([Format Specification Mini-Language](https://docs.python.org/3/library/string.html#format-specification-mini-language) (Python Software Foundation, 2021)).

Assim como visto com valores do tipo float, podemos definir o tamanho mínimo de colunas para qualquer string formatada, independente do tipo do dado. O padrão para isso é f'{<valor>:<alinhamento><colunas>}'.

>>> nome = 'Megan'

>>> f'{nome:10}'

'Megan '

Nos exemplos de formatação de float, percebemos que o alinhamento padrão de números é à direita, já para valores do tipo string o padrão é à esquerda, mas em ambos os casos podemos controlar o alinhamento adicionando um dos símbolos da Tabela 5.1, de acordo com o padrão f'{<valor>:<alinhamento><colunas>}'.

##### 

| **Símbolo** | **Alinhamento** |
| --- | --- |
| **<** | À esquerda. |
| **>** | À direita. |
| **^** | Centralizado. |

Por padrão, colunas não ocupadas por caracteres do valor são preenchidas com espaços. Para personalizar o caractere de preenchimento adiciona-se outro especificador antes do alinhamento: f'{<valor>:<caractere><alinhamento><colunas>}'.

>>> f'{"à esquerda":<30}'

'à esquerda '

>>> f'{"à direita":>30}'

' à direita'

>>> f'{"centro":^30}'

' centro '

Observe que o ponto (.) usado na formatação de cada uma das variáveis na Codificação abaixo tem significado distinto, pois estão em posições diferentes no padrão que define a formatação.

>>> item = 'camiseta'

>>> preco = 49.9

>>> f'{item:.<20} R$ {preco:.2f}'

'camiseta............ R$ 49.90'

**Flags booleanas**

O termo flag vem do inglês e significa “bandeira” e bandeiras servem para sinalizar algo, então uma flag booleana é usada para sinalizar algo no código. Podemos usá-las para facilitar a leitura de estruturas de seleção e também de repetição indefinida, que veremos mais à frente no curso.

Uma flag booleana nada mais é do que uma variável que guarda o resultado de uma comparação ou expressão relacional/lógica. Assim, ao acessarmos o valor dessa variável ao longo do fluxo de execução do código, podemos saber se a condição em questão foi atendida ou não, podemos pensar em uma bandeira levantada (True) ou não (False). Veja a Codificação 5.15, base para a Codificação 5.16 que usará flag booleana.

idade = int(input('Qual a sua idade? '))

if idade >= 18:

print('Você é maior de idade.')

**Codificação 5.15: Programa que verifica se o usuário é maior de idade (sem *flag*).**

Na codificação acima, o bloco de código do if só será executado quando o usuário fornecer como entrada um inteiro maior ou igual a 18. Podemos reescrever este exemplo utilizando uma flag booleana para guardar o “resultado da condição” que queremos averiguar. Recomenda-se que o nome da variável seja representativo em relação ao que será avaliado. O código implementado está na Codificação 5.16.

idade = int(input('Qual a sua idade? '))

maior\_de\_idade = idade >= 18

if maior\_de\_idade:

print('Você é maior de idade.')

**Codificação 5.16: Programa que verifica se o usuário é maior de idade (com *flag*).**

Note que a Codificação 5.15 e Codificação 5.16 são equivalentes, porém, nesta última, não há operadores na condição do if, apenas uma variável com valor booleano. Vamos incluir mais condições para podermos visualizar as vantagens no uso de flags booleanas. Insira a Codificação 5.17 no editor, teste a sua execução e em seguida pense em como substituir as condições por flags para facilitar a leitura do código.

idade = int(input('Qual a sua idade? '))

cnh = input('Você tem CNH? (s/n)')

if idade >= 18 and cnh == 's':

print('Você pode dirigir.')

**Codificação 5.17: Programa que verifica se o usuário pode dirigir (sem *flag*).**

A Codificação abaixo é equivalente à Codificação acima, mas com flags. Não existe apenas uma versão válida, o importante é usar identificadores significativos para o problema, minimizando ambiguidades e prezando pela simplicidade.

idade = int(input('Qual a sua idade? '))

cnh = input('Você tem CNH? (s/n)')

maior\_de\_idade = idade >= 18

possui\_cnh = cnh == 's'

pode\_dirigir = maior\_de\_idade and possui\_cnh

if pode\_dirigir:

print('Você pode dirigir.')

O uso de flags booleanas pode facilitar a legibilidade do código, aproximando a leitura do código-fonte ao que normalmente falaríamos em nosso idioma natural, mas é preciso cuidado, pois a utilização de nomes inadequados pode acarretar erros de interpretação sutís, de difícil correção. Logo, é extremamente importante que haja uma relação direta entre a condição/expressão avaliada e o nome escolhido para a flag.

Note que a escolha entre usar ou não flags booleanas dependerá da situação e experiência da programadora/programador, algo naturalmente desenvolvido com treino. Cabe a cada um encontrar um equilíbrio em seu uso, assim como qualquer recurso em programação, portanto, procure praticar todas as diferentes formas, com foco primeiramente em entender como funcionam e não em sua aplicabilidade definitiva. Cada recurso aprendido é um novo instrumento em sua caixa de ferramentas.

As principais vantagens de se usar flags booleanas são:

* Melhorar legibilidade do código, facilitando a construção e o entendimento de condições mais complexas;
* Guardar a informação relativa a uma determinada condição avaliada durante a execução do código;
* Aprimorar o desempenho do programa quando uma condição deve ser avaliada diversas vezes, por exemplo em laços de repetição, assunto ainda não abordado.

Vale lembrar que o uso exagerado de flags booleanas, seja colocando-as em todas as situações possíveis, seja usando-as muitas vezes ao longo do código, pode resultar no efeito contrário ao esperado, elevando a complexidade do código e piorando sua legibilidade, pois pode aumentar desnecessariamente a verbosidade do código.

O excesso de verbosidade pode ocorrer especialmente em problemas simples, como os vistos nos exemplos desta aula. Em uma situação real na qual fosse necessária a verificação de idade de uma pessoa, boa parte dos desenvolvedores provavelmente realizaria a comparação diretamente na condição do if, pois idade >= 18 é uma comparação com interpretação simples (conhecendo o contexto entende-se que o objetivo é saber se uma pessoa é maior de idade), então o uso da flag maior\_de\_idade aumenta desnecessariamente a complexidade do código.

Um questionamento comum é relativo ao consumo extra de memória pelas flags, o que poderia piorar o desempenho da solução. Quanto a isso, é seguro dizer que para a maioria das aplicações, o consumo extra é insignificante.

Na implementação do Python em C, são alocados 28 bytes para uma variável com o valor True e 24 bytes para o valor False, mais alguns bytes para guardar o nome da variável em si. Portanto, na maioria das situações, é desnecessária a preocupação com a quantidade de memória gasta com variáveis simples, e a decisão sobre o uso de uma flag booleana deve ser feita com base na legibilidade do código, com foco em deixá-lo mais simples e fácil de entender.

**Estruturas de seleção aninhadas**

A única regra para o código dentro de um bloco de if ou else é que ele deve ser um código válido em Python. Portanto, é possível que exista uma estrutura de seleção dentro de outra. Basta seguir as mesmas regras que vimos na definição de blocos de código em Python e de como if e else se relacionam:

1. O bloco de código é introduzido por “dois pontos”;
2. Instruções pertencentes ao bloco estão no mesmo nível de indentação entre si e indentadas em 4 espaços relativamente à instrução que introduz o bloco.

Para compreender as estruturas de seleções aninhadas e a necessidade de usá-las, veremos a Codificação abaixo, um programa que solicita a idade do usuário e, de acordo com a resposta, realiza outras perguntas para descobrir se ele pode dirigir. Tente entender os possíveis caminhos dentro deste código e desenhe um mapa onde cada if é uma bifurcação na “estrada” do fluxo de execução e cada instrução é uma “cidade” neste mapa. Se um if não possui um comando else associado, podemos interpretá-lo como uma bifurcação cujo lado else não contém nenhuma “cidade”.

idade = int(input('Qual a sua idade? ')) #1

if idade >= 18: #2

cnh = input('Você tem CNH? (s/n)') #3

if cnh == 's': #4

cnh\_valida = input('Ela está válida? (s/n)') #5

if cnh\_valida == 's': #6

print('Você pode dirigir.') #7

else:

print('Você precisa renovar sua CNH.') #8

else:

print('Você precisa tirar a CNH para dirigir.') #9

else:

print('Você ainda não pode dirigir.') #10

tempo\_falta = 18 - idade #11

if tempo\_falta <= 2: #12

print('Falta pouco tempo para você poder dirigir.') #13

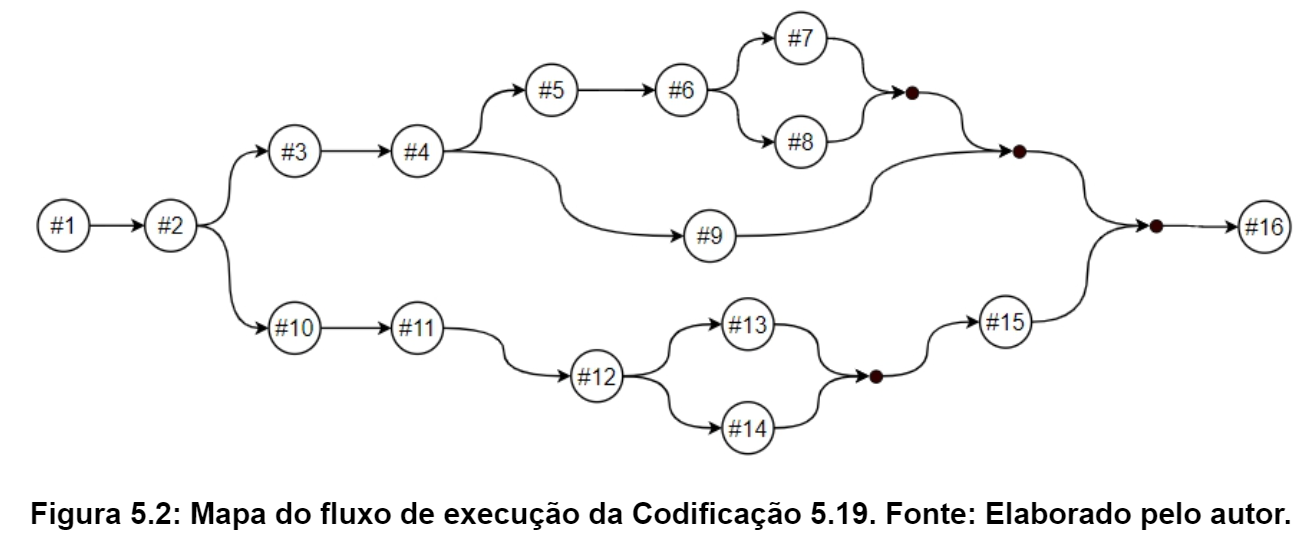
else:

print('Demorará para você poder dirigir.') #14

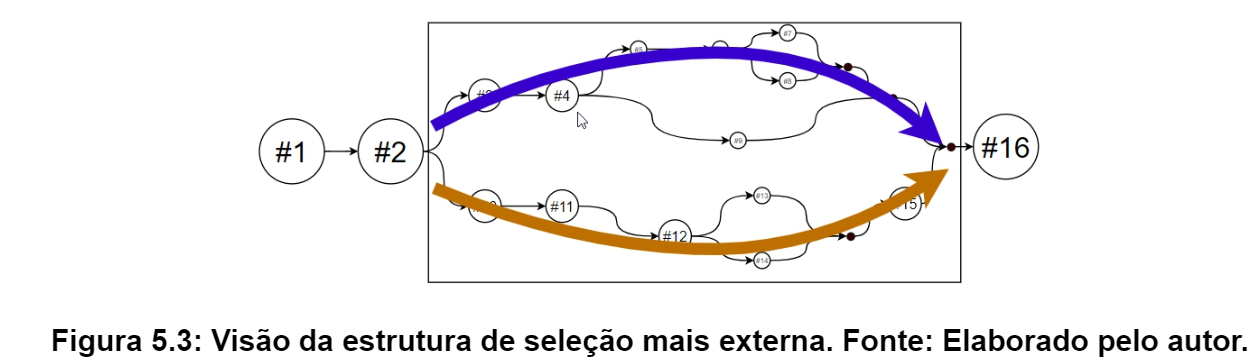
print('Mas você pode dirigir um kart se quiser.') #15

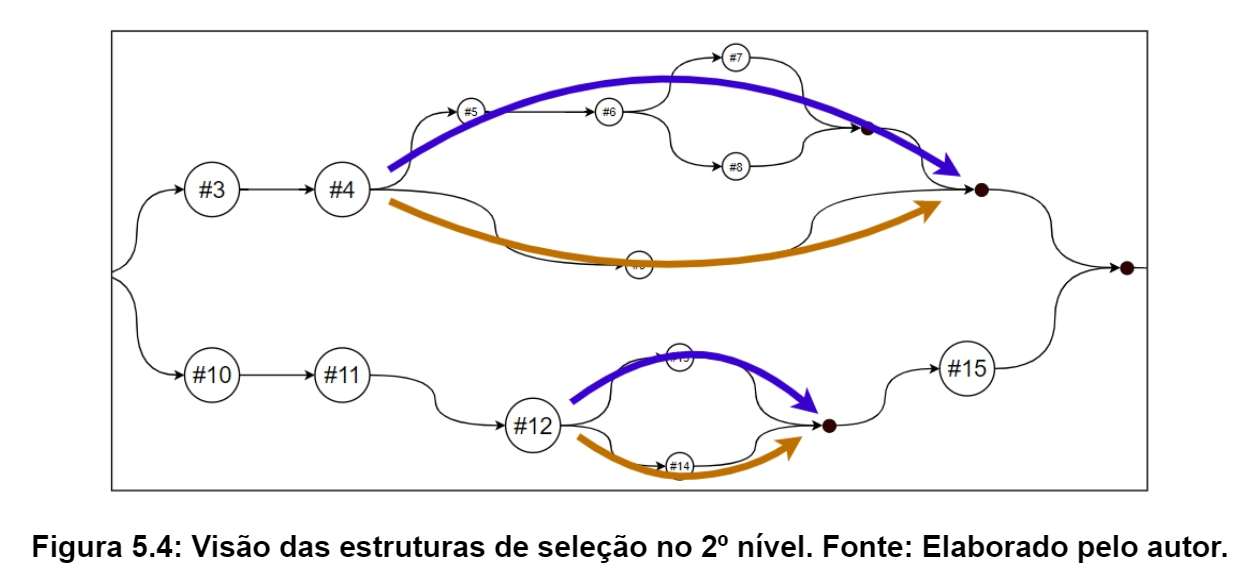
print('Fim!')

Após desenhar, veja se consegue dizer quando cada instrução é executada, isto é, qual o conjunto de entradas que precisam ser dadas para que o fluxo de execução passe por aquela instrução em particular. Em seguida, compare seu mapa com o da Figura 5.2.



Uma forma de entender um trecho de código mais complexo é com a construção destes mapas ou de fluxogramas. Podemos pensar inicialmente em um mapa visto de longe, com apenas 2 caminhos, como mostra a Figura 5.3, e conforme aumentamos o “zoom” neste mapa, caminhos menores surgirão dentro dos caminhos maiores, e assim sucessivamente para cada novo bloco interno, como podemos ver na Figura 5.4. Dessa forma, analisamos o código a partir do bloco mais externo para o mais interno.





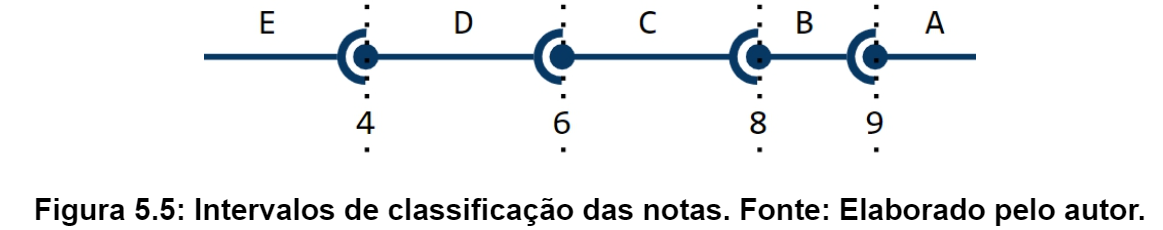
Você possivelmente deve ter concluído que em estruturas de seleção aninhadas a seleção mais externa funciona como um filtro para suas seleções internas, ou seja, para que o fluxo de execução do programa atinja o ponto de uma seleção interna, necessariamente já deve ter passado pela decisão da seleção mais externa.

**Estruturas de seleção encadeadas**

As estruturas de seleção encadeadas são um subtipo das estruturas aninhadas. Dizemos que uma estrutura de seleção A está encadeada em uma estrutura de seleção B, quando A está dentro do bloco else de B, e A é a única instrução desse bloco.

Quando isso ocorre, dizemos que as seleções estão encadeadas entre si, isto é, cada seleção do encadeamento só é analisada se todas as anteriores resultarem em False, e no momento que uma das condições é avaliada True, seu bloco de código é executado e as demais condições subsequentes, se existirem, serão ignoradas.

Vejamos outro exemplo com um programa que lê a nota de um aluno em uma escala de 0 a 10, e converte-a para uma escala usando letras, como mostra a Figura 5.5.



O programa correspondente está na Codificação abaixo. Observe que:

* A primeira bifurcação no fluxo de execução ocorre no if da instrução #2. Caso a condição nota >= 9 resulte True o fluxo será direcionado para #3, caso contrário será direcionado para a primeira instrução do bloco else correspondente, ou seja, a instrução #4, garantindo que nota é menor que 9;
* O bloco do else só é executado quando a condição do if correspondente resultar False. No exemplo anterior, todas as instruções a partir da #4 até #10 fazem parte do bloco do primeiro else, evidente pela indentação, portanto, se #3 é executada, a próxima instrução deste caminho será #11;
* Caso a nota seja menor do que 9, então a instrução #4 será executada, e haverá mais uma divisão de caminhos, entre notas maiores ou iguais a 8 e a negação disso, notas menores que 8. Aqui, vale ressaltar que não é necessário verificar se nota < 9, pois ao executar #4, sabemos que a condição da instrução #2 resultou em False e, portanto, o valor de nota é obrigatoriamente menor que 9.

nota = float(input('Qual a nota? ')) #1

if nota >= 9: #2

letra = 'A' #3

else:

if nota >= 8: #4

letra = 'B' #5

else:

if nota >= 6: #6

letra = 'C' #7

else:

if nota >= 4: #8

letra = 'D' #9

else:

letra = 'E' #10

print(f'Sua letra é: {letra}') #11

**O comando elif**

O comando elif é uma fusão dos comandos else e if quando se encontram na situação de uma estrutura de seleção encadeada, como acabamos de ver. Ele é mais um exemplo de "açúcar sintático”, introduzido na linguagem para facilitar a escrita e, principalmente, a leitura do código.

A Codificação abaixo contém lado a lado a sintaxe de uma estrutura de seleção encadeada sem elif (à esquerda) e com elif (à direita). Os dois códigos apresentam os mesmos fluxos de execução, veja na Figura 5.6 que a única diferença está na sintaxe.

if <condição 1>: #1

<bloco 1>

else: #2

if <condição 2>: #3

<bloco 2>

else: #4

<bloco 3>

if <condição 1>: #1

<bloco 1>

elif <condição 2>: #2 + #3

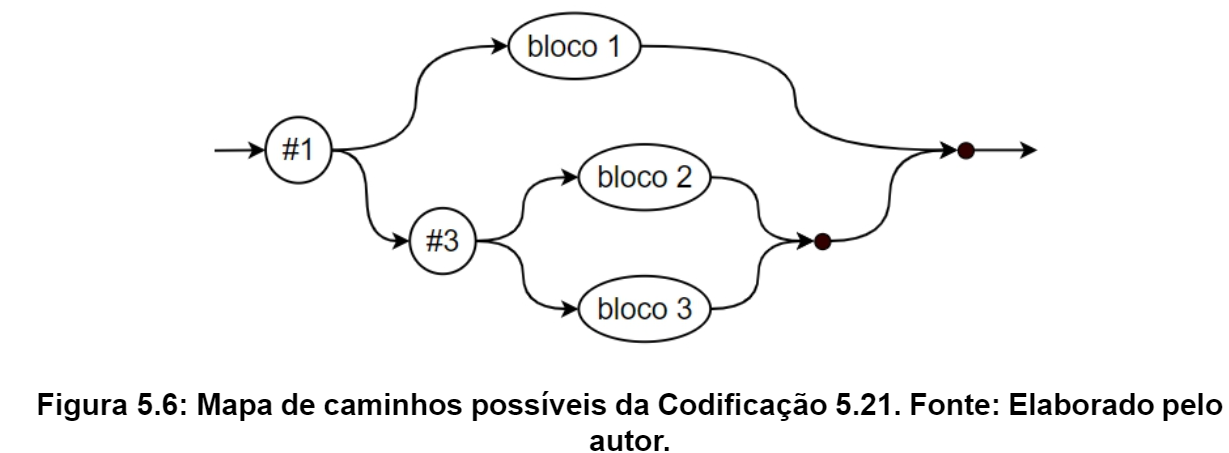
<bloco 2>

else: #4

<bloco 3>

Na Codificação acima, note que ao unirmos #2 com #3 (else e if) formarmos o elif, e as consequências dessa união são:

* O comando elif começa com um else, portanto é sempre um complemento de um if ou outro elif, e não pode existir isoladamente;
* O comando elif termina com um if, portanto podemos complementá-lo com um else ou outro elif na sequência, e assim por diante sem quantidade limite;
* O comando else sempre encerra uma estrutura de seleção, seja ele usado após um if ou elif;
* Esse encadeamento garante que o bloco executado será aquele que estiver dentro da primeira estrutura de seleção que a condição resultar True, e que as seleções posteriores no encadeamento serão puladas.



Podemos então aplicar este novo comando à Codificação abaixo. Observe como o código permanece com apenas dois níveis de indentação, mais conciso, simples e com melhor legibilidade.

nota = float(input('Qual a nota? '))

if nota >= 9:

letra = 'A'

elif nota >= 8:

letra = 'B'

elif nota >= 6:

letra = 'C'

elif nota >= 4:

letra = 'D'

else:

letra = 'E'

print(f'Sua letra é: {letra}')

1) Crie um programa que solicite ao usuário um número de 1 à 7 e exiba o dia da semana correspondente. Assuma que a semana começa no domingo (1) e termina no sábado (7). Use apenas seleção simples, ou seja, sem else nem elif.

dia\_semana = int(input('Digite um número de 1 a 7 para o dia da semana: '))

if dia\_semana == 1:

print('Domingo')

if dia\_semana == 2:

print('Segunda')

if dia\_semana == 3:

print('Terça')

if dia\_semana == 4:

print('Quarta')

if dia\_semana == 5:

print('Quinta')

if dia\_semana == 6:

print('Sexta')

if dia\_semana == 7:

print('Sábado')

2) Refaça o exercício anterior, agora utilizando a seleção encadeada, mas sem usar o comando elif, de modo que seja observada a importância do alinhamento correto de indentação entre os diversos comandos if e else.

if dia\_semana == 1:

print('Domingo')

else:

if dia\_semana == 2:

print('Segunda')

else:

if dia\_semana == 3:

print('Terça')

else:

if dia\_semana == 4:

print('Quarta')

else:

if dia\_semana == 5:

print('Quinta')

else:

if dia\_semana == 6:

print('Sexta')

else:

if dia\_semana == 7:

print('Sábado')

3) Refaça o exercício anterior, agora utilizando elif.

dia\_semana = int(input('Digite um número de 1 a 7 para o dia da semana: '))

if dia\_semana == 1:

print('Domingo')

elif dia\_semana == 2:

print('Segunda')

elif dia\_semana == 3:

print('Terça')

elif dia\_semana == 4:

print('Quarta')

elif dia\_semana == 5:

print('Quinta')

elif dia\_semana == 6:

print('Sexta')

elif dia\_semana == 7:

print('Sábado')

**Parte 6 -** [**Criação de funções e escopo de variáveis**](https://sites.google.com/faculdadeimpacta.com.br/lp-p6?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=ao4-zw2btT4)

**Resumo**

Nesta aula os objetivos são: (I) conhecer as funções importadas; (II) definir nossas próprias funções; (III) entender a relação entre argumentos e parâmetros de funções; (IV) compreender o que é valor de retorno; (V) introduzir o conceito de escopo de variáveis; (VI) escrever documentação nas funções; (VII) organizar o código-fonte.

**Motivação**

Conhecemos funções pré-definidas e sabemos como usá-las, mas não podemos nos limitar a elas, pois frequentemente teremos problemas para os quais não existem funções prontas. Seria inviável criar e integrar à linguagem uma função para cada problema existente, basicamente por dois motivos: 1) novos problemas surgem o tempo todo e; 2) o pacote de instalação do Python ficaria cada vez maior e eventualmente não teríamos mais espaço disponível. Por isso, veremos como criar nossas próprias funções e quais as principais vantagens ao usá-las em nossos programas.

**Introdução**

Após criar soluções para diversos problemas, é comum que certas sequências de código se tornem frequentes, às vezes repetidas no mesmo programa. Uma forma de reduzir a duplicação de código é criar funções com os códigos mais utilizados.

A sintaxe para definição de uma função em Python é bastante simples, mas para usarmos as funções corretamente precisamos entender:

* Quando e por que devemos criar uma função;
* Como o funciona a passagem de valores para uma função;
* Como a função pode retornar um valor de resposta;
* O que acontece com as variáveis que criamos dentro de uma função.

As principais vantagens do uso de funções são:

* Abstração e reusabilidade, evitando duplicidade de código;
* Modularização, permitindo que um problema inicial seja dividido em problemas menores e mais fáceis de serem resolvidos;
* Separação de escopo, criando área de trabalho local para a função e evitando conflito entre variáveis internas da função e demais variáveis do programa.

Estas vantagens levam a um código com maior legibilidade, que será mais fácil de manter, atualizar e corrigir. A abstração está relacionada a separação entre “o que” precisa ser feito e “como” será feito. Quando utilizamos apenas funções integradas e importadas, a preocupação é apenas com “o que” a função faz, ao criar nossas próprias funções o “como” também é nossa responsabilidade. Porém, uma vez que nossas funções estão definidas e devidamente testadas, aumentando a confiança que estão corretas, voltamos a lidar apenas com o “o que”, podendo utilizá-las de modo semelhante às funções integradas e importadas, usufruindo das mesmas vantagens.

**Funções importadas**

Vimos em Python as funções integradas, que podem ser usadas em qualquer trecho de código, pois são reconhecidas automaticamente pelo interpretador e, consequentemente, estão sempre disponíveis. Agora, aprenderemos a criar nossas próprias funções, mas antes disso, veremos como usar funções desenvolvidas por outros programadores e que estão disponíveis para uso em pacotes ou módulos extras.

O Python possui um conjunto de módulos extras que chamamos de biblioteca padrão. Os módulos dessa biblioteca são instalados junto com o interpretador, mas não são carregados automaticamente ao executarmos o Python. Quando precisamos usar um módulo da biblioteca padrão, primeiro temos que importá-lo para o nosso código-fonte, indicando ao interpretador para carregá-lo e disponibilizá-lo para uso. Veja mais detalhes em: <https://docs.python.org/pt-br/3/library/>

A importação é feita com o comando import.

>>> import math

>>> math.pi

3.141592653589793

>>> math.log(10)

2.302585092994046

Primeiro, importamos o módulo math. Isso criará uma variável de mesmo nome que nos dará acesso ao módulo (descubra o tipo do dado referenciado pela variável usando a função integrada type). Denominamos os conteúdos de um módulo como propriedades. Para acessar as propriedades, usamos um ponto após o nome da variável, seguido pelo nome da propriedade que queremos acessar. A descrição completa do módulo de matemática está disponível na documentação do Python (PSF, 2021).

Lembrando que se a propriedade for uma função, será necessário usar um par de parênteses para executá-la, semelhante a qualquer função que usamos até agora, pois caso contrário, acessaremos somente a referência para o objeto da função na memória.

Ao criar códigos-fonte, recomenda-se que as importações estejam no começo do arquivos, sendo necessário importar cada módulo uma única vez em um mesmo arquivo. Inclusive, instruções que tentem importar um módulo já importado não terão efeito.

**Funções definidas pelo programador**

Para criarmos novas funções é preciso defini-las. Em Python, para definir uma função, utiliza-se o comando def.

def <nome da função>(<parâmetros>):

<bloco de código>

Crie nomes de funções que sejam claros indicativos do que ela faz, pois isso facilitará a legibilidade e uso. As regras para nomes de funções são idênticas àquelas para variáveis, até porque em Python o nome da função também é uma variável, só que uma variável que referencia um código de função que está na memória.

Portanto evite criar funções com o mesmo nome de outras variáveis, pois isso poderá gerar problemas em que a variável que referencia a função tem seu conteúdo sobrescrito por outro valor, ou uma variável qualquer tem seu conteúdo perdido por passar a referenciar o código de uma função.

Após o nome da função, deve-se colocar um par de parênteses e, entre eles, os parâmetros, se existirem. Os parâmetros são variáveis internas da função que recebem os argumentos passados quando a função é chamada. Note que o par de parênteses é obrigatório, mas, dependendo da função, pode não haver parâmetros, se houver mais de um, devem ser separados por vírgulas.

Na linha seguinte aos dois pontos, que sinalizam o fim da assinatura da função, está o bloco de código, indentado para indicar que pertence à função, assim como é feito nas estruturas de seleção. Nesse bloco de código, pode-se inserir qualquer instrução válida, inclusive chamadas a outras funções integradas, importadas ou definidas pelo próprio programador. É possível definir funções dentro de outra, mesmo não sendo uma prática frequente. No entanto, chamar funções dentro de outra é algo bastante utilizado!

def soma(n1, n2):

s = n1 + n2

return s

Note que soma é o nome da função e, como mencionamos, em Python o nome da função é uma variável que faz referência ao seu código. A referência é feita por meio do endereço da função, ou seja, a variável guarda o endereço de memória onde está o código da função para ser executado. Isso permite, por exemplo, que outra variável também receba o endereço e possamos chamar a função por meio dessa nova variável.

>>> imprime = print

>>> imprime('Programando em português!')

Programando em português!

Retomando à criação da função, repare que existem dois momentos distintos ao lidarmos com funções definidas pelo próprio programador: (I) a definição da função e; (II) a invocação da função, também conhecido como “chamar a função”.

O comando def só indica que a função deve ser definida e, após ser executado, disponibiliza a função na memória e a associa ao nome escolhido, porém não a chama automaticamente. Obviamente, toda função deve estar definida antes de ser chamada.

Após a função estar definida, para executá-la basta chamá-la de modo idêntico às funções integradas e importadas, usando parênteses após seu nome e com os argumentos entre eles. Veja exemplos de chamadas à soma.

>>> soma(3, 5)

8

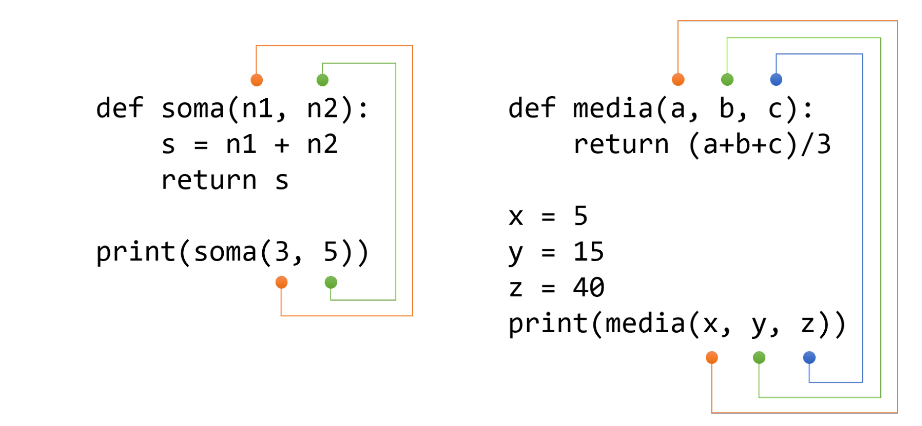
>>> soma(7, -9)

-2

Sabemos que ao chamar uma função podemos passar valores como argumentos se, é claro, a função for definida com parâmetros que receberão esses argumentos. Assim, podemos entender os parâmetros com variáveis de entrada da função, e os argumentos como os valores que serão atribuídos a essas variáveis.

Por padrão, os argumentos são atribuídos aos parâmetros seguindo a ordem em que são passados na chamada da função. Logo, o primeiro argumento será atribuído ao primeiro parâmetro, o segundo argumento será atribuído ao segundo parâmetro e assim por diante. No exemplo import math duas funções definidas pelo programador e suas respectivas chamadas, note a relação entre os argumentos e os parâmetros.

No exemplo “def soma” os parâmetros da função soma são n1 e n2, e ela foi chamada duas vezes no exemplo acima, uma com os argumentos 3 e 5, e outra com os argumentos 7 e -9, retornando, em cada chamada, o resultado da soma de seus parâmetros. O retorno de um valor é feito com o comando return seguido pelo valor retornado, que pode ser originado de uma variável ou expressão mais complexa.



def soma(n1, n2): #1

print('Início do bloco da função') #2

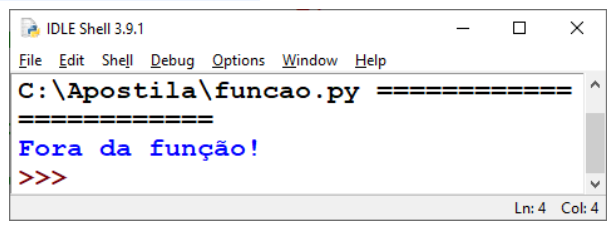
s = n1 + n2 #3

return s #4

print('Fim do bloco da função') #5

print('Fora da função!') #6

Após a execução da Codificação acima, o resultado é ilustrado abaixo. A função soma foi definida, mas como ela não foi chamada, seu bloco de código ainda não foi executado. Agora que está carregada na memória, vamos refazer as chamadas à soma e observar o resultado.



>>> soma(3, 5)

Início do bloco da função

8

>>> soma(7, -9)

Início do bloco da função

-2

Notamos que a cada chamada à soma o primeiro print é executado, mas não o segundo, por quê? A razão é que o segundo print está após a instrução return, que sempre é executada antes dele. Quando uma função executa um comando return, ela retorna o valor da expressão à direita de return, se existir, e encerra sua execução, devolvendo o controle do fluxo para à instrução que a chamou. Logo, qualquer instrução no bloco da função posterior à execução de um return é inalcançável.

É possível que uma função tenha mais de um return, o que pode ser útil caso estejam em blocos de código diferentes como, por exemplo, em uma estrutura de seleção, que a depender de sua condição executará um ou outro return, jamais os dois.

def par(n):

if n % 2 == 0:

return True

else:

return False

print(f'{4} é par? {par(4)}')

print(f'{7} é par? {par(7)}')

**Retorno de valor vs. exibição de valor**

Na Codificação Def Soma, poderíamos ficar tentados a afirmar que a função soma exibe o resultado da soma, mas ao olharmos para o seu bloco de código, vemos que ela retorna o valor da variável s ao invés de exibi-lo. Nem mesmo há print na função!

O retorno da função soma só foi exibido devido ao comportamento interativo da Shell, que exibe o resultado de qualquer expressão, o que inclui os valores retornados por funções. Para visualizar a diferença, faremos as mesmas chamadas, porém agora no editor do IDLE.

def soma(n1, n2): #1

print('Início do bloco da função') #2

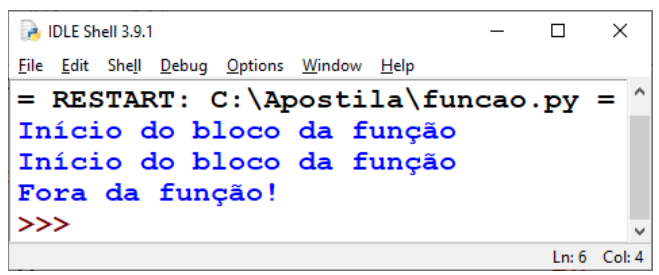
s = n1 + n2 #3

return s #4

soma(3, 5) #5

soma(7, -9) #6

print('Fora da função!') #7

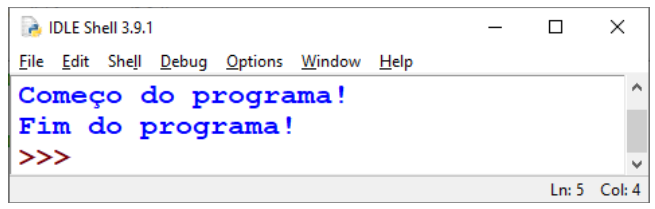


A função retornou, normalmente, um valor a cada chamada, porém esse valor não foi aproveitado para outra operação como, por exemplo, uma exibição. O mesmo ocorre ao criar uma instrução que seja unicamente uma expressão, como na segunda linha da Codificação abaixo e o resultado.

print('Começo do programa!')

2 \* 3 + 4

print('Fim do programa!')



Portanto, temos basicamente duas opções para que o valor final produzido por uma função seja exibido:

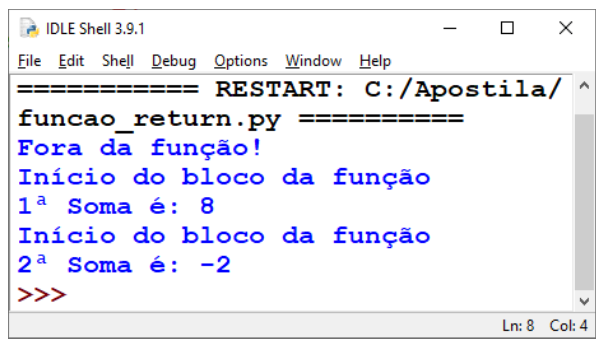
1. Exibir o valor retornado pela função na instrução em que ela foi chamada, como na “def par”, ou atribuir o valor à uma variável e depois exibi-la;
2. Alterar o código da função para que o valor seja exibido e não retornado.

print('Fora da função!') #5

resultado1 = soma(3, 5) #6

print('1ª Soma é:', resultado1) #7

print('2ª Soma é:', soma(7, -9)) #8

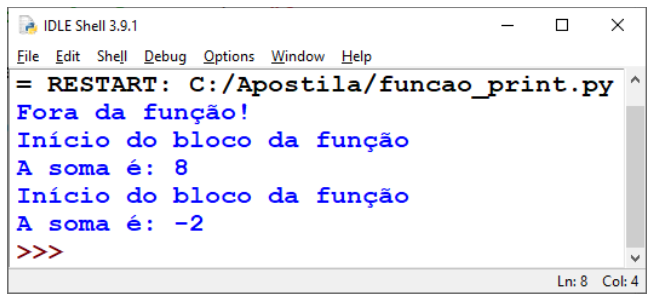


def soma(n1, n2): #1

print('Início do bloco da função') #2

s = n1 + n2 #3

print('A soma é:', s) #4



>>> teste = soma(2, 3)

A soma é: 5

Note que a função foi executada corretamente, inclusive sua instrução print cumpriu o papel esperado, exibindo a soma. Porém, há algo estranho na codificação acima. Qual o valor retornado por uma função que nem mesmo tem um return explícito? Descubra ao tentar exibir o valor da variável teste, como na Codificação abaixo. Repare que não é possível visualizar o conteúdo desta variável sem o uso de print.

>>> print(teste)

None

Exatamente! Em Python, uma função que não executa um comando return, ou que o executa sem um valor à direita para ser retornado, devolve como resposta um valor especial em Python, o None. Em Python, esse valor é um indicativo do “nada”. Essa é uma característica de Python. Em outras linguagens, tentar atribuir a uma variável o valor de retorno de uma função que não retorna explicitamente algo pode gerar erro.

A pergunta final é: “quando usar return e quando usar print?”. A resposta é: “depende”. Depende porque é necessário saber qual o objetivo da função que será criada.

Na maioria das vezes, utilizaremos return, pois em geral construímos funções que respondem uma pergunta: “qual a soma de dois números?”, “qual a raiz quadrada de um número?”, “quantos caracteres uma string possui?”, “este número é par?”. Fazemos uma pergunta e esperamos um valor como resposta, que pode ser um número, um valor booleano, uma string ou qualquer outro tipo de dado do Python.

Já o print será usado quando dermos uma ordem à função para exibir algo na tela, sem esperar uma resposta, como: “exiba o menu de opções!”, “mostre uma mensagem de saudação!”, e assim por diante. Nestes casos não há necessidade de retorno de valor. Esse tipo de função faz parte de uma classe de funções conhecidas como funções nulas, que são funções que executam alguma ação (como exibir algo na tela ou outros efeitos) e, em Python, sempre retornam o valor None.

Portanto, em Python, toda função retorna um valor: (a) explicitamente, quando está à direita de um comando return ou; (b) implicitamente, quando não há return ou quando este não está acompanhado de um valor a ser retornado, neste caso retorna-se None. Note que há funções que não precisam retornar um valor, o algoritmo não exige, mas por questão de projeto da linguagem Python retornarão automaticamente None.

**Escopo de variáveis**

De modo simplificado, podemos pensar no escopo como a região do programa na qual a ligação de um identificador com um determinado valor na memória seja válida. Por exemplo, após executar a instrução n = 2, o escopo de n é toda parte do código na qual podemos acessar exatamente essa variável usando o identificador n. Se tentarmos acessar n fora do seu escopo, teremos acesso a outro espaço de memória ou um erro de identificador não definido.

Em Python, variáveis criadas na raíz do código, isto é, fora de funções, terão escopo global, e serão acessíveis em todo o código-fonte onde foram criadas, a partir de sua instrução de criação. Já variáveis criadas no interior de funções terão escopo local e, portanto, acessíveis apenas dentro da função em que foram criadas.

def diga\_ola():

print(f'Olá {nome}!') # acessa a variável global

nome = 'Megan' # variável global

diga\_ola()

A variável nome foi criada fora de qualquer função, então é global, isso significa que quando o Python não encontrar o identificador nome no escopo local da função, irá procurá-lo no escopo global. Caso o nome também não esteja no escopo global, será disparado um erro de “nome não definido”.

Variáveis locais têm prioridade sobre globais, caso existam duas variáveis como o mesmo identificador a local será a utilizada por padrão.

def diga\_ola(nome): # parâmetros são variáveis locais

print(f'Olá {nome}!') # acessa a variável local

nome = 'Megan' # variável global

diga\_ola('Maria')

print(f'Tchau {nome}!') # acessa a variável global



Qualquer atribuição a variável dentro de uma função gera uma variável local, porém se a intenção é alterar variáveis globais, deve-se indicar com o comando global sucedido pelo nome das variáveis entre vírgulas.

def diga\_ola():

global nome, titulo # são identificadores globais

print(f'Olá {titulo} {nome}!') # acessa a variável global

nome = 'Megan' # modifica a variável global

titulo = 'Sra.' # modifica a variável global

titulo = 'Dra.' # variável global

nome = 'Maria' # variável global

diga\_ola()

print(f'Tchau {titulo} {nome}!')

O uso de variáveis globais em funções é desencorajado, pois viola o encapsulamento das funções, dificulta a leitura do código e correções. Note que as funções que usam variáveis globais dependem de recursos que podem ser alterados por outros trechos de código, o que pode gerar caos em programas mais complexos.

Por isso, caso precise informar valores externos às funções, use os parâmetros. Desta forma a função terá os dados necessários para executar seu algoritmo sem violar o encapsulamento.

**Texto de documentação de funções**

Ao definirmos novas funções, é comum acoplarmos comentários para explicar o funcionamento para quem as usará, que podem ser outros programadores ou nós mesmos. Se feito conforme um padrão, muitos editores de código poderão identificar essa documentação e mostrá-la de modo conveniente, para isso existem as docstrings. Vamos criar uma documentação para nossa função soma.

def soma(n1, n2):

"""Retorna a soma de dois números.

Parâmetros

----------

n1, n2: int ou float

Números a serem somados.

Retorno

-------

int ou float

Resultado da soma de n1 com n2.

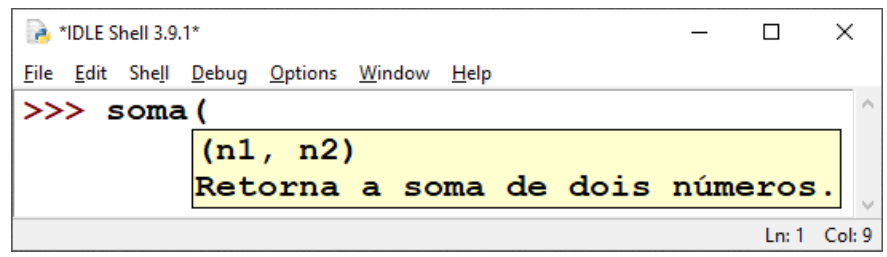
"""

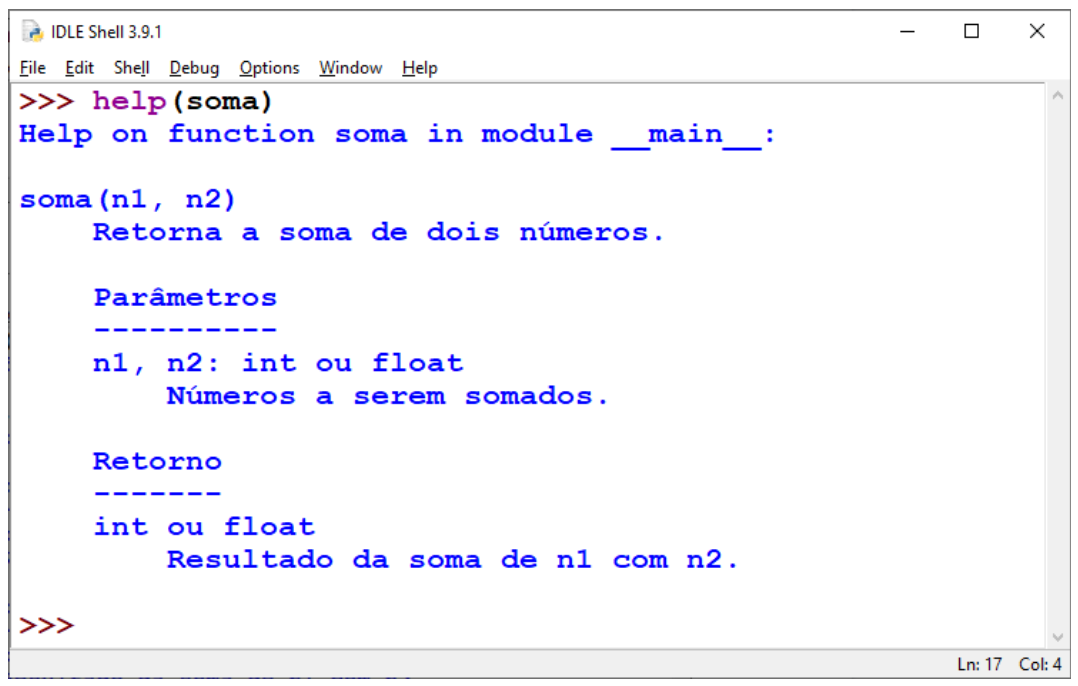
s = n1 + n2

return s

A primeira linha é um resumo sobre o que a função faz, em seguida podemos adicionar outras informações sobre a função, como seus parâmetros e valor de retorno. Aqui utilizamos o guia de docstrings criado pelo NUMPY (2021). O guia recomenda que tanto o código quanto as strings de documentação estejam em inglês, mas como toda recomendação de guia de estilo, o mais importante é manter a coerência interna em um projeto, por isso fizemos o exemplo em português.

Após inserir uma docstring na função, podemos visualizar seu conteúdo de algumas formas diferentes, dentre as quais abordaremos duas: (I) digitando o nome da função seguida da abertura de parênteses, tanto na Shell quanto no editor, aguardando alguns segundos para a exibição de um balão com a assinatura da função, junto com a primeira linha da docstring, como podemos ver na Figura abaixo; (II) usando a função help, passando como argumento o nome da função que contém a docstring, desta forma a documentação completa será exibida, conforme a Figura abaixo.





**Organização do código-fonte**

Aprendemos a importar módulos da biblioteca padrão e a criar nossas próprias funções, agora veremos algumas recomendações de organização para melhorar a legibilidade, rastreamento de erros e manutenção de nossos códigos-fonte:

1. As importações de bibliotecas devem ser feitas no início do código-fonte;
2. Após as importações, se definem as funções;
3. Por último, escreve-se o código principal que chamará as funções e resolverá o problema. É comum denominar o código principal de programa principal.

**# Importações**

import math

**# Definição de funções**

def dobro(x):

return 2 \* x

def saudacao(nome):

print(f'Bem-vindo {nome}!')

**# Código principal**

nome = input('Qual o seu nome? ')

saudacao(nome)

dobro\_pi = dobro(math.pi)

print(f'O dobro de {math.pi:.4f} é {dobro\_pi:.4f}')

[Vídeo - Funções Guanabara](https://youtu.be/ezfr9d7wd_k?si=7FtWMEiOTs4OwPZB)

**Parte 7 -** [**Estrutura de repetição while**](https://sites.google.com/faculdadeimpacta.com.br/lp-p7?pli=1&authuser=1) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=5NsJePjbKzY)

**Estrutura de repetição indefinida (while)**

**Resumo**

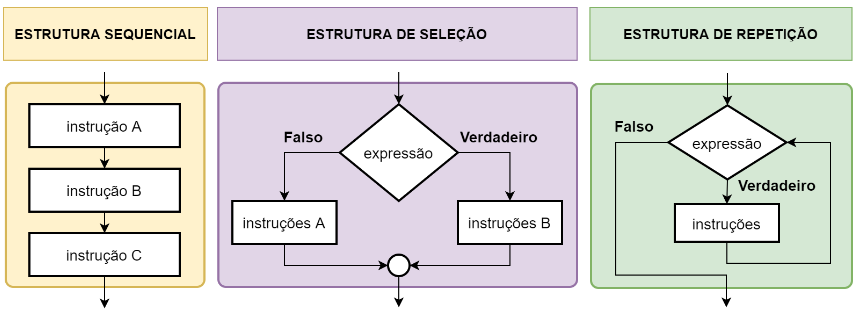
Nesta aula os objetivos são: (I) entender a necessidade da estrutura de repetição para controle do fluxo de execução; (II) compreender a estrutura de repetição com quantidade de repetições indefinida: laço while; (III) usar variáveis contadoras, acumuladoras e de sinalização booleana para controlar a execução do while; (IV) conceituar “laço infinito” e suas consequências; (V) utilizar estruturas de repetição combinadas com estruturas de seleção.

**Motivação**

Durante nosso curso aprendemos diversos recursos de programação, porém há algo muito importante realizado pelos computadores que precisamos abordar: a repetição de instruções. É natural que programas executem uma sequência de instruções repetidas vezes, aliás essa é uma das principais características em que as máquinas superam os seres humanos. Nesta aula compreenderemos a importância de utilizar uma estrutura de repetição e como escrevê-la na linguagem de programação Python.

**Introdução**

Já aprendemos duas formas para controlar o fluxo de execução de um programa: (I) estrutura de controle sequencial, em que o fluxo de execução é dado pela ordem em que as instruções são escritas e; (II) estrutura de controle de seleção ou condicional, onde o fluxo de execução sofre desvios de acordo com o resultado da avaliação de uma condição. Agora aprenderemos a controlar o fluxo de execução de modo que uma sequência de instruções possa ser repetida diversas vezes sem que seja necessário escrevê-la diversas vezes, para tanto usaremos as estruturas de repetição. Para recordar, as três estruturas básicas de controle de fluxo estão ilustradas abaixo.



As estruturas de repetição também são conhecidas como estruturas de iteração, malhas de repetição, laços e loops.

Os loops são aplicados quando criamos algoritmos que demandam a repetição de uma sequência de instruções várias vezes. Vamos iniciar com a análise de um problema simples que desperta a reflexão sobre a necessidade das estruturas de repetição: criar um programa que exiba os cinco primeiros números naturais positivos. O conhecimento que temos permite criação da solução que está no código abaixo.

print(1)

print(2)

print(3)

print(4)

print(5)

A solução para este problema é simples, no entanto, imagine que ao invés de exibir os cinco primeiros números, o problema exigisse os cinquenta primeiros. Por mais que a lógica aplicada ao novo cenário seja equivalente, a repetição de tantas instruções tornaria o trabalho do programador lento e tedioso, além de aumentar a chance de inserção de instruções incorretas, consequência do aumento do código-fonte.

Agora, modificaremos um pouco o enunciado do problema original. Suponha que devemos exibir os cinco números naturais posteriores ao natural dado pelo usuário como entrada. A Codificação abaixo é uma solução válida para esse enunciado.

x = int(input('Valor: '))

print(x + 1)

print(x + 2)

print(x + 3)

print(x + 4)

print(x + 5)

A solução pode ser melhorada, tornando todas as instruções de exibição iguais e potencialmente reduzindo erros de digitação do programador, permitindo replicação das instruções apenas com o “copiar e colar”. Veja a alteração na Codificação abaixo, em que incrementamos o valor da variável x antes de cada exibição.

x = int(input('Valor: '))

x += 1

print(x)

x += 1

print(x)

x += 1

print(x)

x += 1

print(x)

x += 1

print(x)

Você deve ter observado que tanto a solução para o primeiro problema quanto para o segundo são possíveis de serem implementadas com o conhecimento que temos até o momento, porém são trabalhosas e, eventualmente, tornam-se inviáveis quando a quantidade de repetições é muito grande. Porém, existem problemas que simplesmente são impossíveis de serem resolvidos apenas copiando e colando instruções.

Um exemplo no qual existe esse impedimento é a terceira variação do problema para exibição de números naturais: suponha que o programa deverá exibir uma sequência de números naturais delimitada pelos valores início e fim, que são dois naturais dados pelo usuário como entrada. Note que o programa deverá exibir a sequência completa, incluindo os extremos início e fim. Por exemplo, caso o usuário insira o valor 5 como início e 12 como fim, a saída será 5 6 7 8 9 10 11 12.

Usando o que sabemos até agora, qual a solução para essa terceira variação do problema? Impossível construí-la! Não há como prever a quantidade de valores exibidos, afinal isso dependerá dos valores de entrada dados pelo usuário, algo que só será descoberto quando o programa estiver em execução. Então, precisamos ampliar nossos recursos de programação para resolver esse problema, precisamos dos laços!

**Tipos de estruturas de repetição**

Essencialmente as estruturas de repetição podem ser classificadas em dois tipos:

1. **Estruturas de repetição com quantidade de repetições indefinida:** não é possível determinar quantas vezes as instruções do loop serão executadas antes dele ser iniciado;
2. **Estruturas de repetição com quantidade de repetições definida:** é possível determinar quantas vezes as instruções do loop serão executadas antes mesmo dele ser iniciado.

As linguagens de programação podem ter mais de um loop de cada tipo, permitindo ao programador decidir qual o mais adequado de acordo com o contexto, porém a razão de existência de todos os laços é a mesma: executar uma sequência de instruções repetidas vezes. Em Python temos um único laço de cada tipo: while e for.

**Estrutura de repetição while**

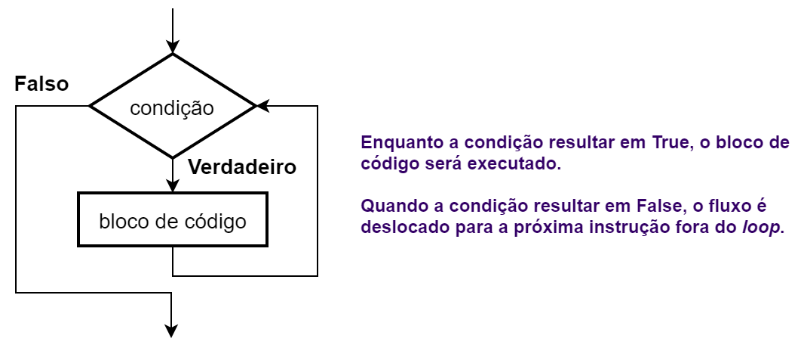
Inicialmente abordaremos somente o representante da estrutura de repetição com quantidade de repetições indefinida, que em Python é representada pelo comando while. A estrutura do comando while é similar à do if.

while <condição>:

<bloco de código>

As mesmas regras que aprendemos para as estruturas de seleção se aplicam ao loop while, portanto toda instrução que estiver indentada no <bloco de código> pertence ao loop e só será executada caso <condição> resulte em True, caso resulte em False o laço é encerrado e a instrução imediatamente seguinte ao seu bloco será a próxima a ser executada. Aparentemente, um fluxo de execução semelhante ao do if.

Porém, o while possui uma notável diferença em relação ao if. No while, após a execução da última instrução de seu bloco de código, o fluxo de execução é deslocado novamente para sua condição, que será avaliada mais uma vez, repetindo todo o procedimento explicado no parágrafo anterior. Podemos visualizar o funcionamento da estrutura de repetição while no fluxograma abaixo.



Para exemplificar o funcionamento do while, reescreveremos com um laço a solução para a primeira variação do problema de números naturais (Codificação anterior). Para ampliar o entendimento, acrescentaremos a exibição da frase “Tchau!” como uma instrução imediatamente posterior ao fim do loop, conforme Codificação abaixo.

x = 1

while x <= 5:

print(x)

x += 1

print('Tchau!') # instrução imediatamente posterior ao laço.

O bloco de código do while será executado exatamente cinco vezes, pois após a quinta execução de x += 1 a variável x estará com o valor 6, fazendo x <= 5 resultar em False. No entanto, é importante observar que a condição do laço será sempre avaliada uma vez mais, neste caso exatamente seis vezes, pois apenas na sexta avaliação que a condição resultará em False e, consequentemente, encerrará o loop. A instrução que exibe “Tchau!” é executada só uma vez, pois está fora do while.

A segunda variação do problema de números naturais, cuja primeira solução consta na Codificação anterior, seria facilmente solucionada com while, conforme Codificação abaixo:

x = int(input('Valor: '))

qtd\_exibidos = 0

while qtd\_exibidos < 5:

x += 1

print(x)

qtd\_exibidos += 1

Na Codificação 7.6 a variável qtd\_exibidos é usada para verificar quantos naturais foram exibidos. Ela começa em zero, pois antes do laço nenhum número foi exibido, e é incrementada a cada execução do bloco do laço. Veja que para exibir os cinquenta naturais posteriores ao da entrada, basta alterar a condição para qtd\_exibidos < 50. Se o enunciado solicitasse os primeiros cinquenta mil números posteriores, bastaria alterar uma única instrução. Você entendeu o poder dos laços!

Por fim, você deve se recordar da terceira variação do problema de números naturais, em que dois valores (início e fim) são dados pelo usuário como entrada e que devemos exibir todos os valores do intervalo de naturais [início..fim]. Agora, com o uso de um laço, é possível resolvê-lo facilmente, como consta na Codificação abaixo.

inicio = int(input('Início: '))

fim = int(input('Fim: '))

x = inicio

while x <= fim:

print(x)

x += 1

Observe que agora colocamos o incremento da variável x no final do bloco do laço, isso é comum, pois ela agora é a variável que controlará a execução do laço e uma forma de interpretar este laço é: (a) primeiro fazemos as operações necessárias com o x atual (neste caso apenas a exibição); (b) depois de terminar, passamos para o próximo valor de x (neste caso um incremento de 1); (c) voltamos à condição para verificar se x ainda atende à condição; (d) caso atenda (condição resulta True) executamos mais uma vez o laço, repetindo o processo; (e) caso não atenda (condição resulta False), encerramos o laço e passamos para a próxima instrução fora do bloco do while.

**Variável contadora**

Para construir laços de repetição muitas vezes utilizamos uma variável contadora, que é útil para contar quantas vezes o bloco de instruções do laço foi executado e/ou controlar quantas vezes ele será executado. A variável contadora é incrementada/decrementada por um valor constante, geralmente de um em um.

A Codificação abaixo é um exemplo de programa que usa uma variável contadora apenas para contabilizar a quantidade de vezes que o bloco de instruções do loop foi executado. Note que a variável contador não influencia o número de repetições do while, isto é, ela não é usada como variável de controle deste laço.

executa = input('Executar o bloco do laço: ')

contador = 0

while executa == 'sim':

contador += 1

executa = input('Executar o bloco do laço de novo: ')

print(f'O bloco do laço foi executado {contador} vezes')

A Codificação abaixo é um exemplo de programa que usa uma variável contadora para controlar a quantidade de vezes que o bloco de instruções do laço será executado. Note que a variável contador influencia o número de repetições do while, isto é, agora ela é usada também como variável de controle deste laço.

contador = 10

while contador > 0:

print(contador)

contador -= 1

print('Fogo!')

**Variável acumuladora**

Uma variável acumuladora é utilizada para acumular valores que, em geral, não são constantes, ou seja, seu incremento ou decremento pode ser variável.

Vejamos um exemplo de problema que precisa de uma variável acumuladora: “crie um programa que receba como entrada os preços de itens comprados em um supermercado por um cliente, no final o programa deverá exibir o total da compra. Para informar que não há mais itens a serem comprados, o cliente digitará o valor -1”. A Codificação abaixo é uma solução válida para esse enunciado.

total = 0 # variável acumuladora

preco = float(input('Preço do item: '))

while preco != -1:

total += preco

preco = float(input('Preço do item: '))

print(f'Total da compra: R$ {total:.2f}')

Vejamos outro exemplo de problema com duas variáveis acumuladoras, porém com uma delas sendo decrementada: “crie um programa que receba como entrada o crédito de um cliente e depois o preço de itens comprados por esse cliente, o programa deverá parar de solicitar novos preços quando o crédito disponível for insuficiente para pagar por um deles. Ao final, exiba o total da compra e o crédito restante”. A Codificação below é uma solução válida para esse enunciado.

credito = float(input('Seu crédito: ')) # variável acumuladora

total = 0 # variável acumuladora

preco = float(input('Preço do item: '))

while credito >= preco:

total += preco

credito -= preco

preco = float(input('Preço do item: '))

print(f'Total da compra: R$ {total:.2f}')

print(f'Crédito restante: R$ {credito:.2f}')

Observe que a variável credito, além de acumuladora, também é usada como variável de controle do laço. Neste exemplo, temos duas variáveis usadas para controlar a execução do loop.

**Variável flag booleana**

As vezes criamos laços em que a condição é dada por uma variável com valor booleano. Nestes casos dizemos que a variável usada para controlar o laço é uma flag booleana, pois ela sinaliza se o laço deve ou não ser encerrado. Também é possível criar uma condição com mais de uma flag booleana. Veja um exemplo na Codificação abaixo.

total = 0

quero\_comprar = True # será usada como flag booleana no loop.

while quero\_comprar:

preco = float(input('Preço: '))

total += preco

opcao = input('Continuar comprando (s/n)? ')

if opcao != 's':

quero\_comprar = False

print(f'Total da compra: R$ {total:.2f}')

Note que a clareza do nome dado à variável definida como flag booleana é muito importante, pois influenciará na forma como o programador entenderá o fluxo de execução. No exemplo da Codificação *above*, podemos entender a leitura da primeira linha do loop como “enquanto eu quiser comprar, execute o bloco de instruções”. Por isso, é evidente que quando a variável quero\_comprar receber False, poderemos entender como “não quero comprar”, indicando que o loop deve ser encerrado.

**Laço infinito**

Um laço infinito é uma estrutura de repetição que nunca é encerrada, ou que é encerrada apenas por um eventual erro no programa ou por um comando externo, como um pedido de finalização feito por meio do gerenciador de tarefas do sistema operacional. Geralmente laços infinitos são gerados por erros de lógica, como uma instrução esquecida pelo programador que faz com que a condição do laço jamais resulte em False. Portanto, se quisermos gerar um while infinito devemos garantir que sua condição sempre resulte em True.

Veja na Codificação below, dois exemplos de laços infinitos, em que o fluxo de execução, uma vez no laço, jamais atingirá a instrução print('acabou!').

n = 0

while n <= 10:

print(n)

print('Acabou!')

n = 0

while n >= 0:

print(n)

n += 2

print('Acabou!')

**Combinação de estruturas de controle de fluxo**

Como visto na Codificação, é bastante comum combinarmos estruturas de repetição com estruturas de seleção. Em algoritmos mais complexos a combinação de estruturas de controle de fluxo ocorre frequentemente.

Um exemplo de combinação de estruturas de controle de seleção e repetição está no seguinte problema: “Crie uma função que exiba apenas as 21 consoantes minúsculas do alfabeto latino.”. A Codificação below é uma solução válida para esse enunciado.

def consoantes():

codigo\_unicode = ord('a')

while codigo\_unicode <= ord('z'):

letra = chr(codigo\_unicode)

if (letra != 'a' and letra != 'e' and letra != 'i' and

letra != 'o' and letra != 'u'):

print(letra)

codigo\_unicode += 1

Outro exemplo de combinação de estruturas de controle de seleção e repetição ocorre na solução para o seguinte enunciado: “Crie uma função que receba como argumento um número natural n e devolva um valor booleano indicando se n é primo”. Um número natural é considerado primo se possui exatamente dois divisores naturais, o número 1 e o próprio n. A Codificação below é uma solução válida para esse problema.

def primo(n):

qtd\_divisores = 0

divisor = 1

while divisor <= n:

if n % divisor == 0:

qtd\_divisores += 1

divisor += 1

if qtd\_divisores == 2:

return True

else:

return False

**Curso Python #014​ - Estrutura de repetição while** [Curso Python #014 - Estrutura de repetição while](https://youtu.be/LH6OIn2lBaI)

**Parte 8 -** [**Estrutura de repetição Aninhadas**](https://sites.google.com/faculdadeimpacta.com.br/lp-p8?pli=1&authuser=1) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=-EHtkRDvaqw)

**Resumo**

Nesta aula os objetivos são: (I) aprender os comandos de encerramento e interrupção de laços; (II) simular a estrutura de repetição repeat…until usando while; (III) utilizar laços de repetição para validação de entradas; (IV) entender a necessidade das estruturas de repetição aninhadas; (V) resolver problemas com repetições aninhadas.

**Motivação**

Há situações em que um laço de repetição deve ser interrompido caso algo específico ocorra e Python possui comandos que podem nos ajudar nisso. Usando os comandos de interrupção de laços também poderemos simular um tipo de estrutura de repetição não embutida na linguagem Python, porém útil para a resolução elegante de alguns tipos de problemas! Com a simulação dessa nova estrutura de repetição poderemos realizar validações de dados de entrada, algo muito importante quando lidamos com interação com o usuário. Por fim, você já imaginou repetir uma repetição? Acredite, esse tipo de situação é muito comum na programação e vamos estudá-la!

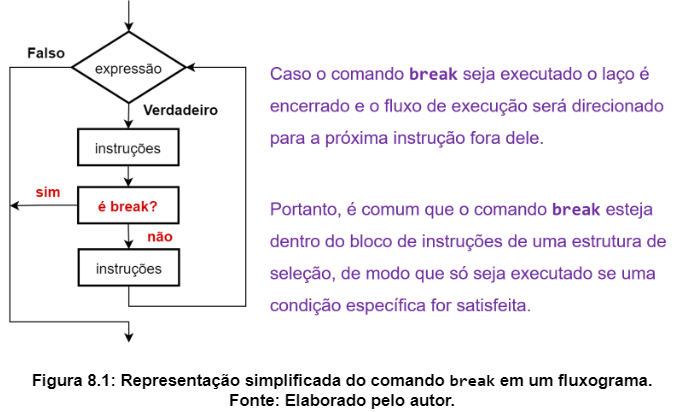
**Introdução**

Até agora aprendemos três formas básicas para controlar o fluxo de execução de um programa: (I) estrutura de controle sequencial; (II) estrutura de controle de seleção ou condicional e; (III) estrutura de controle de repetição.

Nesta aula nos aprofundaremos nas estruturas de repetição. Veremos como encerrar um loop por meio do comando break e como interromper apenas uma rodada com o comando continue. Com esses novos recursos, conseguimos facilmente simular uma estrutura de repetição que em outras linguagens de programação necessitam de comandos específicos, a repeat…until. O conceito desta estrutura pode ser aplicado, por exemplo, para a validação de dados de entrada, isto é, para verificar se os dados fornecidos como entrada pelo usuário são válidos de acordo com o requisitado pelo programa. Por fim, usaremos estruturas de repetição aninhadas, que são laços dentro do bloco de instruções de outros laços, algo recorrente em soluções para problemas mais complexos, em que um laço precisa ser repetido diversas vezes.

**Comando break**

A execução de um comando break encerra o loop mais interno em que está contido. O comando break geralmente está condicionado à um if, de modo que só seja executado quando algo específico ocorrer. A Figura 8.1 ilustra simplificadamente o deslocamento do fluxo de execução proporcionado por esse comando.



Um problema simples para observarmos o funcionamento do comando break é este: “crie um programa que receba como entrada o crédito de um cliente e depois o preço de itens comprados por esse cliente, o programa deverá parar de solicitar novos preços quando o crédito disponível for insuficiente para pagar um dos itens. Ao final, exiba o crédito restante”. A Codificação 8.1 é uma solução válida para esse enunciado.

credito = float(input('Seu crédito: R$ '))

while credito > 0:

item = float(input('Preço do item: R$ '))

if item > credito:

print('Compra negada! Ultrapassa seu crédito.')

break

credito -= item

print(f'Crédito restante: R$ {credito:.2f}')

Outro exemplo de problema que pode ser beneficiado com o uso de break é o seguinte: “Crie um programa que receba como entrada um número real x, um caractere op simbolizando um operador aritmético (+, -, \* ou /) e outro real y. O programa exibirá o resultado da expressão x op y. O procedimento deve ser repetido enquanto o usuário desejar, porém deve ser encerrado antes de ocorrer um erro por causa de uma divisão por zero.”. A Codificação 8.2 é uma solução válida para esse enunciado.

calcular = input('Calcular (s/n)? ')

while calcular == 's':

x = float(input('x: '))

op = input('operador: ')

y = float(input('y: '))

if op == '+':

resultado = x + y

elif op == '-':

resultado = x - y

elif op == '\*':

resultado = x \* y

elif op == '/':

if y == 0:

print('Divisão por zero!\n')

break

else:

resultado = x / y

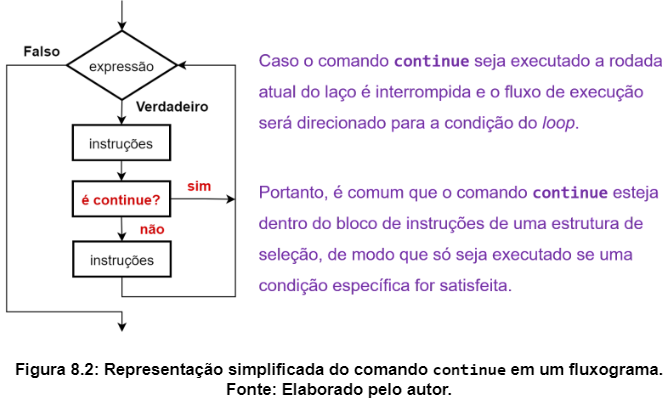
print(f'{x} {op} {y} = {resultado}\n')

calcular = input('Calcular (s/n)? ')

print('Calculadora encerrada')

**Comando continue**

A execução de um comando continue interrompe a rodada atual do loop mais interno em que está contido. O comando continue geralmente está condicionado à um if, de modo que só seja executado quando algo específico ocorrer. A Figura 8.2 ilustra simplificadamente o deslocamento do fluxo de execução gerado por esse comando.



Observaremos o funcionamento do comando continue na Codificação 8.3 que resolve o seguinte problema: “crie um programa que receba como entrada o crédito de um cliente e os preços de itens comprados por ele. Se o preço de um item for superior ao valor do crédito, negue a compra do item, exiba o crédito restante e continue a compra. O programa deve parar de solicitar novos itens quando o crédito for zerado”.

credito = float(input('Seu crédito: R$ '))

while credito > 0:

item = float(input('Preço do item: R$ '))

if item > credito:

print(f'Compra negada! Restam: R$ {credito:.2f}')

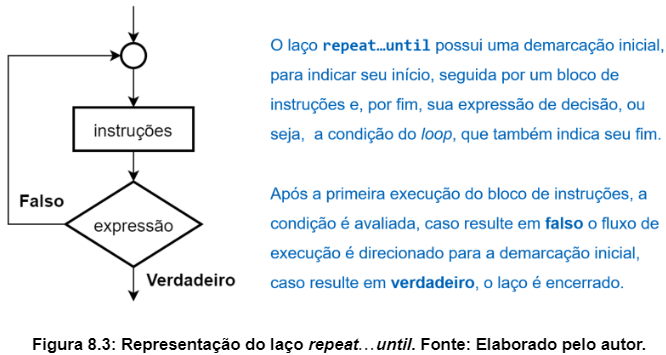
continue

credito -= item

Em geral, principalmente em programas mais simples, break e continue podem ser substituídos por uma adequação da condição do laço e por uma melhor organização de seu bloco de código. Portanto, evite o uso generalizado destes recursos, pois podem tornar o código mais confuso, uma vez que criam desvios do fluxo de execução que podem não ser óbvios à primeira vista, consequentemente dificultando a leitura e manutenção do código. Existem, no entanto, situações pertinentes para uso desses comandos, como a evidenciada em nosso próximo tópico.

**Simulação da estrutura de repetição repeat…until**

Python tem duas estruturas de repetição, while e for (ainda não abordamos). Porém, há problemas que são solucionados de forma mais elegante com uma terceira estrutura de repetição, conhecida como repeat…until ou, em português, repita…até que. Essa estrutura obriga que o bloco de instruções do laço seja executado pelo menos uma vez, pois, diferente do while, a condição para repetição está no fim do laço e não no início. Logo, para que o fluxo de execução atinja a condição deste laço, é necessário que primeiramente passe por seu bloco de código. A Figura 8.3 ilustra esse funcionamento.



O funcionamento do repeat…until justifica o nome da estrutura, indicando que o laço “repetirá as instruções até que sua condição resulte em verdadeiro”. Como já mencionado, o Python não possui esse laço, mas podemos simulá-lo facilmente com a combinação dos comandos while, if e break, conforme a Codificação 8.4.

while True:

<bloco de instruções>

if <condição>: break

###### **Codificação 8.4: Modelo de estrutura simulando o laço repeat**…**until.**

Observe que na Codificação 8.4 definimos a condição do while como True, logo sempre será verdadeira. A princípio, poderíamos imaginar que isso tornaria o loop infinito, porém, no final do bloco de instruções, temos um if com uma condição que se resultar em verdadeiro executará o comando break e, como consequência, encerrará o laço. Veja na Codificação 8.5 um exemplo de solução com essa estrutura para o problema a seguir: crie uma função que receba como argumento um número natural e exiba seus dígitos na ordem do último para o primeiro.

def invertido(n):

while True:

print(n % 10, end='')

n = n // 10

if n == 0: break

###### **Codificação 8.5: Exemplo de aplicação do laço repeat**…**until.**

**Validação de dados de entrada**

Quando os programas solicitam entradas ao usuário, é comum que ocorram inconsistências entre os dados esperados e aquilo que é inserido, erros cometidos pelo usuário propositalmente ou não. Quando é fornecido um dado inesperado, o programa pode falhar, encerrando inesperadamente ou produzindo resultado inconsistente. Para minimizar esse tipo de ocorrência, é útil realizar validações de dados de entrada.

Geralmente, validações de entrada são feitas com loops, solicitando um valor para o usuário repetidamente, até que seja inserido algum válido de acordo com o esperado pelo algoritmo que é estipulado pelo programador. Existem várias abordagens, uma das mais simples é com a estrutura repeat…until. Veja na Codificação 8.6 a solução para o seguinte problema: “crie um programa que receba como entrada um número natural n no intervalo [0..100], e exiba a soma dos n primeiros naturais positivos. A soma deverá ser feita por uma função, que a devolverá como resposta. Note que o programa deverá validar a entrada e só chamará a função quando n for válido”.

def somatorio(n):

soma = 0

natural = 1

while natural <= n:

soma += natural

natural += 1

return soma

while True:

n = int(input('Natural entre 0 e 100: '))

if 0 <= n <= 100: break

print(f'A soma dos {n} primeiros naturais é {somatorio(n)}.')

###### **Codificação 8.6: Programa com validação de dados de entrada usando repeat**…**until.**

Note que na Codificação 8.6 a chamada à função somatorio só ocorrerá caso o fluxo de execução ultrapasse a validação de entrada feita com o laço while (usado para simular a estrutura repeat…until). Portanto, o programa impede a ocorrência de um erro na função somatorio caso seja inserido um valor negativo na entrada, pois se isso ocorrer, será solicitado outro valor de entrada. A Codificação 8.7 exibe uma versão alternativa, em que a estrutura de repetição repeat…until é trocada por um laço while convencional, note que apenas o trecho alterado está escrito.

...

n = int(input('Natural entre 0 e 100: '))

while n < 0 or n > 100:

n = int(input('Número inválido, digite novamente: '))

...

###### **Codificação 8.7: Programa com validação de dados de entrada usando while tradicional.**

Uma vantagem desta abordagem é que podemos personalizar a mensagem exibida quando o usuário digita um valor incorreto, no entanto, temos a desvantagem de precisar inserir duas vezes a instrução que pede o dado ao usuário, algo que pode ser evitado quando a personalização não é necessária.

Na Codificação 8.7, a condição do while é complementar à Codificação 8.6, pois na estrutura repeat..until a estratégia é “repita algo até que a condição seja atendida”, e a condição é o número ser válido. Já na estrutura tradicional do while, a estratégia é “repita enquanto a condição for atendida”, logo o laço só deve ser repetido enquanto o número for inválido, que é condição complementar à de número válido.

Para obter a condição complementar à utilizada na Codificação 8.6, podemos simplesmente negá-la com o operador not, ou simplificá-la a negação, como a seguir:

not (0 <= n <= 100) ⇒ not (0 <= n and n <= 100)

⇒ not(0 <= n) or not(n <= 100) ⇒ 0 > n or n > 100

⇒ n < 0 or n > 100

Todas as expressões do exemplo acima são equivalentes, então podemos escolher qualquer uma como condição para o laço de validação que o resultado será exatamente o mesmo. Consequentemente, podemos escolher aquela que fizer mais sentido para nós.

Por fim, a Codificação 8.8 exibe uma terceira versão em que é utilizada uma flag booleana. Isso pode ser útil em situações em que temos mais de uma condição que pode encerrar o laço ou caso seja necessário, em algum momento posterior, verificar quais restrições foram obedecidas pelo usuário, em especial se o programa precisar da validação de múltiplas entradas.

...

n\_valido = False

while not n\_valido:

n = int(input('Natural entre 0 e 100: '))

if 0 <= n <= 100:

n\_valido = True

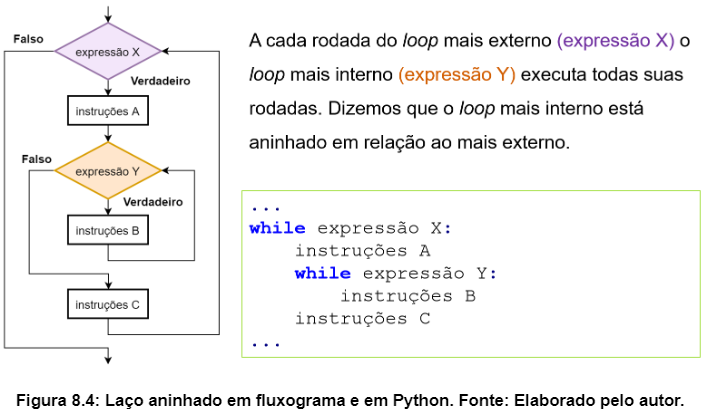
...

###### **Codificação 8.8: Programa com validação de dados de entrada usando flag booleana.**

De início, atribui-se False a flag booleana para forçar que o fluxo de execução acesse o bloco de instruções do laço, que pode ser lido como “repita enquanto a flag não for válida”. Uma vez no laço, recebe-se a entrada e verifica-se as condições da validação e, caso elas sejam atendidas, altera-se a flag para True. Observe que esta solução é semelhante à do laço repeat…until, pois a condição no if é a mesma: “o número é válido?”. Porém, ao invés de break, alteramos o valor da flag, e o laço será interrompido na próxima verificação à condição do while (com break, o laço é interrompido instantaneamente).

**Estruturas de repetição aninhadas**

Há problemas complexos que exigem soluções contendo estruturas de repetição aninhadas, ou seja, um loop dentro do bloco de código de outro loop. Veja na Figura 8.4 um trecho de fluxograma em que há um laço aninhado e a correspondência em Python.



Para compreender a necessidade do uso de laços aninhados, começaremos por um problema bastante simples que evoluirá seu nível de complexidade à medida em que surgirem mais exigências no enunciado. Criaremos um relógio!

De início, a função que simulará um relógio é dada pelo enunciado: “Crie uma função que exibe todos os segundos de um minuto. Lembre-se que o primeiro segundo é 0 e o último é 59.”. A Codificação 8.9 é uma solução válida para esse enunciado.

def relogio():

s = 0

while s < 60:

print(s)

s += 1

A Codificação 8.9 é válida, mas a forma como os segundos são exibidos ainda não se assemelha à de um relógio, então vamos para o próximo enunciado: “Altere a função para que os segundos sejam exibidos no formato de um relógio digital, ou seja, hh:mm:ss. Use a função sleep da biblioteca time para aguardar um segundo entre as exibições”. A Codificação 8.10 é uma solução válida para essa atualização do problema.

from time import sleep

def relogio():

s = 0

while s < 60:

print(f'00:00:{s:02}')

sleep(1)

s += 1

###### **Codificação 8.10: Segunda versão da função que simula um relógio.**

Agora ampliaremos nosso relógio conforme o enunciado: “Altere a função para que a cada 60 segundos o mostrador de minutos seja incrementado e o de segundos zerado. Repita o procedimento até completar 00:59:59”. Precisaremos usar um loop aninhado, pois a cada um minuto executaremos o loop de segundos completamente, lembre-se que temos 60 minutos! A Codificação 8.11 uma solução válida.

def relogio():

m = 0

while m < 60:

s = 0

while s < 60:

print(f'00:{m:02}:{s:02}')

sleep(1)

s += 1

m += 1

###### **Codificação 8.11: Terceira versão da função que simula um relógio.**

Vamos ao próximo enunciado: “Altere a função para que a cada 60 minutos o mostrador de horas seja incrementado e o de minutos zerado. Repita o procedimento até completar 23:59:59”. Agora precisaremos de mais um loop para as horas, veja que com isso teremos três níveis de repetição! A Codificação 8.12 é uma solução válida.

E, em nossa última atualização da função, seguiremos o enunciado: “Altere a função para que a cada 24 horas os três mostradores sejam zerados e o relógio reinicie a contagem.”. Novamente, precisaremos de mais um loop em que todos os anteriores estarão contidos, porém agora será um laço infinito para que o relógio nunca pare. A Codificação 8.13 é uma solução válida para esse novo enunciado.

def relogio():

h = 0

while h < 24:

m = 0

while m < 60:

s = 0

while s < 60:

print(f'{h:02}:{m:02}:{s:02}')

sleep(1)

s += 1

m += 1

h += 1

###### **Codificação 8.12: Quarta versão da função que simula um relógio.**

def relogio():

while True:

h = 0

while h < 24:

m = 0

while m < 60:

s = 0

while s < 60:

print(f'{h:02}:{m:02}:{s:02}')

sleep(1)

s += 1

m += 1

h += 1

###### **Codificação 8.13: Quinta versão da função que simula um relógio.**

Outro exemplo de aplicação de laços aninhados: “crie uma função que receba como parâmetro um número natural n e desenhe um tabuleiro n x n, no qual os quadrados escuros sejam desenhados com dois caracteres █, dado pelo código Unicode 9608, e os quadrados claros com dois espaços. Dica: assumindo que a primeira linha do tabuleiro é a zero, assim como a primeira coluna, podemos descobrir como desenhar os quadrados verificando se a soma de sua linha com sua coluna é par, isto é, quando a soma for par, temos um quadrado escuro, e quando a soma for ímpar, temos um quadrado claro.”. A solução pode ser vista na Codificação 10.14, e a Figura 10.5 ilustra um exemplo de execução da função tabuleiro de 5 por 5 casas.

def tabuleiro(n):

linha = 0

while linha < n:

coluna = 0

while coluna < n:

if (linha+coluna) % 2 == 0:

print(2 \* chr(9608), end='')

else:

print(2 \* ' ', end='')

coluna += 1

print()

linha += 1

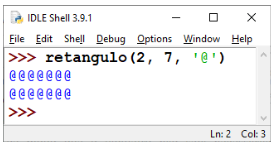
###### **Codificação 8.14: Função que desenha um tabuleiro de tamanho personalizado.**

2) Crie uma nova versão da função relogio em que o horário inicial seja 23:59:59 e a função exiba a contagem regressiva até 00:00:00.

3) Refaça a Codificação 10.14 de modo que o tabuleiro não seja necessariamente quadrado, ou seja, o número de linhas possa ser diferente do número de colunas. Note que será necessário incluir mais um parâmetro na função para representar as colunas.

4) Crie um programa que chame uma função com três parâmetros. O primeiro parâmetro é um natural linhas, o segundo é um natural colunas e o terceiro um caractere s. A função deve exibir um retângulo de tamanho linhas x colunas composto apenas por s, conforme exemplo à direita.

Exemplo:



**Curso Python #15** - [Interrompendo repetições while](https://youtu.be/1OFp_-R2B2A)

**Parte 9 -** [**Sequências e estrutura de repetição for**](https://sites.google.com/faculdadeimpacta.com.br/lp-p9?pli=1&authuser=1) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=AeprvUjsI30)

**Resumo**

Nesta aula os objetivos são: (I) compreender o que são sequências; (II) conhecer os tipos de sequências em Python: strings, listas, tuplas e intervalos; (III) distinguir sequências mutáveis e imutáveis; (IV) distinguir sequências homogêneas e heterogêneas; (V) percorrer sequências por meio de seus índices; (VI) entender a estrutura de repetição com quantidade de repetições definida: laço for; (VII) percorrer sequências por meio de seus itens com o laço for.

**Motivação**

Ao lidarmos com programas maiores e mais complexos, torna-se evidente a necessidade manipular mais dados e, consequentemente, melhorar o gerenciamento deles. Logo, precisamos de estruturas eficientes para manipulação de dados. Neste capítulo introduziremos estruturas que permitirão armazenar grandes quantidades de dados e também um laço de repetição bastante útil para trabalhar com elas!

**Introdução**

Não é raro lidarmos com problemas que exigem manipulação de uma grande quantidade de dados. Nestes casos, pensar em armazená-los em variáveis simples é algo difícil, indesejável e, às vezes, impossível. Para essas situações aprenderemos o conceito de sequências, um recurso importante e frequente em programas mais avançados.

No entanto, não é muito útil armazenar uma imensa quantidade de dados se não puderem ser acessados de modo simples e rápido, para isso aprenderemos a segunda estrutura de repetição do Python, o laço de repetição for, em que a quantidade de repetições do bloco de código é definida antes do loop ser iniciado. Este laço em Python possui uma sintaxe que o torna especialmente útil para acessar itens de sequências, facilitando a escrita e a leitura do código.

**Sequências**

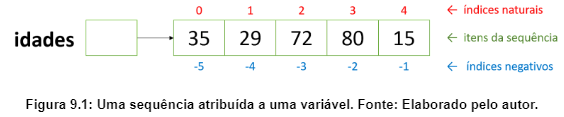
Ao trabalhar com soluções para problemas mais complexos, é frequente a necessidade de manipulação de grandes quantidades de dados. Para lidar com muitos dados, é indispensável uma forma eficiente de armazenamento e acesso, evitando, por exemplo, a criação de diversas variáveis. Até porque, em certos problemas, é impossível criar a quantidade necessária de variáveis para guardar todos os valores, pois muitas vezes a quantidade exata só será conhecida durante a execução do programa. Para esses casos, usaremos um recurso da linguagem Python: as sequências.

Podemos imaginar uma sequência como uma variável com diversos compartimentos, chamados de itens da sequência, inclusive o termo “sequência” indica que há noção de ordem entre os itens (1º, 2º, 3º etc.). Cada item é identificado por dois valores: o nome da variável à qual a sequência foi atribuída e um número inteiro referente à sua posição na sequência, chamado índice.

Em Python, os índices são contados a partir do zero, em ordem crescente, logo o primeiro item tem índice zero, o segundo tem índice um, o terceiro tem índice dois e assim sucessivamente.

Denominamos como tamanho da sequência a quantidade de itens que ela possui. Note que, como consequência da forma como os índices são associados aos itens, começando em zero, o último índice sempre será um a menos que o tamanho da sequência, ou seja, se tivermos uma sequência com 5 itens, o último terá índice 4. O último índice é, portanto, dado pela fórmula tamanho da sequência -1.

No entanto, Python possui uma característica pouco usual em outras linguagens, que é a existência de índices negativos ou regressivos, isto é, os itens são associados a dois índices, um que marca a posição relativa ao início da sequência e outro que marca a posição relativa ao fim dela. Logo, o último item da sequência também está associado ao índice -1, o penúltimo ao índice -2 e assim sucessivamente, sempre reduzindo uma unidade a cada item mais próximo ao início da sequência. A Figura 9.1 é uma ilustração simplificada de uma sequência de números inteiros que foi atribuída à variável idades.

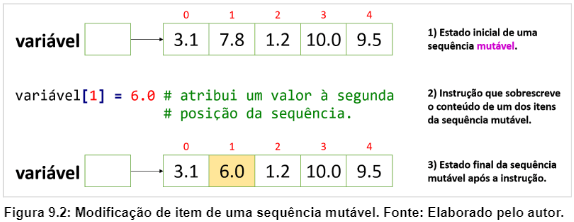


Para acessar um item específico da sequência basta indexá-la adequadamente com a posição em que está o item desejado. A indexação é feita colocando o índice desejado entre colchetes, após o nome da variável. Por exemplo, utilizando a sequência ilustrada na Figura 9.1, podemos acessar o quarto item, que corresponde ao número 80, tanto com a instrução idades[3] quanto com a instrução idades[-2]. A escolha de qual índice usar dependerá do algoritmo, o uso do índice natural é mais comum, mas quando, por algum motivo, for necessário acessar o penúltimo item, é mais fácil usar o índice -2, pois torna desnecessário conhecer o tamanho da sequência.

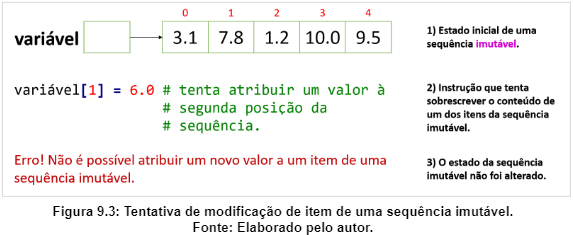
O Python possui diversas funções e métodos que nos ajudam a trabalhar com sequências, por exemplo a função len, que recebe como argumento uma sequência e devolve um número natural indicando o tamanho desta sequência. Com base na Figura 9.1, ao executar a instrução len(idades), obtemos como retorno o valor 5. Neste capítulo não focaremos nessas funções e métodos, porém aprenderemos sobre as características e conceitos associados aos diferentes tipos de sequências do Python.

**Sequências mutáveis e imutáveis**

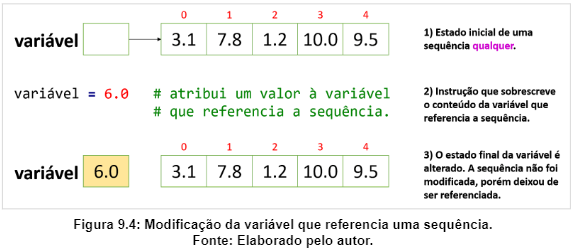
Existem sequências mutáveis e imutáveis. Nas sequências mutáveis cada item se comporta de modo semelhante a uma variável comum, portanto pode receber atribuições, sobrescrevendo seus valores iniciais. Nas sequências imutáveis, uma vez definida a sequência, seus itens não podem mais ser modificados. Veja na Figura 9.2 uma atribuição de valor a um item de uma sequência mutável e sua consequente mudança de estado em decorrência desta operação.



Na Figura 9.3, há uma tentativa de atribuição de valor a um item de uma sequência imutável, o que irá gerar um erro por ser uma operação inválida.



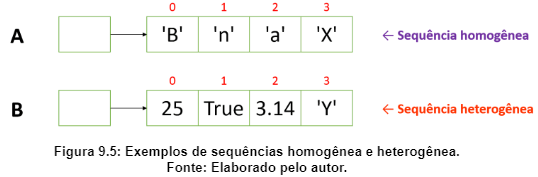
Note que itens de sequências imutáveis não podem ser modificados, porém isso não impede que a variável que referencia a sequência receba um novo valor. É importante observar que, desta forma, sobrescreve-se o valor da variável e não o valor de um item da sequência. A Figura 9.4 ilustra uma atribuição a uma variável que referencia uma sequência (mutável ou imutável, é irrelevante) e, consequentemente, deixa de referenciá-la.



No exemplo da Figura 9.4, caso a sequência não esteja referenciada por outra variável qualquer, será automaticamente apagada da memória pelo Python, pois será interpretado que é uma sequência sem uso, pois não há como acessá-la. A relação completa de operações disponíveis para sequências é chamada de “Operações Comuns de Sequências” (PSF, 2021a). Para as sequências mutáveis, também existem as operações listadas em “Tipos de Sequências Mutáveis” (PSF, 2021b).

**Sequências homogêneas e heterogêneas**

Existem sequências homogêneas e sequências potencialmente heterogêneas, característica que indica a flexibilidade em armazenar itens de tipos distintos simultaneamente. Em sequências homogêneas, todos os itens devem ser do mesmo tipo, portanto, caso um item seja um número inteiro, todos os demais deverão ser números inteiros, caso seja um caractere, todos os demais também serão caracteres, e assim por diante. Em sequências potencialmente heterogêneas os itens podem ser de tipos distintos, logo uma mesma sequência pode conter itens float, int, bool etc. A Figura 9.5 ilustra esse conceito.



**Tipos de sequências em Python**

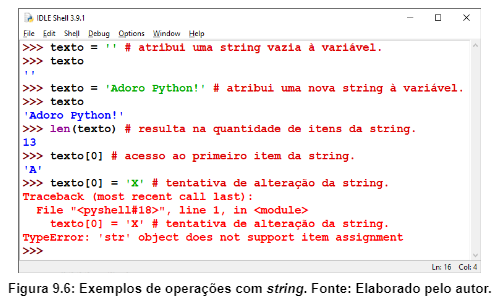
O Python possui três tipos básicos de sequências: listas (list), tuplas (tuple) e intervalos (range). Também há um tipo de sequência específica para armazenar caracteres (strings) e outros tipos de sequência dedicadas para manipulação de dados binários. Nesta aula abordaremos os tipos string, lista, tupla e intervalo.

**Strings**

Já introduzimos o tipo de dados string, porém, propositalmente, não trabalhamos com suas características de sequência. As strings em Python são sequências imutáveis e homogêneas, em que todos os itens devem ser caracteres Unicode. Assim, como em outras sequências, é possível acessar um item específico da string indexando-a. Podem ser definidas de várias formas:

1. Pares de apóstrofos: 'a casa amarela';
2. Pares de aspas: "a casa amarela";
3. Pares de três apóstrofos ou aspas, permitindo que tenham múltiplas linhas: '''a casa amarela''' ou """a casa amarela""";
4. Com o construtor de tipo str(), que permite também a conversão de outros tipos de dados para string.

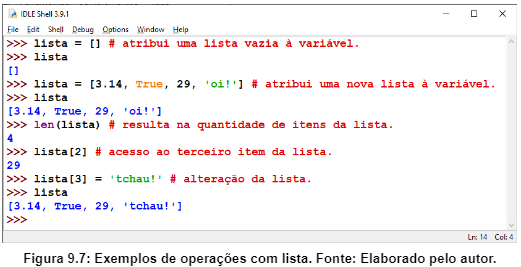
Indica-se string vazia com um par de delimitadores sem nenhum conteúdo entre eles ''. Observe que a string ' ' não está vazia, pois contém um caractere de espaço. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.6.



**Listas**

Sequências do tipo list são mutáveis e potencialmente heterogêneas. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.7, que pode ser definidas de várias formas:

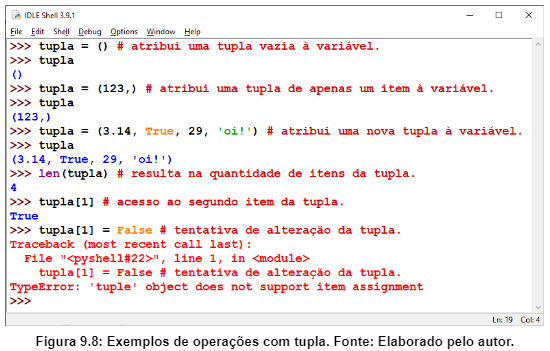
1. Apenas um par de colchetes, para indicar lista vazia: [];
2. Itens separados por vírgulas entre um par de colchetes: [4, True, 1.5];
3. Com compreensão/abrangência de lista (não será abordado por enquanto);
4. Com o construtor de tipo list(), que pode ser usado também para converter outros tipos de sequências em listas.



**Tuplas**

Sequências do tipo tuple são imutáveis e potencialmente heterogêneas, seu uso é comum quando é necessário garantir que a sequência não seja modificada, como em chaves de dicionários ou ao ser passada como argumento para funções que produzam efeitos colaterais. Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.8. Assim como listas, tuplas podem ser definidas de várias formas:

1. Apenas um par de parênteses para indicar tupla vazia: ();
2. Uma vírgula à direita, indicando um tupla de só um item: 7, ou (7,);
3. Itens separados por vírgulas: 4, True, 1.5 ou (4, True, 1.5);
4. Com o construtor de tipo tuple(), que pode ser usado também para converter outros tipos de sequências em tuplas.



**Intervalos**

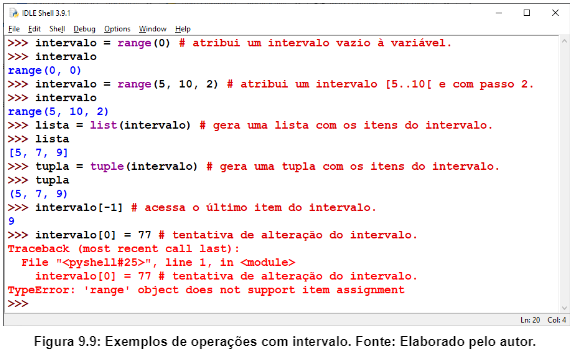
Sequências do tipo range são imutáveis e homogêneas, gerando um intervalo de números inteiros, algo útil principalmente para controle de laços de repetição. Intervalos são criados com uso do construtor range() acompanhado dos argumentos início, fim e passo, que devem ser números inteiros. Existem três variações para criação de intervalos, dependendo do número de argumentos:

1. Três argumentos: range(início, fim, passo), gera uma sequência com itens no intervalo [início..fim[, variando de passo em passo;
2. Dois argumentos: range(início, fim), gera uma sequência crescente com itens no intervalo [início..fim[, variando de 1 em 1;
3. Um argumento: range(fim): gera uma sequência crescente com itens no intervalo [0..fim[, variando de 1 em 1.

Observe algumas implicações da forma como sequências range funcionam:

1. O intervalo é aberto à direita, portanto fim não pertence à sequência;
2. Dependendo de como o construtor é especificado, pode-se gerar um intervalo vazio, por exemplo em range(10, 10), pois não existem inteiros no intervalo [10..10[. Ficou em dúvida? Isso equivale a solicitar um intervalo que comece em 10 e termine antes de 10;
3. No construtor com três argumentos, range(início, fim, passo), se passo for um número negativo, a sequência será decrescente, desde que início seja maior do que fim, caso contrário será gerado um intervalo vazio. Logicamente, passo zero é inválido;
4. Podemos interpretar range como uma função que retorna um intervalo com base em uma progressão aritmética cuja razão entre os termos é o passo.

Veja alguns exemplos de manipulação desse tipo de sequência na Figura 9.9.



**Percorrendo sequências**

Percorrer, iterar ou “varrer” uma sequência equivale a acessar sistematicamente diversos de seus itens seguindo alguma ordem. Frequentemente, percorre-se uma sequência para acessar seus itens visando exibição, alteração ou utilização dos valores. Percursos em sequências costumam ser feitos com uso de laços de repetição.

Para entender a necessidade das sequências e a razão de percorrê-las, pensaremos sobre o seguinte enunciado: “Crie um programa que receba como entrada quatro salários, exiba a média salarial e os salários abaixo da média”. A Codificação 9.1 contém uma solução válida para esse problema, porém sem o uso de sequências.

soma = 0

salario\_0 = float(input('Salário: R$ '))

soma += salario\_0

salario\_1 = float(input('Salário: R$ '))

soma += salario\_1

salario\_2 = float(input('Salário: R$ '))

soma += salario\_2

salario\_3 = float(input('Salário: R$ '))

soma += salario\_3

media = soma / 4

print(f'Média = R$ {media:.2f}')

if salario\_0 < media:

print(f'Abaixo da média: R$ {salario\_0:.2f}')

if salario\_1 < media:

print(f'Abaixo da média: R$ {salario\_1:.2f}')

if salario\_2 < media:

print(f'Abaixo da média: R$ {salario\_2:.2f}')

if salario\_3 < media:

print(f'Abaixo da média: R$ {salario\_3:.2f}')

###### **Codificação 9.1: Programa que exibe os salários abaixo da média (sem sequências).**

Na Codificação 9.1 foi necessário criar quatro variáveis para guardar quatro entradas do usuário. Se fossem mil salários, seriam necessárias mil variáveis. Porém, podemos construir uma versão alternativa que, ao invés de usar variáveis simples, usará uma lista para guardar os salários, conforme a Codificação 9.2.

salarios = [0, 0, 0, 0]

soma = 0

salarios[0] = float(input('Salário: R$ '))

soma += salarios[0]

salarios[1] = float(input('Salário: R$ '))

soma += salarios[1]

salarios[2] = float(input('Salário: R$ '))

soma += salarios[2]

salarios[3] = float(input('Salário: R$ '))

soma += salarios[3]

media = soma / 4

print(f'Média = R$ {media:.2f}')

if salarios[0] < media:

print(f'Abaixo da média: R$ {salarios[0]:.2f}')

if salarios[1] < media:

print(f'Abaixo da média: R$ {salarios[1]:.2f}')

if salarios[2] < media:

print(f'Abaixo da média: R$ {salarios[2]:.2f}')

if salarios[3] < media:

print(f'Abaixo da média: R$ {salarios[3]:.2f}')

###### **Codificação 9.2: Programa que exibe os salários abaixo da média (com uso de lista).**

Você notou que a Codificação 9.2 é maior que a Codificação 9.1 e talvez imagine que não houve benefício em substituir as variáveis simples pela lista. E você está certo! Do modo como a lista foi usada, não houve ganho e a legibilidade foi prejudicada, isso porque a usamos de modo idêntico às variáveis simples, escrevendo uma instrução para cada acesso e atribuição, mesmo que essas instruções estejam praticamente idênticas, variando apenas os índices dos itens. O que precisamos é de uma abordagem diferente.

Podemos percorrer listas variando apenas seu índice, uma vez que o nome da variável que referencia a sequência é sempre o mesmo. Com esse recurso, podemos usar uma variável como índice e alterá-la de acordo com a necessidade de indexação. Essa flexibilidade permite a abordagem da Codificação 9.3, em que repetimos as atribuições e acessos aos itens usando loop ao invés de reescrever as instruções.

salarios = [0, 0, 0, 0]

soma = 0

i = 0 # variável que será usada como índice.

while i < 4:

salarios[i] = float(input('Salário: R$ '))

soma += salarios[i]

i += 1

media = soma / 4

print(f'Média = R$ {media:.2f}')

i = 0 # variável que será usada como índice.

while i < 4:

if salarios[i] < media:

print(f'Abaixo da média: R$ {salarios[i]:.2f}')

i += 1

###### **Codificação 9.3: Programa que exibe os salários abaixo da média (com while).**

Assim, reduziu-se o código e, principalmente, o programa tornou-se mais flexível, pois caso seja necessário armazenar mais salários, basta realizar pequenas mudanças na codificação, algo potencialmente inviável se estivéssemos limitados às variáveis simples ou sem laços para percorrer sequências.

Agora nossos programas podem trabalhar com grandes quantidades de dados com o uso de sequências e laços de repetição para percorrê-las. No entanto, a linguagem Python possui um segundo loop que pode tornar nossos códigos ainda mais concisos e elegantes, principalmente ao lidarmos com sequências, é a estrutura de repetição for.

**Estrutura de repetição for**

Python tem uma estrutura de repetição com quantidade de repetições definida, o for. Geralmente, esse laço é usado quando sabemos antecipadamente a quantidade de vezes que o bloco de código deve ser repetido, ou para percorrer sequências.

Em muitas linguagens, for é “atalho” para um while com número pré-definido de iterações, agrupando em uma mesma linha as três partes que controlam o laço: (I) inicialização da variável de controle; (II) condição de repetição e (III) incremento da variável de controle. Na Codificação 9.4 há um exemplo deste tipo de laço for.

for (i = 0; i < n; i += 1)

<bloco de código>

No entanto, esse tipo de laço for tradicional não existe em Python. Em seu lugar temos um laço que opera de maneira mais intuitiva sobre as sequências, e que reduz a chance de erros na manipulação dos índices, que é necessária para escrevermos tanto o laço while, vistos nos exemplos anteriores, quanto o laço for tradicional.

Esse laço é conhecido, em inglês, como for each, que pode ser traduzido como “para cada”, e podemos então interpretar essa estrutura de repetição como “para cada item da sequência, faça…”. Veja na Codificação 9.5 a sintaxe adotada pelo Python.

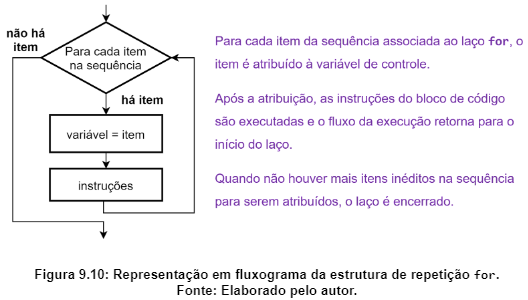
for <variável> in <sequência>:

<bloco de código>

###### **Codificação 9.5: Estrutura do laço de repetição for do Python.**

Simplificadamente, podemos entender <sequência> como uma sequência de qualquer tipo e tamanho, até vazia. Já <variável> é uma variável de controle que, caso não exista previamente, será criada no início da execução do laço, se a sequência não for vazia, pois o primeiro item será atribuído a ela dando início à primeira execução do <bloco de código>. Caso exista variável com identificador igual à <variável>, seu valor será sobrescrito pela atribuição do primeiro item de <sequência>.

O for executa seu bloco de código uma vez para cada item de <sequência> atribuído à <variável>, tornando a quantidade de rodadas previsível. Não é recomendado alterar <sequência> durante a execução do for, pois pode gerar efeitos inesperados. A Figura 9.10 ilustra o comportamento deste laço.



Reescreveremos a Codificação 9.3 usando for. Usaremos também um método muito requisitado quando trabalhamos com listas: o append(), que anexa ao final da própria lista um novo item. Assim, é desnecessário que a lista inicie já com todas as posições necessárias, deixando o Python gerenciar a alocação de memória da forma que ele julgar mais eficiente. Veja na Codificação 9.6 uma possível solução com for.

salarios = []

soma = 0

for \_ in range(4):

salario = float(input('Salário: R$ '))

soma += salario

salarios.append(salario)

media = soma / 4

print(f'Média = R$ {media:.2f}')

for salario in salarios:

if salario < media:

print(f'Abaixo da média: R$ {salario:.2f}')

###### **Codificação 9.6: Programa que exibe os salários abaixo da média (com for).**

Na Codificação 9.6 foram usados dois laços for: (I) um para receber os valores dos salários, acumulá-los na variável soma e adicioná-los à lista e; (II) outro para percorrer a lista de salários e exibir os inferiores a média. Note que o nome da variável referência seu conteúdo e está no plural, indicando que é uma sequência de salários.

No primeiro laço, usamos como identificador da variável de controle apenas um sublinhado (\_), o que pode parecer estranho a princípio, porém indica ao leitor do código que não há interesse nos valores assumidos por essa variável. Poderíamos ter usado qualquer outro nome válido, mas esse identificador é comum quando o objetivo do laço é apenas ser executado um número fixo de vezes, sem pretensão de usar os valores da sequência atribuídos à variável de controle.

Note que a função range retorna um intervalo [0..4[, e portanto o laço será executado 4 vezes, como usamos append para incluir os itens na lista, não precisamos nos preocupar em gerenciar índices.

No segundo laço, percorremos a lista de salários, independente do seu tamanho, e os valores dos itens serão usados para algo, no caso uma exibição condicionada a uma verificação. Portanto, foi adotado um identificador que faz referência ao conteúdo da lista, só que agora no singular, pois está representando apenas um item da lista por vez.

Essas convenções adotadas na Codificação 9.6 não são obrigatórias, mas facilitam a leitura do código ao dar maior expressividade às instruções, tornando nosso trabalho como programadores mais fácil e agradável.

**Exemplos de aplicação do laço for**

Para melhorar a compreensão sobre o laço de repetição for, teste os programas das Codificações 9.7 a 9.10.

# Crie um programa que exiba o alfabeto minúsculo e maiúsculo.

for letra in 'abcdefghijklmnopqrstuvwxyz':

print(letra)

print()

for letra in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':

print(letra)

###### **Codificação 9.7: Programa que exibe as letras do alfabeto (com for).**

# Crie um programa que leia cinco nomes e exiba a quantidade de

# nomes que começam com vogal.

nomes = []

for \_ in range(5):

nomes.append(input('Nome: '))

qtd = 0

for nome in nomes:

if (nome[0]=='A' or nome[0]=='E' or nome[0]=='I' or

nome[0]=='O' or nome[0]=='U'):

qtd += 1

print(f'{qtd} dos nomes começam com vogal')

###### **Codificação 9.8: Programa que conta nomes que começam com vogal (com for).**

# Crie um programa que exiba os sete dias da semana.

dias = ('domingo', 'segunda', 'terça', 'quarta', 'quinta',

'sexta', 'sábado')

for dia in dias:

print(dia)

###### **Codificação 9.9: Programa que exibe os setes dias da semana (com for).**

# Crie um programa que exiba, em ordem crescente, os pares de 10

# até 100 e, em ordem decrescente, os ímpares de 100 até 10.

for par in range(10, 101, 2):

print(par, end=' ')

print()

for impar in range(99, 10, -2):

print(impar, end=' ')

###### **Codificação 9.10: Programa que exibe os pares e ímpares de um intervalo (com for).**

Curso Python #013​ - Estrutura de repetição for. Disponível em: [Curso Python #013 - Estrutura de repetição for](https://youtu.be/cL4YDtFnCt4)