INSTITUTO DE GESTÃO E TECNOLOGIA
DA INFORMAÇÃO

Fundamentos de Desenvolvedor(a) Python

Antônio Carlos de Nazaré Júnior

2022

Fundamentos de Desenvolvedor(a) Python

Bootcamp Desenvolvedor(a) Python

Antônio Carlos de Nazaré Júnior

Sumário

Capítulo 1. Introdução ao Python

Python é uma linguagem livre e moderna, criada por Guido van Rossum em 1991, e tem como um dos maiores diferenciais a facilidade de aprender, por causa do seu código limpo e organizado. A linguagem é bastante versátil, sendo empregada no desenvolvimento de soluções em diversas áreas, como Ciência de Dados e Programação *WEB*. Adicionalmente, a demanda por desenvolvedores Python é muito alta, afinal, é a linguagem de programação mais popular entre os desenvolvedores do mundo todo[1].

Muitas pessoas acreditam que o nome Python tem alguma relação com a espécie de cobra com o mesmo nome (píton, em português), até mesmo porque o logotipo do Python é composto por duas cobras, uma azul e outra amarela. Entretanto, segundo a documentação oficial[2], o nome foi uma homenagem ao grupo humorístico britânico *Monty Python*, do qual o criador da linguagem, Guido van Rossun, era fã. Assim, quando estava desenvolvendo a linguagem, ele pensou que precisava de um nome que fosse curto, único e misterioso. Assim, nomeou o projeto de **Python**.

Este módulo foi planejado de forma a introduzir o Python, apresentando e discutindo seus principais conceitos e características, de uma forma prática. Você, aluno, irá se familiarizar com a sintaxe, os tipos e estruturas de dados, os fluxos de controle e as principais funcionalidades da linguagem. Portanto, espero que, ao final do curso, você esteja apto a desenvolver o seu primeiro programa e que possa prosseguir na caminhada para se tornar um desenvolvedor Python.

[1] De acordo com o ranking da empresa TIOBE, especializada em avaliação de softwares (TIOBE, 2022).

[2] https://docs.python.org/3/faq/general.html

Características da Linguagem

De acordo com Sebesta (SEBESTA, 2018), o Python pode ser classificado como uma linguagem de *alto-nível*, *interpretada*, com *tipagem dinâmica* e *multiparadigma*. A seguir, iremos discutir cada uma destas características.

Linguagem de Alto Nível	Enquanto uma linguagem de baixo nível é projetada com foco no controle do hardware, uma linguagem de alto nível, por outro lado, é projetada pensando no desenvolvedor, na sua facilidade de escrita, legibilidade do código, capacidade de abstração e flexibilidade.
Linguagem Interpretada	As linguagens podem ser *compiladas* ou *interpretadas*. Uma linguagem *compilada* tem o seu código-fonte traduzido (compilado) para um código de máquina antes de ser executado. Linguagens *interpretadas* pulam esta etapa, e o código-fonte do desenvolvedor é executado (interpretado) diretamente por um outro software chamado **interpretador.**
Tipagem Dinâmica	Tipagem dinâmica talvez seja uma das funcionalidades mais conhecidas do Python. Essa característica possibilita que não seja necessária a declaração dos tipos de dados. Desta forma, o Python é capaz de selecionar, de forma dinâmica, qual o tipo de dado mais adequado para cada variável declarada. O Capítulo 3 apresenta mais detalhes sobre esta característica do Python.
Multiparadigma	Python é considerado uma linguagem multiparadigma, pois oferece a possibilidade de o desenvolvedor utilizar diferentes paradigmas de programação: *procedural*, *funcional* e *orientado a objetos*. Ao longo dos próximos capítulos serão apresentados conceitos e exemplos destes paradigmas oferecidos pela linguagem.

Vantagens e Desvantagens

Muitos desenvolvedores, ao iniciarem um novo projeto, se perguntam se a linguagem escolhida é a mais adequada para a solução. Por um lado, Python é versátil, é simples de utilizar e possuí suporte de uma comunidade entusiasmada; por outro lado, Python também tem suas desvantagens.

Esta seção apresenta os principais aspectos do Python que são considerados vantajosos, e aqueles que fazem com que ele não seja a melhor escolha para um determinado projeto.

Vantagens

Existem diversas vantagens na utilização do Python e a seguir discutiremos as principais:

Facilidade de Aprendizado	Python se concentra na legibilidade do código. É elegante, versátil, didático e bem estruturado. Por ser *dinamicamente tipado*, se torna uma linguagem amigável e de rápido desenvolvimento. A sua curva de aprendizado é baixa e isto faz com que o Python seja a linguagem escolhida por vários professores, de diversas universidades no mundo, para as aulas dos cursos introdutórios de programação.
Versatilidade e Flexibilidade	Devido à flexibilidade da linguagem, é fácil realizar análises exploratórias, básicas e complexas. O Python permite extrair o melhor dos diferentes paradigmas de programação. Apesar de ser orientado à objetos, ele também adota ativamente recursos de linguagens procedurais e funcionais.

Prototipação e Produtividade	O principal lema da linguagem é "*Fazer mais, com menos código*!", o que significa construir protótipos de soluções e testar ideias muito mais rapidamente em Python do que em outras linguagens. Ou seja, sua utilização não apenas economiza muito tempo, mas também aumenta a produtividade e reduz os custos de desenvolvimento.
Diversidade de Bibliotecas	É possível encontrar uma biblioteca[3] para as mais diversas aplicações em que podemos pensar: desenvolvimento web, desenvolvimento de jogos, computação numérica, mercado financeiro, inteligência artificial e muitas outras. E se ainda não houver uma disponível, podemos facilmente criar a nossa!
Open Source	É possível baixar o Python gratuitamente e começar a escrever códigos em questão de minutos. Seu licenciamento permite que qualquer pessoa o utilize, o modifique e o distribua livremente. Além disso, a extensa comunidade de desenvolvedores Python é muita ativa. Facilmente conseguiremos ajuda dos mais diversos e experientes especialistas.
Portabilidade	Por ser uma linguagem interpretada, e não compilada, o Python é compatível com qualquer sistema operacional, sem a necessidade de modificações no código. Ou seja, é necessário escrever o código apenas uma vez e ele será compatível com qualquer plataforma que suporte Python.

Desvantagens

Apesar de todas as vantagens apresentadas, o Python possuí certas limitações que devem ser consideradas antes de sua adoção em um novo projeto. A seguir discutiremos as principais desvantagens do Python em relação à outras linguagens.

[3] Bibliotecas são coleções de códigos que podem ser distribuídas e reaproveitadas em diversos projetos.

Limitações de Desempenho	Como visto anteriormente, Python é uma linguagem interpretada e dinamicamente tipada, o que a torna poderosa em diversos aspectos. Entretanto, esta natureza versátil dela, é também motivo para que ela não tenha o mesmo desempenho computacional de outras linguagens populares. Determinados algoritmos, quando escritos em linguagens como Java ou C++, podem ser executados em até centenas de vezes mais rápidos (SHAW, 2018). Mas se velocidade de execução não for o requisito mais importante para o seu projeto, o Python ainda poderá ser uma ótima escolha. Além disso, existem diversas maneiras de se otimizar o código para aumentar, consideravelmente, o desempenho computacional (SLATKIN, 2016).
Consumo de Memória	Outro ponto a ser considerado é o alto consumo de memória, que, quando comparado com outras linguagens, é uma grande desvantagem. Portanto, o Python pode não ser a melhor escolha para tarefas com uso intensivo de memória. Este alto consumo, na maioria das vezes, é causado por problemas de liberação de memória por parte do *Garbage Collector* (*GC*)[4] da linguagem. Entretanto, estratégias também podem ser adotadas para amenizar esta limitação.

[4] Garbage collector (GC) é um processo interno do Python para automação do gerenciamento de memória, possibilitando recuperar uma área de memória inutilizada por um programa (SALES, 2020).

Dispositivos Móveis	Python não possui desenvolvimento nativo para plataformas móveis. Os principais sistemas operacionais deste ambiente, *Android* e *iOS,* não suportam Python como linguagem de programação oficial, criando assim uma limitação para os desenvolvedores que desejam utilizar suas aplicações em *smartphones* e *smartwatches*. Ainda assim existem ferramentas que permitem este tipo de utilização, entretanto requerem um esforço adicional. O exemplo mais notável é o *Kivy*[5], que permite a compatibilidade de aplicações Python com diferentes plataformas móveis.
Problemas com Paralelização	Escrever códigos que utilizam o conceito de *threading* está longe de ser um ponto forte do Python. Threads são fluxos de códigos que podem ser executados de forma paralela e, consequentemente, permitem um ganho considerável no desempenho computacional. Ou seja, o programa será capaz de executar várias tarefas ao mesmo tempo, utilizando diferentes núcleos do processador. Entretanto, devido às limitações do *Global Interpreter Lock* (*GIL*)[6], o Python restringe que apenas uma *thread* pode ser executada por vez, perdendo, assim, os benefícios de desempenho que o paralelismo proporciona. Felizmente, existem bibliotecas de multiprocessamento que foram desenvolvidas para contornar este problema. Entretanto, elas exigem graus de esforço e conhecimento maiores do desenvolvedor, além de que, aumentam consideravelmente o consumo de memória na maioria das situações (GORELICK e OZSVALD, 2020).

[5] Página do projeto Kivy: https://kivy.org

[6] *GIL* é um mecanismo adotado em linguagens interpretadas, como o Python, para o gerenciamento de execução de *threads*.

Preparação do Ambiente Python

Como vimos, uma das vantagens do Python é a sua portabilidade. Por ser uma linguagem de programação *cross-platform*, ela pode ser executada em várias plataformas, como os sistemas operacionais *Windows*, *macOS* e *Linux* e em distintas arquiteturas como *desktops* (32 ou 64 *bits*) e dispositivos embarcados como o *raspberry*.

Enquanto os sistemas *Windows* não possuem o Python pré-configurado, a maioria das distribuições *Linux* e os sistemas *MacOS*, já possuem o Python instalados e configurados. Entretanto, mesmo com uma versão pré-configurada, é sempre recomendado instalar e atualizar a versão mais recente do Python.

Apesar de não ser o foco do curso, pois utilizaremos uma ferramenta *online* com o Python já configurado, nesta seção iremos indicar os passos para a instalação do Python, em diferentes sistemas operacionais.

Talvez, a maneira mais fácil de instalar o Python, em qualquer um dos sistemas operacionais mencionados, seja a utilização do *Anaconda*[7]. Ao utilizá-lo, além do próprio Python, teremos acesso à um conjunto de pacotes e ferramentas para o desenvolvimento de códigos em Python, principalmente para o uso em *Ciência de Dados* e *Inteligência Artificial*. Entretanto, se você não deseja utilizar o *Anaconda*. Você pode instalar e/ou atualizar o Python separadamente no seu sistema operacional, de acordo com os tutoriais[8] (em inglês ou português) indicados a seguir:

SISTEMA	TUTORIAIS

[7] O Anaconda é gratuito para utilização individual. O *download*, assim como as instruções de instalação, para cada sistema operacional está disponível na página do projeto: https://www.anaconda.com

[8] Os tutoriais indicados são dos *websites* (REAL PYTHON, 2020) e (PYTHON BRASIL, 2022), foram escolhidos por serem simples, mas ao mesmo tempo completos.

Windows	🕮 *How to Install Python on Windows* (inglês): https://realpython.com/installing-python/#how-to-install-python-on-windows 🕮 *Instalando o Python 3 no Windows* (português): https://python.org.br/instalacao-windows/
Linux	🕮*How to Install Python on Linux* (inglês): https://realpython.com/installing-python/#how-to-install-python-on-linux 🕮 Instalando o Python no Linux (português): https://python.org.br/instalacao-linux/
MacOs	🕮 *How to Install Python on macOS* (inglês): https://realpython.com/installing-python/#how-to-install-python-on-macos 🕮 Instalando o Python no Mac OS (português): https://python.org.br/instalacao-mac/

Uma vez que tenhamos o Python instalado, é possível iniciarmos a execução de código imediatamente, por meio do chamado *modo interativo*. Para isto, basta abrirmos o nosso terminal/console de comandos e digitar o comando `python`. Isto irá abrir o interpretador do Python imediatamente em seu modo interativo.

A partir deste momento podemos digitar o nosso código e pressionar `enter`. Em seguida o interpretador retornará o resultado da execução do código declarado. Por exemplo, vamos digitar `1 + 1` e apertar `enter` em seguida. O interpretador nos retornará o resultado (`2`). Para sair, basta digitar `quit()` e pressionar `enter`. Veja no exemplo abaixo a nossa sequência de comandos, que está destacada em amarelo.

```
(base) acnazarejr@antoniomb ~ % python
Python 3.8.8 (default, Apr 13 2021, 12:59:45)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 1
2
>>> quit()
(base) acnazarejr@antoniomb ~ %
```

Outra maneira de executarmos um código Python é o *modo de script*. Em que escrevemos todo o código em um arquivo texto, com a extensão .py, e em seguida o executamos por meio do comando:

```
python arquivo.py
```

A escrita do código pode ser realizada em qualquer editor de texto, entretanto os desenvolvedores utilizam ferramentas específicas que são chamadas de *IDE* (*Integrated Development Environment*). Estas ferramentas permitem, além da escrita do código, a execução do programa, a visualização dos resultados, a realização de testes e várias outras tarefas que fazem parte da rotina de um desenvolvedor. Atualmente, as IDE's mais utilizadas para o Python são: *VSCode*[9], *PyCharm*[10], *Atom*[11] e *Sublime Text*[12].

Entretanto, aprender a programar uma nova linguagem pelo *modo de script* pode não ser produtivo, pois, a cada novo comando que testarmos, teremos que salvar o arquivo, abrir o terminal e mandar executar o arquivo com o código.

Desta forma, para nos concentrarmos apenas no aprendizado da linguagem e não termos que nos preocupar com configurações e comandos extras (que não sejam da própria linguagem), iremos utilizar uma ferramenta gratuita e *online*, desenvolvida pelo Google, chamada *Google Colab*.

[9] https://code.visualstudio.com/

[10] https://www.jetbrains.com/pt-br/pycharm/

[11] https://atom.io/

[12] https://www.sublimetext.com/

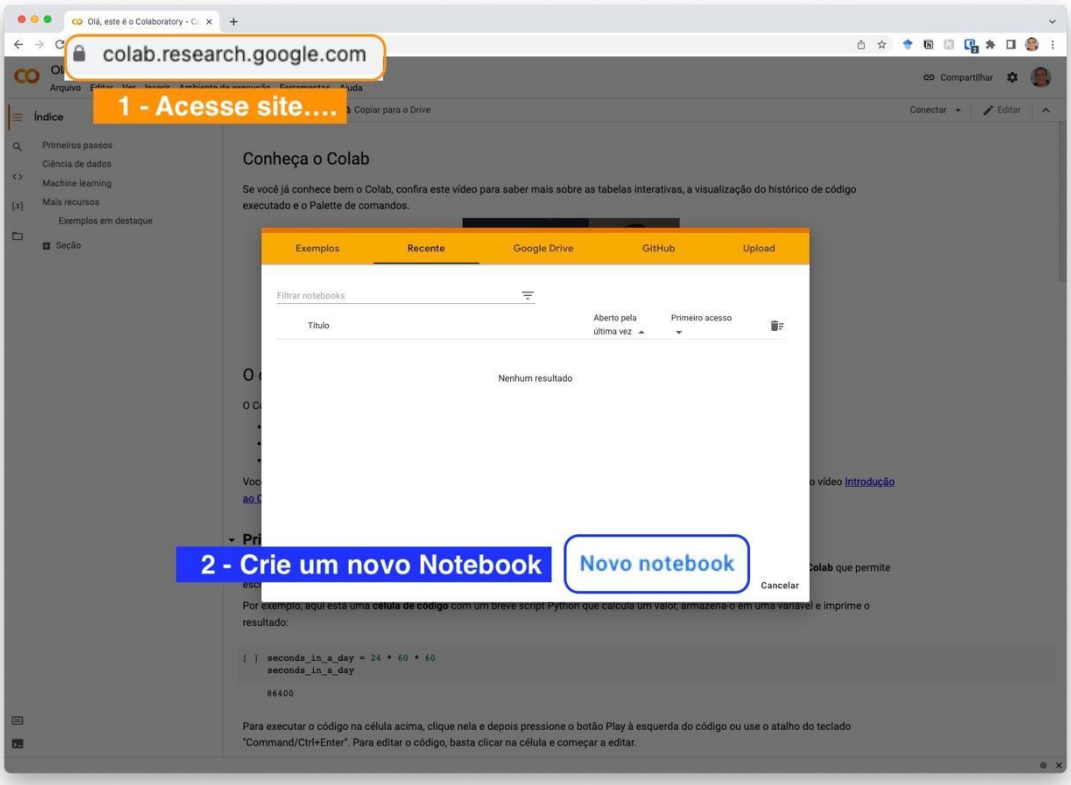
Google Colab

O *Google Colab* é uma ferramenta gratuita hospedado na nuvem da própria Google e que foi criada para possibilitar que desenvolvedores e pesquisadores, principalmente da área de Inteligência Artificial, possam escrever seus códigos em um ambiente integrado, sem a necessidade de configuração e com um poder computacional necessário para executar seus programas.

Ele utiliza o conceito de *notebooks* (cadernos), que permite que intercalemos o código fonte (escritos em células), com a saída de cada execução (*outputs*). E tudo isso em um ambiente colaborativo (por isso o nome *Colab*, de *Colaboratory*), onde podemos compartilhar nossos códigos com outras pessoas.

A escolha desta ferramenta para o nosso curso é devido à sua facilidade em executar um código Python, sem necessitar de configurações adicionais. Vamos utilizá-lo para executarmos todos os códigos de exemplos do curso. E uma vez que a ferramenta estará sempre disponível, pronta para ser utilizada, é fortemente recomendado que não nos limitemos apenas aos códigos de exemplos. É importante irmos além, alterando estes códigos e, ao mesmo tempo, aprendendo novos conceitos.

Para utilizar esta ferramenta, basta ter uma conta ativa do Google, acessar o endereço do serviço: https://colab.research.google.com/ e em seguida clicar em *Novo notebook*, conforme a imagem abaixo. Também é possível explorar o *notebook* de boas-vindas, que é sempre aberto ao acessar este endereço. Ele possuiu um tutorial de introdução dos principais conceitos do *Colab*.



Uma vez que criarmos um *notebook*, já estaremos prontos para escrevermos e executarmos o nosso primeiro código, conforme a próxima seção. A ferramenta também permite que você salve os seus *notebooks* (ou seja, os seus arquivos de códigos) diretamente no *Google Drive*[13], para que você possa acessá-los posteriormente.

Executando o Primeiro Código

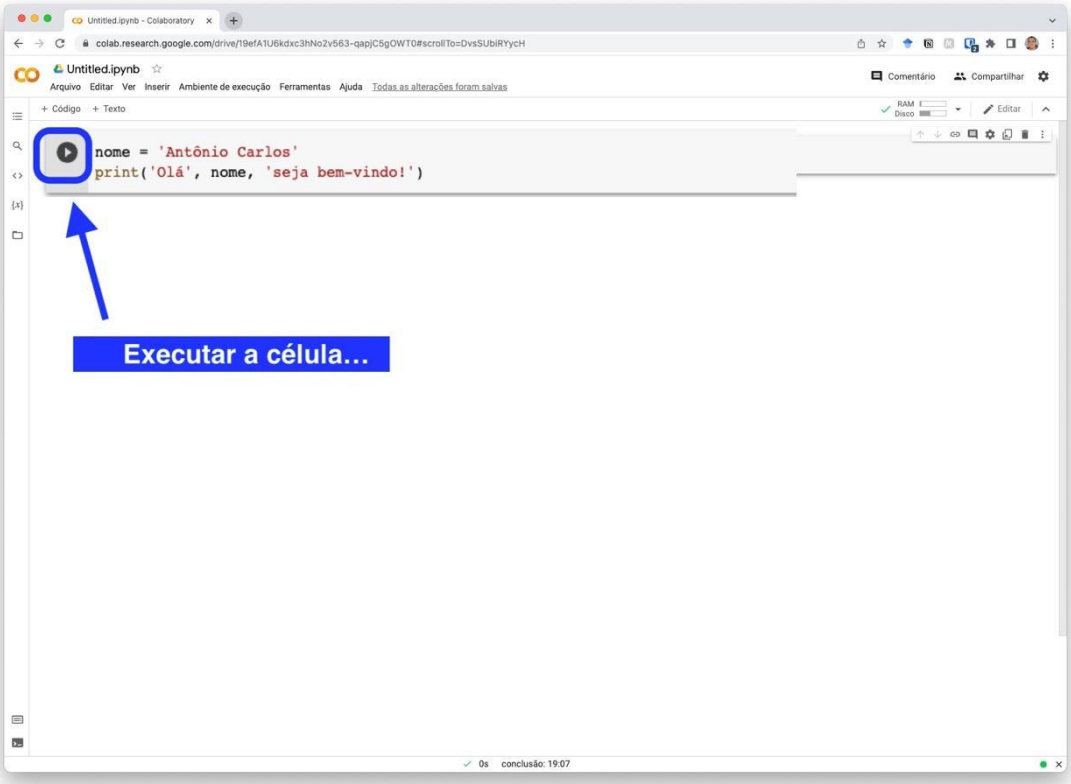
Uma vez que agora já sabemos como entrar no *Google Colab*, podemos então escrever e executar o nosso primeiro código. Para tanto, vamos utilizar as células do

[13] Google Drive é o serviço de armazenamento gratuito do Google: https://drive.google.com/

notebook que criamos. Digite na primeira célula o seguinte código (ou fiquem à vontade para copiar e colar):

```
nome = 'Antônio Carlos'
print('Olá', nome, 'seja bem-vindo!')
```

Por enquanto, o significado de cada uma das palavras (instruções) e a estrutura do código podem não fazer nenhum sentido e é justamente isso que vamos aprender ao longo do curso! Após digitarmos o código na célula, ela deve ter a aparência da imagem abaixo.



Em seguida, para executarmos a célula basta clicarmos no botão de execução, conforme a imagem também ilustra. Ou também podemos executar teclando "crtl +

enter" (ou "cmd + enter" em computadores com MacOs). Após a execução teremos a seguinte saída:

```
Olá Antônio Carlos seja bem-vindo!
```

Pronto! Acabamos e executar o nosso primeiro código em Python. Agora convido a todo a substituir o nome "Antônio Carlos" pelo seu próprio nome, ou outras palavras, e executar a célula novamente.

Capítulo 2. Escrita de Códigos em Python

Inicialmente, o Python era utilizado como uma linguagem didática para o ensino de programação (FANGOHR, 2004). Entretanto, sua facilidade de uso e sua clareza o levaram a ser adotado por desenvolvedores de diversos níveis. Assim, o Python passou a ser chamado de "pseudocódigo[14] executável" por alguns. De fato, a experiência mostra que na maioria das vezes é muito mais fácil ler e entender um código escrito em Python, do que em outras linguagens.

Neste capítulo iremos discutir os principais elementos e regras de escrita que tornam o desenvolvimento de códigos em Python tão atraente, tanto nos aspectos de simplicidade e clareza, quanto na facilidade de aprendizado.

A Sintaxe da Linguagem

Existem dois componentes principais que definem uma linguagem de programação (SLONNEGER e KURTZ, 1995):

Sintaxe	A sintaxe de uma linguagem de programação refere-se à sua estrutura de escrita, sem considerar o significado das palavras. Ela é composta por um conjunto de regras, que valida a sequência de palavras, símbolos e/ou instruções que é utilizada durante a escrita de um programa.
Semântica	Trata-se da análise do significado das palavras, expressões, símbolos e instruções da linguagem. A semântica é importante para que os desenvolvedores saibam precisamente o que as instruções de uma linguagem fazem.

[14] Pseudocódigo é uma forma de representação de algoritmos, escrito em português ou em outro idioma, que, posteriormente, podem ser transcritos para uma linguagem de programação.

Neste capítulo, iremos discutir apenas aspectos da sintaxe do Python. A semântica (significado das palavras e dos símbolos), serão apresentados detalhadamente nos capítulos e seções seguintes.

Para iniciarmos a discussão, considere o código[15] abaixo. Para uma melhor ilustração, exclusivamente neste exemplo, as ocorrências de quebras de linha (↲) e de espaços em branco no início das linhas (␣) foram sinalizadas.

```
>>>  # define o valor do limiar↲
...  limiar = 5↲
...  ↲
...  menores = [] # cria lista menores↲
...  maiores = [] # cria lista maiores↲
...  ↲
...  # divide os números de 1 a 10 em maiores e menores↲
...  for i in range(10): ↲
...  ␣␣␣␣if (i < limiar):↲
...  ␣␣␣␣␣␣␣␣menores.append(i)↲
...  ␣␣␣␣elif (i > limiar):↲
...  ␣␣␣␣␣␣␣␣maiores.append(i)↲
...  ↲
...  # imprime na tela os valores das listas↲
...  print('Resultado final')↲
...  print('menores:', menores)↲
...  print('maiores:', maiores)↲
Resultado final
menores: [0, 1, 2, 3, 4]
maiores: [6, 7, 8, 9]
```

Apesar de simples, o código abaixo nos permite ilustrar, de forma sucinta, diferentes aspectos importantes da sintaxe do Python. São estes aspectos que iremos apresentar a seguir.

Comentários

O código inicia com um comentário.

```
# define o valor do limiar↲
```

[15] Exemplo de código adaptado de (VANDERPLAS, 2016).

Comentários são trechos de códigos que são ignorados pelo interpretador da linguagem. O uso de comentários auxilia na documentação do programa, pois fornece algumas informações e/ou orientações sobre o seu funcionamento. Imagine você, escrevendo um código complexo hoje e tendo que entendê-lo daqui cinco anos? Tenho certeza de que o seu "eu" do futuro irá te agradecer se você documentar este código!

Em Python os comentários, são identificados por um sinal de cerquilha (#), e qualquer conteúdo após este símbolo é ignorado pelo interpretador. Isto significa que é possível escrevermos comentários de uma linha inteira ou após uma instrução válida de código, como por exemplo:

```
menores = [] # cria lista menores
maiores = [] # cria lista maiores
```

Em geral, é recomendável escrevermos comentários enquanto estivermos desenvolvendo ou atualizando o nosso código, pois, desta forma, o conteúdo a ser documentado será facilmente lembrado.

Quebras de Linhas

A primeira linha que o interpretador irá considerar no código é:

```
limiar = 5
```

Esta é uma operação de atribuição de valor à uma variável (iremos aprender mais sobre variáveis na Seção Declaração e Atribuição de Variáveis), onde é criada uma variável com o nome `limiar` e em seguida é atribuído a ela o valor `5`. Observe que esta instrução é finalizada com uma quebra de linha (↵) somente. Isso contrasta com linguagens como C++ e *Java*, onde cada instrução é obrigatoriamente terminada com um ponto e vírgula (;).

A seguir temos o bloco de código principal do exemplo:

```
# divide os números de 1 a 10 em maiores e menores
for i in range(10):
    if (i < limiar):
        menores.append(i)
    elif (i > limiar):
        maiores.append(i)
```

Esta é uma instrução de fluxo de controle composta, incluindo um *loop* de repetição (for … in) e uma instrução condicional (if – elif). Estas instruções serão apresentadas detalhadamente no Capítulo 4, mas, por enquanto, considere-as para a demonstração da diferença mais significante da sintaxe do Python em relação à outras linguagens: a indentação!

A indentação é uma forma de organizar o código em blocos, de forma que determinados trechos fiquem mais afastados, à medida que são inseridos espaçamentos no início de suas linhas. Para a maioria das linguagens de programação, a indentação não é obrigatória, mas são utilizadas para melhorar a legibilidade do código.

Entretanto, em Python, os blocos são identificados por meio da indentação e, por isso, a indentação é uma característica muito importante da linguagem. Blocos indentados são sempre precedidos por uma linha terminada com dois pontos (:), conforme o exemplo:

```
for i in range(10):
  #indentação indicando bloco
  in  (i < limiar):
    #indentação indicando um outro novo bloco
    menores.append(i)
```

É importante ressaltar que a quantidade de espaços em branco utilizada para indentar os blocos é uma escolha livre do próprio desenvolvedor, desde que seja

consistente em todo o código. Ou seja, devemos utilizar a mesma quantidade de espaços em branco para indentar todos os blocos em nosso código. Por convenção, a maioria dos especialistas recomendam utilizar quatro espaços para realizar a indentação, e, portanto, adotaremos esta convenção em todo o nosso curso.

O uso de indentação ajuda a aumentar a legibilidade e uniformidade do código, o que muitos desenvolvedores acham atraente no Python. Entretanto, é necessário ter atenção, pois, se a indentação não for realizada de forma adequada, o programa poderá se comportar de maneira inesperada ou até mesmo apresentar erros de sintaxe.

Por exemplo, os dois trechos de código a seguir estão corretos, mas apresentarão resultados diferentes:

```
>>>  # código 01
...  x = 4
...  if x > 5:
...      x = x * 2
...      print(x)
...  print('fim')

fim
```

```
>>>  # código 02
...  x = 4
...  if x > 5:
...      x = x * 2
...  print(x)
...  print('fim')

4
fim
```

No código da esquerda (código 01) a instrução print(x) está dentro do bloco indentado, e será executada somente se o valor de x for maior que 5. No da direita (código 02) a instrução print(x) está fora do bloco, fazendo com que seja executada independentemente do valor de x.

Espaços em Branco Inseridos no Meio da Linha

Embora a quantidade de espaços utilizados no início da linha faz a diferença na estrutura do código (indentação), os espaços inseridos dentro das linhas não importam. Por exemplo, todas as três expressões a seguir são equivalentes:

```
limiar = 5
limiar=5
limiar     =       5
```

Utilização de Parênteses

Em Python, o emprego de parênteses, "(" e ")" tem algumas funcionalidades próprias, dependendo do contexto em que são utilizados. Por hora, vamos conhecer as duas maneiras mais comuns de se utilizar os parênteses na linguagem. A primeira, é a utilização típica como uma forma de agrupar e estabelecer a precedência em operações matemáticas, como por exemplo:

```
x = 2 * (3 + 4)
y = (5 - 2) * (3 + 4)
z = (x + y) * 3
a = (x + y) / z
```

Os parênteses também podem ser utilizados para indicar que uma função deverá ser chamada[16]. Por exemplo, no código de exemplo no início desta seção, a função `print` foi chamada para exibir a frase "Resultado final" na tela. A chamada de funções é iniciada por um parêntese de abertura, seguido pelos argumentos[9] da função e finalizada por um parêntese de fechamento, conforme os exemplos a seguir:

```
print('Resultado final')
print('menores:', menores)
print('maiores:', maiores)
```

Algumas funções podem ser chamadas sem nenhum argumento. Neste caso, os parênteses de abertura e fechamento ainda devem ser utilizados. Um exemplo é a própria função `print`, que se chamada sem nenhum argumento, irá exibir apenas uma linha em branco:

```
>>>    print('linha 1')
```

[16] Veremos os detalhes sobre funções e argumentos no Capítulo 6.

```
...     print()
...     print('linha 2')
linha 1

linha 2
```

Variáveis e Keywords

As variáveis fazem parte dos recursos mais básicos e importantes de uma linguagem de programação. Elas são responsáveis por armazenar valores em memória, possibilitando a escrita e leitura dos dados com facilidade, a partir um nome definido pelo desenvolvedor.

A seguir iremos aprender como declarar e atribuir valores às variáveis do nosso código, assim como as regras que deverão ser seguidas no momento que formos escolher os nomes para as nossas variáveis.

Declaração e Atribuição de Variáveis

A declaração (criação) das variáveis é realizada por meio de um comando de atribuição, que define o seu valor, com a seguinte sintaxe:

```
variavel = valor
```

Primeiro é declarado o nome da variável (existem regras para os nomes que serão apresentadas a seguir), em seguida é utilizado o operador de atribuição (=) e por fim é atribuído o valor a ser armazenado pela variável. O Python permite manipular variáveis de diferentes tipos de dados, sejam eles numéricos, binários, literais (palavras), estruturas ou qualquer outro tipo suportado. É possível atribuir o valor diretamente à variável ou por meio de uma expressão, como os exemplos abaixo:

```
#atribuição por meio da declaração de um valor
nome = 'João'
```

```
var_booleano = True
raio = 2
pi = 3.14159265359

#atribuição por meio de uma expressão
area = pi * (raio * raio)
nome_completo = nome + 'da Silva'
media = (10 + 15 + 20 + 25)/4
```

Observe que não foi necessário realizar uma declaração explícita do tipo ao qual pertence cada variável, apenas foi declarado o valor (ou a expressão que resultará em um valor). Esta característica faz com o que o Python seja uma linguagem *dinamicamente tipada*, onde os tipos de variáveis são inferidos durante a execução do programa.

Por exemplo, no trecho de código anterior, a variável `var_booleano` será do tipo booleano, por receber `True` como valor atribuído e a variável `media` será do tipo numérica por receber o resultado de uma expressão numérica. Iremos aprender mais sobre os tipos de dados e as expressões no Capítulo 3.

Regras para Nomeação de Variáveis

Ao desenvolver o programa, é importante escolhermos nomes significativos e intuitivos para as nossas variáveis. Para tanto, existe um conjunto de regras básicas que devem ser seguidos na hora da escolha do nome:

- Uma variável pode conter letras e números, mas não pode começar com um número. Portanto, `variavel1` e `var1avel` são nomes válidos, enquanto `1variavel` é inválido.

- É permitido utilizar letras minúsculas e maiúsculas para nomes de variáveis. Entretanto, é recomendável utilizar apenas letras minúsculas. É importante ressaltar que Python é *case sensitive*, ou seja letras maiúsculas e minúsculas, são diferentes para o interpretador. Assim `nome` e `Nome` são duas variáveis diferentes, pois uma começa com letra minúscula e a outra começa com uma letra maiúscula.

- Não é permitido utilizar caracteres especiais (`@ # $ % * ^ ' " [ ] ( ) { }`), espaço em branco e pontuação (`. ! : ? ;`). Portanto, nomes como `valor em R$`, `orçamento(%)`, `pessoa:)` são inválidos.

- Versões mais recentes do Python permitem a utilização de caracteres acentuados (ex.: `é ê ã à ç`). Entretanto, é extremamente recomendável não utilizar estes caracteres.

- O único símbolo permitido é o caractere *underscore* (_) (conhecido em português por sublinhado). Geralmente utilizamos este símbolo para representar espaços em branco nas variáveis com nomes compostos, por exemplo: `nome_completo` e `maior_numero`. Ele também pode ser utilizado tanto no início (`_variavel`), quanto no final (`variavel_`.).

Por fim, existe também um conjunto de palavras que são reservadas do Python e por isso não podemos utilizá-las como nomes de variáveis. Este conjunto é chamado de *keywords* (palavras-chaves) e são palavras que já possuem alguma função pré-estabelecida na linguagem. É importante ressaltar que *keywords* também são *case sensitive* (diferencia letra maiúscula de minúscula). A seguir são apresentadas as principais *keywords* do Python.

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

Tipos de Erros do Python

Quando estamos desenvolvendo ou executando o nosso código, podemos deparar com erros que são causados por violações das regras de escrita ou por inconsistência durante as operações realizadas. Em Python, existem três tipos básicos de erros com os quais é necessário nos preocuparmos: *erros de sintaxe*, *erros em tempo de execução* e *erros lógicos*.

Erros de Sintaxe

Como mencionado anteriormente, sintaxe é o conjunto de regras que regem uma linguagem. Na linguagem escrita e falada dos humanos, as regras podem ser flexibilizadas ou até mesmo quebradas. No entanto, em uma linguagem de programação as regras são rígidas e devem ser seguidas conforme a especificação.

Um *erro de sintaxe* ocorre quando o desenvolvedor escreve uma instrução utilizando a sintaxe incorretamente e o interpretador não consegue compreender o código. Ao tentar executar uma instrução sintaticamente incorreta, o interpretador Python retornará um erro, indicando a linha onde há alguma inconsistência. Por exemplo, no código a seguir, não foram declarados os parênteses da função print. Portanto, ao executar este código, o interpretador retornará um erro com a seguinte mensagem:

```
>>>   x = 3
...   y = 5
...   print x + y

File "<ipython-input-6-04df4e714b51>", line 3
  print  "Esse é um tipo de erro"
       ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(x + y)?
```

Apesar do erro acontecer durante a execução do programa, existem ferramentas de desenvolvimento (IDE's) que conseguem acusar este tipo de erro durante o processo de escrita do código. Nestes casos, geralmente, as ferramentas sublinham ou destacam o trecho com o erro de sintaxe (semelhante ao que um processador de texto faz com os erros gramaticais):

```
x = 3
1var = x
y = 5
print x + y
```

Erros em Tempo de Execução

Os *erros em tempo de execução*, também conhecidos como *runtime errors*, surgem quando o Python não consegue executar a instrução devido à alguma inconsistência. Como o Python é uma linguagem interpretada, estes erros não ocorrerão até que o fluxo de controle em seu programa atinja a linha com o problema. Um exemplo comum de erro de tempo de execução é utilizar uma variável indefinida ou declarar incorretamente o nome da variável:

```
>>>   # utilização de uma variável inexistente (não definida)
...   x = 0
...   print(xx)

NameError                 Traceback (most recent call last)
<ipython-input-3-eab9121ba877> in <module>()
     1 # utilização de uma variável não existente
     2 x = 0
----> 3 print(xx)

NameError: name 'xx' is not defined

>>>   # declaração incorreta da variável dia, como Dia
```

```
...     dia = 'Segunda Feira'
...     print(Dia)

NameError                          Traceback (most recent call last)
<ipython-input-2-2db379d425a4> in <module>()
    1 # declaração incorreta da variável dia, como Dia
    2 dia = 'Segunda Feira'
----> 3 print(Dia)

NameError: name 'Dia' is not defined
```

Outro exemplo de erro em tempo de execução, é a divisão de um número ou variável por zero, pois esta é uma operação não definida e, consequentemente, não permitida:

```
>>>     x = 0
...     25 / x

ZeroDivisionError                  Traceback (most recent call last)
<ipython-input-4-b619e9ea33a1> in <module>()
    1 x = 0
----> 2 25 / x
ZeroDivisionError: division by zero
```

Erros Lógicos

O tipo de erro mais difícil de lidar em Python é o chamado *erro lógico*. Por exemplo, considere que um desenvolvedor deseja calcular a média entre três variáveis (`x`, `y` e `z`), com os seus respectivos valores (`x=2.0`, `y=7.0`, `z=12.0`). Para tanto ele escreve o código abaixo:

```
>>>     x = 5.0
...     y = 7.0
...     z = 12.0
...     media = x + y + z / 3
...     print(media)

16.0
```

Observe que, por não ter nenhuma regra violada, o interpretador foi capaz de executar todas as instruções e retornar `16.0`, que por usa vez não é a média dos números `5.0`, `7.0` e `12.0`. **A resposta correta deveria ser `8.0`**. Este erro é devido à ordem de

execução das operações. Primeiramente, o valor de `z` foi dividido por três, resultando em `4.0` e em seguida somado com x e y: `media = 5.0 + 7.0 + 4.0 => 16.0`. O correto seria indicar a precedência das somas, utilizando parênteses, e em seguida dividir por `3`.

```
>>>  x = 5.0
...  y = 7.0
...  z = 12.0
...  media = (x + y + z) / 3
     print(media)

8.0
```

Portanto, erros de lógicas são causados por um mal planejamento e especificação do *software* durante a etapa de desenvolvimento e estes erros não serão acusados pelo interpretador como um erro da linguagem.

Erros como este podem custar muito caro. Imagine, por exemplo, se um *software* de um banco calculasse de forma errada o somatório de todos os depósitos de um cliente? Felizmente, para evitar comportamentos inesperados, existem técnicas de teste de *software* que devem ser aplicadas em projetos do mundo real e que irão identificar estes tipos de erro antes do programa ser utilizado para valer.

Capítulo 3. Tipos Primitivos de Dados e Operadores

Durante a execução de um programa, o computador processa informações de diferentes tipos, como: números inteiros, números racionais, palavras, textos e outras estruturas mais complexas. Para categorizar a natureza destas informações, utilizamos uma determinada classificação, chamada de *tipos de dados*. Assim, todo valor que um programa manipular será, por definição, de um *tipo de dado* (ou *data type*, em inglês).

O tipo de dado também é responsável por definir quais operações podemos realizar com aquele dado em específico. Por exemplo, podemos multiplicar números entre si, mas não podemos multiplicar uma palavra com outra. Da mesma forma, conseguimos juntar duas palavras (concatenar), mas esta operação não é definida para os tipos de dados numéricos.

As linguagens de programação, normalmente, possuem alguns tipos de dados predefinidos, chamados de *tipos primitivos de dados*. Elas também possuem um conjunto de *operadores* básicos, de forma semelhante à matemática, que possibilita operações simples com estes dados, como: adições, multiplicações, comparações e atribuições. Os operadores podem ser interpretados como o conjunto mais básico de instruções simples que uma linguagem pode executar.

Enquanto, no capítulo anterior, aprendemos a escrever e estruturar um código utilizando a *sintaxe* do Python, neste capítulo iremos começar a entender a *semântica* da linguagem, ou seja, o significado de suas instruções e declarações. Iniciaremos com os tipos de dados primitivos do Python, explorando as suas particularidades e os operadores que manipulam esses tipos primitivos. Conhecer os tipos de dados, em conjunto com os operadores, é essencial para qualquer aspirante à desenvolvedor de *software*, pois são eles que possibilitam a execução de tarefas mais complexas.

O Python oferece quatro tipos primitivos de dados (`int`, `float`, `boolean`, `string`), que definem o tipo mais genérico possível da informação que um valor carrega. Estes tipos primitivos serão apresentados a seguir:

int
Números Inteiros
	O *int* (inteiro) é um tipo composto por caracteres numéricos (algarismos), utilizado para representar um número que pode ser escrito sem um componente decimal, podendo ter ou não sinal, isto é, ser positivo ou negativo. Os valores do tipo inteiro podem ter um tamanho ilimitado.
	`x = 3`
	`x = -3`
	`y = -52334235230985082582458245809243860924338502474598249867234 8`

float
Números Racionais
	O *float* (flutuante) também é composto por caracteres numéricos, mas é utilizado para representar números racionais (números que podem ser escritos como frações), também conhecidos como *ponto flutuante*[17]. Assim como os inteiros, podem ter sinal ou não e possuem um tamanho ilimitado. As casas decimais são separadas da porção inteira por um ponto (.).
	`x = 3.5`
	`x = -3.583344082`
	`pi = 3.14159265358979323846264338327950288419716939937510582097749`

bool
Booleanos
	O *bool* é um tipo de dado lógico que pode assumir apenas dois valores: *falso* ou *verdadeiro* (`False` ou `True`, respectivamente, em Python). Na lógica computacional, podem ser considerados como *0* (*falso*) ou *1* (*verdadeiro*). É um tipo de dado muito útil para a verificação de condições, conforme será apresentado no Capítulo 4.
	`x = True`
	`y = False`

str
Sequência de Caracteres
	Por fim, o *str* (abreviação de *string*, que em português podemos traduzir como *cadeia de caracteres*) é uma sequência/cadeia de caracteres utilizada para representar palavras, frases ou textos. As *strings*[18] podem ser declaradas de diferentes formas, por meio de aspas simples (`'`), aspas duplas (`"`) ou tripla de aspas (`'''`) – três aspas simples consecutivas.
	`x = 'aspas simples'`
	`y = "aspas duplas"`
	`z = '''aspas simples'''`

Diferentemente das regras para nomes de variáveis que vimos no capítulo anterior, podemos utilizar qualquer caractere *Unicode*[19] para os valores de uma *string*. Isso significa que podemos ter *strings* com caracteres japoneses (*kanji*), caracteres coreanos (*hangul*) e, principalmente, os caracteres especiais que utilizamos na língua portuguesa.

```
exemplo_kanji = 'キャラクター例'
exemplo_hangul = '문자 예'
exemplo_português = 'programação'
```

Como mencionado anteriormente, o tipo *string* pode ser declarado de três formas diferentes (dependendo da quantidade e tipo de aspas que se utiliza). Apesar de semelhantes, cada declaração tem uma particularidade que veremos a seguir. As aspas simples permitem que seja declarada uma sequência que contenha aspas duplas dentro dela. Por sua vez, as aspas duplas permitem a declaração de uma sequência com aspas simples, por exemplo:

```
>>>  x = 'Este é um valor de string com "aspas" duplas dentro dela.'
...  print(x)
Este é um valor de string com "aspas" duplas dentro dela.
>>>  y = "Agora um valor de string com 'aspas' simples."
...  print(y)
Agora um valor de string com 'aspas' simples.
```

[17] Em termos gerais, Ponto Flutuante, do inglês *floating point*, é um formato de representação dos números racionais, que é utilizada na lógica digital dos computadores (OVERTON, 2001).

[18] Por uma questão de padronização, utilizaremos o termo *string* para se referir à uma sequência de caracteres, pois é o termo utilizado por desenvolvedores do mundo todo, independente do idioma.

[19] O alfabeto Unicode possuí mais de 1 milhão de caracteres e é um único conjunto de caracteres contendo os idiomas de todo o mundo (https://home.unicode.org/).

Por fim, as triplas de aspas permitem a declaração de sequências com várias linhas, conforme o exemplo abaixo. Todos os espaços em branco declarados, assim como as quebras de linha, serão inclusos no valor da *string*.

```
>>>   z = '''Este é um valor      de string
...     declarado utilizando
...         uma tripla de aspas que inclusive pode conter
...       aspas 'simples' e "duplas" dentro dela.'''
...     print(z)

Este é um valor      de string
declarado utilizando
    uma tripla de aspas que inclusive pode conter
  aspas 'simples' e "duplas" dentro dela.
```

Por padronização, a maioria dos desenvolvedores sempre utilizam a declaração com aspas simples, por serem mais fáceis de escrever e deixam o código mais legível. Sendo utilizada as aspas duplas e triplas, quando as particularidades apresentadas anteriormente são necessárias. Por este motivo, adotaremos o padrão de aspas simples durante o curso.

O Tipo Especial NoneType

Para finalizar, aprendemos que, em Python, existem quatro tipos de dados primitivos principais que podem ser utilizados para representarmos números (`int` e `float`), valores booleanos (`bool`) e sequências de caracteres (`str`). Entretanto, existe também um tipo de dados especial para representar um valor nulo:

NoneType
Valores Nulos
	Este tipo de dados especial é utilizado para representar um valor nulo. Em algumas situações, pode ser necessário termos uma variável sem nenhum valor ou ainda indefinida. É para esses casos que utilizaremos tipo NoneType. Em Python, o valor nulo é representado pela palavra-chave None, conforme o exemplo:
	`x = None`
	`y = None`

É importante ressaltar que o valor nulo é diferente de zero, que é o elemento nulo na matemática, pois zero continua sendo um valor numérico não-nulo. Também é diferente de uma *string* vazia, que por sua vez é uma *string* não-nula, porém sem nenhum caractere.

Talvez seja um pouco difícil para o novato em Python entender o conceito de um valor nulo. Por isso, nos próximos capítulos, serão apresentadas algumas situações e exemplos que nos ajudarão a entender um pouco mais sobre a importância do valor nulo.

A função type()

O Python possuí diferentes tipos de dados que classificam a categoria da informação que está sendo manipulada no código. Entretanto, podem existir situações em que precisaremos descobrir qual é o tipo de um determinado valor. Para tanto, o Python oferece um método especial que permite retornar qual o tipo de dado de um valor ou uma variável, como ilustra os exemplos abaixo:

```
>>>   x = 10
...   print(type(x))
<class 'int'>
>>>   y = 'palavra'
...   print(type(y))
<class 'str'>
>>>   print(type(3.59))
```

```
<class 'float'>
>>>  print(type(True))
<class 'bool'>
>>>  z = None
...  print(type(z))
<class 'NoneType'>
```

Esta função é importante para caso seja necessário verificar o tipo de um dado em específico. Nas próximas seções utilizaremos o método `type()` para ilustrar alguns conceitos.

Operadores dos Tipos Primitivos de Dados

Conforme introduzimos, os *operadores* são símbolos especiais que definem as operações que podemos realizar com os tipos de dados primitivos do Python. Por sua vez, os valores que os operadores utilizam na computação são chamados de *operandos* e são os tipos de dados destes operandos que irão definir quais operações poderão ser realizadas entre eles, assim como o tipo de dado do valor resultante.

Existem diferentes operadores em Python e os principais podem ser classificados em três categorias: *operadores aritméticos*, *operadores de comparação* e *operadores lógicos*. Existem também as *operações com strings*, que por terem métodos e comportamentos exclusivos, operam de uma maneira distinta dos demais operadores.

A seguir apresentaremos os principais operadores do Python, assim como suas particularidades e exemplos.

Operadores Aritméticos

O Python possuí sete operadores aritméticos que são utilizados para realizar operações matemáticas como adição, subtração e multiplicação. São eles:

Operação		Descrição	Utilização
+	*Adição*	Adiciona os valores dos dois operandos.	x + y
-	*Subtração*	Subtrai do operando esquerdo o valor do operando direito.	x - y
*	*Multiplicação*	Multiplica os valores dos dois operandos.	x * y
/	*Divisão*	Divide o operando esquerdo pelo valor operando da direita. O resultado será sempre um número racional (float).	x / y
%	*Módulo*	Módulo (resto) da divisão do operando esquerdo pelo da direita. O resultado será um número inteiro (int).	x % y
//	*Divisão Inteira*	Divisão inteira do operando da esquerda pelo operando da direita. Retorna apenas a parte inteira do resultado da divisão tradicional. O resultado será sempre um número inteiro (int).	x // y
**	*Exponenciação*	Realiza a exponenciação (potenciação) do operando da esquerda, elevando-o a potência do operando da direita.	x ** y

Enquanto para alguns operadores o tipo de dado do resultado é fixo (/, % e //), para outros o resultado dependerá do tipo dos operandos (+, -, * e **). Por exemplo, o resultado da adição (+) de dois operandos do tipo inteiro (int) é um outro inteiro. Entretanto, se um dos operandos for do tipo racional (float), o resultado também será do tipo float:

```
>>>  print(type(10 + 5))
<class 'int'>
>>>  print(type(10 + 3.5))
<class 'float'>
```

Uma vez que o tipo bool assume os valores True ou False, que matematicamente são representados por 1 e 0 (respectivamente), os operadores aritméticos também podem ser aplicados ao tipo booleano:

```
>>>  print(10 + False) # Equivale a 10 + 0
...  print(True + 5) # Equivale a 1 + 5
...  print(True + True + False) # Equivale a 1 + 1 + 0

10
6
2
```

A seguir, no exemplo abaixo, podemos observar mais alguns exemplos de operações aritméticas:

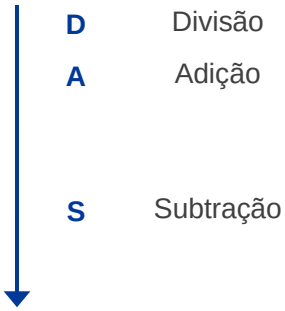
```
>>>  x = 10
...  y = 4
...  z = 4.5
...  print('x + y + z=', x + y + z)
...  print('x - 3 =', x - 3)
...  print('x * z =', x * z)
...  print('x * 2 =', x * 2)
...  print('x / (y + 2) =', x / (y + 2))
...  print('x % y =', x % y)
...  print('x // y =', x // y)
...  print('x ** z =', x ** z)

x + y + z = 18.5
x - 3 = 7
x * z = 45.0
x * 2 = 20
x / (y + 2) = 1.666666666666
x % y = 2
x // y = 2
x ** z = 31622.776601683792
```

O Python possuí regras de precedência que regem a ordem de execução das operações dento de uma expressão aritmética. É muito importante para o desenvolvedor ter ciência destas regras, pois a utilização incorreta da ordem de precedência dos operadores pode gerar resultados inesperados.

O Python segue a mesma convenção tradicional da matemática. Essa regra é conhecida com *PEMDAS*, e obedece a seguinte ordem:

P	Parênteses
E	Exponenciação
M	Multiplicação

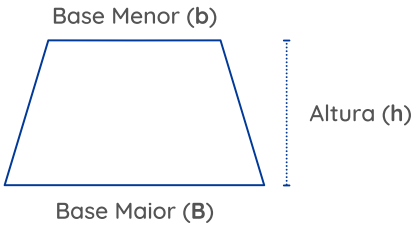


Vejamos um exemplo em que a utilização incorreta da ordem de precedência das operações pode gerar resultados inesperados. Considere o problema de calcular a área de um trapézio, que é dada pela equação abaixo. Para calcularmos esta área é necessário somarmos os valores da *base maior* (*B*) com a *base menor* (*b*), em seguida, multiplicarmos o resultado pelo valor da *altura* (*h*) e, por fim, dividirmos tudo por *2*.

$$area = \frac{(B + b) \times h}{2}$$

Para $B = 15$, $b = 10$ e $h = 8$ temos:

$$area = \frac{(15 + 10) \times 8}{2} = 100$$



Um desenvolvedor despercebido, e que não conhece as regras de precedência do Python, poderia escrever o seguinte código para o cálculo. Observe que o resultado encontrado (55.0) não corresponde ao valor esperado para $B = 15$, $b = 10$ e $h = 8$. Conforme o exemplo da figura, **o valor correto seria** 100.0**!**

```
>>>  B = 15 # Base maior (B)
...  b = 10 # Base menor (b)
...  h = 8 # Altura (h)
...
...  # Calcula a área do Trapézio
...  area = B + b * h / 2
...  print('Área do Trapézio: ', area)
Área do Trapézio: 55.0
```

O erro foi causado pela ordem em que as operações foram realizadas, que, segundo a regra apresentada anteriormente (*PEMDAS*), foi a seguinte:

1. Com os valores de exemplo, temos a equação inicial: `15 + 10 * 8 / 2`;
2. Pela ordem de precedência do Python, primeiramente foi resolvido a operação de multiplicação (`10 * 8`), restando a equação: `15 + 80 / 2`;
3. Segundo a regra, a precedência é da divisão (`80 / 2`), restando apenas `15 + 40`;
4. Por fim, a adição (`15 + 40`) será realizada, resultando no valor de `55`.

Portanto, para corrigirmos o código, é necessário priorizar a adição `B + b`, por meio da declaração de parênteses:

```
>>>   area = (B + b) * h / 2
...   print('Área do Trapézio: ', area)

Área do Trapézio: 100.0
```

Operadores de Comparação

Essa categoria de operadores é utilizada para comparar valores entre si. Eles também são chamados de operadores relacionais e sempre retornaram um resultado do tipo booleano (`bool`), com o valor `True` ou `False`, de acordo com a condição avaliada.

Operação		Descrição	Utilização
>	*Maior*	Verdadeiro (True) se o operando da esquerda for **maior que** o operando da direita.	x > y
>=	*Maior ou igual*	Verdadeiro (True) se o operando da esquerda for **maior ou igual que** o operando da direita.	x >= y
<	*Menor*	Verdadeiro (True) se o operando da esquerda for **menor que** o operando da direita.	x < y
<=	*Menor ou igual*	Verdadeiro (True) se o operando da esquerda for **menor ou igual** que o operando da direita.	x <= y

==	*Igual*	Verdadeiro (True) se os dois operandos são iguais (possuem o mesmo valor).	x == y
!=	*Diferente*	Verdadeiro (True) se os dois operandos são diferentes (não possuem o mesmo valor).	x != y

Estes operadores podem ser utilizados para todos os tipos primitivos de dados do Python, inclusive em conjunto com os operadores aritméticos, como podemos ver nos exemplos a seguir:

```
>>>  x = 5
...  y = 1
...  s = 'palavra'
...
...  print('x > y:', x > (y + 2))
...  print('x <= 4.564:', x <= (4.564 + 1))
...  print('s == palavra:', s != 'palavra')
...  print('(y * 0) == False:', (y * 0) == False)
...  print('s != abacaxi:', s != 'abacaxi')

x > y: True
x <= 4.564: True
s == palavra: False
(y * 0) == False: True
s != abacaxi: True
```

Apesar de ser possível realizar comparações entre tipos de dados distintos, as operações de comparação com *strings* (`str`) merecem uma atenção especial. Primeiramente, *strings* somente podem ser comparadas com outros tipos de dados por meio dos operadores de igualdade (`==`) e desigualdade (`!=`), se elas forem comparadas com outro tipo de dados utilizando outros operadores (`>`, `>=`, `<`, `<=`), um erro de execução ocorrerá, avisando que o operador não é suportado para o tipo `str`:

```
>>>  x = 3
...  y = 'palavra'
...  print(x == y)
...  print(x != y)
...  print(x > y)

False
True
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-19a87f04b970> in <module>()
```

```
    3 print(x == y)
    4 print(x != y)
----> 5 print(x > y)

TypeError: '>' not supported between instances of 'int' and 'str'
```

Outro ponto importante é quando os operadores de maior (>), maior ou igual (>=), menor (<) e menor ou igual (<=) são aplicados em operandos do tipo *string*, a verificação ocorrerá pela comparação lexicográfica entre as duas *strings*. Ou seja, sequências que, na ordem alfabética, antecedem outras, são consideradas menores. Por exemplo:

```
>>>  a = 'casa'
...  b = 'predio'
...  c = 'elevador'
...  d = 'casamento'
...  print(x < y)
...  print(x > z)
...  print(y > z)
...  print(a > d)
...  print('abacaxi' >= 'abacaxi')

True
False
True
False
False
```

Operadores Lógicos

Os *operadores lógicos*, como o próprio nome diz, são operadores que permitem realizar operações lógicas entre valores booleanos e o resultado será sempre verdadeiro (True) ou falso (False). Estes operadores são importantes para a formação de lógicas completas, em conjunto com outros operadores, que possibilitam a verificação de diferentes condições no programa. Fazem parte desta categoria os seguintes operadores lógicos:

	Operação	Descrição	Utilização
and	*Conjunção*	Verdadeiro (True) se, e somente se, os dois operandos também forem verdadeiros.	x and y
or	*Disjunção*	Verdadeiro (True) se qualquer um dos dois operandos for verdadeiro também.	x or y
not	*Complemento*	Inverte o valor do operando (complemento). Verdadeiro (True) se o operando for falso. Falso (False) se o operando for verdadeiro.	not x

Um bom desenvolvedor precisa dominar esses operadores, pois é a partir deles que conseguiremos criar diferentes tarefas em um programa. Por exemplo, poderia ser necessário escrever um programa que verifica se, para poder dirigir, uma pessoa é maior de idade **E** possuí carteira de habilitação. Outro exemplo seria verificar se um passageiro está autorizado a viajar sozinho, se ele é maior de idade **OU** se está acompanhado de um responsável. Veja o código abaixo para estas duas situações:

```
>>> # verifica se uma pessoa pode dirigir
idade = 23
possui_cnh = False
print(idade > 18 and possui_cnh)

# verifica se um passageiro pode viajar sozinho
idade = 15
possui_autorizacao = True
print(idade > 18 or possui_autorizacao)
False
True
```

Pode parecer difícil no início, mas, com a prática e o desenvolvimento de alguns exemplos, logo você estará acostumado a escrever lógicas mais complexas. Para auxiliar neste início, é sempre bom recorrer à uma referência (conhecida como *tabela verdade*) com os resultados dos operadores lógicos, para todos os possíveis operandos (***A*** e ***B***).

A	B	A AND B	A OR B	NOT A	NOT B
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

Abaixo são apresentados alguns exemplos de utilização dos operadores lógicos, que também podem ser combinados com outros operadores:

```
>>>  print(True or False)
...  print((10 > 1) and True)
...  print(not (True or False))
...  print(not ('palavra' == 'palavra'))

True
True
False
False
```

Operações com Strings

Como apresentado anteriormente, uma *string* (str) é uma sequência de caracteres, que são utilizadas para representar palavras, frases e textos, e elas possuem diferentes operações úteis aos desenvolvedores. Antes de prosseguirmos com a discussão sobre as operações do tipo *string*, é necessário entendermos como uma string é representada no Python.

Para tanto, vamos considerar a palavra **consolação**, que possuí 10 letras. A imagem a seguir ilustrar como esta *string* é representada na forma de uma sequência de 10 caracteres e podemos observar que cada caractere ocupa uma posição:

c	o	n	s	o	l	a	ç	ã	o

Podemos então enumerar cada uma destas posições, ou seja, podemos atribuir um índice para cada posição (caractere) da *string*. Esse processo é conhecido como

indexação e em Python, assim como em várias outras linguagens, os índices sempre começam a partir de zero[20]:

c	o	n	S	O	l	a	ç	ã	o
0	1	2	3	4	5	6	7	8	9

Um recurso interessante do Python é que ele também permite uma indexação negativa, iniciando de -1, para realizar um acesso invertido (ou seja, de trás para frente). Assim, o índice -1 representa o último caractere, o -2 o penúltimo caractere, prosseguindo até o primeiro caractere da string, conforme ilustrado abaixo:

c	o	n	s	o	l	a	ç	ã	o
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Agora que foram introduzidos os conceitos de indexação e de índice, poderemos conhecer as principais operações que podem ser realizadas com as *strings*. Veremos desde a maneira como é realizada o acesso aos caracteres até transformações úteis, como substituir parte dos caracteres.

Operador de Acesso das *Strings*

As *strings* possuem um operador importante que permite acessar um caractere, indicando o índice desejado. Este operador tem a seguinte sintaxe: `[indice]`. Onde `indice` é o operando e representa a posição do caractere desejado. Abaixo são

[20] Por esse motivo, Python é conhecido como uma linguagem *0-indexed* ou *0-based*.

apresentados exemplos de uso do operador de acesso aos caracteres da *string* 'consolação':

```
>>>    palavra = 'consolação'
...
...    #indices positivos
...    print('Primeiro caractere:', palavra[0])
...    print('Segundo caractere:', palavra[1])
...    print('Sexto caractere:', palavra[5])
...    print('Último caractere:', palavra[9])
...
...    #indices negativos
...    print('Último caractere:', palavra[-1])
...    print('Penúltimo caractere:', palavra[-2])
...    print('Antepenultimo caractere:', palavra[-3])
...    print('Primeiro caractere:', palavra[-10])

Primeiro caractere: c
Segundo caractere: o
Sexto caractere: l
Último caractere: o

Último caractere: o
Penúltimo caractere: ã
Antepenultimo caractere: ç
Primeiro caractere: c
```

Também é possível acessarmos uma faixa de caracteres (ou seja, um subconjunto dos caracteres), indicando o índice inicial e final desejados. O resultado, será então, uma *sub-string* da *string* original. A sintaxe para esta operação é: [inicio:fim]. Esta notação (inicio:fim) é chamada de *slicing* (fatiamento, em português) e é um recurso muito importante do Python (ROMANO, 2018).

É importante ressaltar que, enquanto o caractere do índice inicio será incluso na *sub-string* resultante, o caractere do índice fim não será incluso. Em outras palavras, ao indicar [inicio:fim], será retornado a sub-string composta pelos caracteres dos índices inicio até fim-1. Meio confuso? Vamos ver alguns exemplos e logo vai ficar fácil de entender. Considere novamente a palavra **consolação**. Ela possui outras palavras menores embutidas dentro dela, como: *consola*, *sol*, *sola* e *ação*. Vamos então acessar

estas palavras menores utilizando o conceito de *slicing* e para um melhor entendimento, vamos também observar a figura que ilustra os índices da palavra:

```
>>>    palavra = 'consolação'
...
...    # consola
...    print(palavra[:7])
...
...    # sol
...    print(palavra[3:6])
...
...    # sola
...    print(palavra[3:7])
...
...    # ação
...    print(palavra[-4:])
...
...    # palavra original
...    print(palavra[:])
consola
sol
sola
ação
consolação
```

c	o	n	s	o	l	a	ç	ã	o
0	1	2	3	4	5	6	7	8	9
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Por exemplo, para a palavra sol indicamos que deveríamos acessar os caracteres da posição 3 até 5. Para tanto, especificamos o *slicing* [3:6], uma vez que o índice 6 não será incluso.

c	o	n	s	o	l	a	ç	ã	o
0	1	2	3	4	5	6	7	8	9

Observe que, para as palavras *consola* e *ação*, não foi indicado de forma explicita o inicio e o fim, respectivamente. Isso é possível, pois o Python identifica que deverá ser usado o primeiro índice (no caso de omitir o inicio) e o último índice (se fim

for omitido). Por fim, se os dois valores forem omitidos, a *string* original será retornada, conforme o último exemplo palavra[:];

Duas observações importantes sobre o operador de acesso das *strings* são: os operandos (índices) deverão sempre ser do tipo *inteiro* (int) e válidos (ou seja, a posição do índice deve existir). Se isso não ocorrer, um erro de execução será retornado:

```
>>>  palavra = 'consolação'
...  palavra[3.5]

...
TypeError: string indices must be integers
>>>  palavra[99]

...
IndexError: string index out of range
```

Operadores de *Strings*

Além dos operadores de comparação, que foram apresentados na Seção Operadores de Comparação, as strings possuem outros operadores úteis. Vamos começar pelos operadores de concatenação e repetição:

	Operação	Descrição	Utilização
+	*Concatenação*	Concatena (une) duas *strings* em uma única.	s1 + s2
*	*Repetição*	Repete o valor da *string* (operando da esquerda) *n* vezes, onde *n* é o valor do operando da direita.	s1 * n

Estes operadores são utilizados quando queremos formar novas *strings* a partir de outras já existentes. O operador %, quando utilizado com *strings*, realiza uma operação especial que é a formatação de *strings* e veremos mais detalhes na Seção Formatação de Strings. Os demais operadores (-, /, //, e **), que são implementados para os outros tipos de dados primitivos, não podem ser utilizados para as *strings* e ao tentar utilizá-los, irá ocorrer um erro de execução.

A seguir serão apresentados exemplos de utilização dos operadores de concatenação (+) e repetição (*) de *strings*:

```
>>>   s1 = 'Belo'
...   s2 = 'Horizonte'
...
...   # Concatenação (+)
...   print(s1 + s2)
...   print(s1 + ' ' + s2)
...   print(s1 + ' Monte')
...
...   # Repetição
...   print(s1 * 5)
...   print((s1 + ', ') * 4)
BeloHorizonte
Belo Horizonte
Belo Monte

BeloBeloBeloBeloBelo
Belo, Belo, Belo, Belo,
```

Além dos operadores de concatenação e repetição, as strings possuem operadores para verificar se uma *string* está contida ou não em uma outra *string*.

Operação		Descrição	Utilização
in	*Filiação (Membership)*	É utilizado para verificar se uma determinada *string* (operando da esquerda) está contido em outra *string* (operando da direita). O retorno será verdadeiro se a *string* s1 estiver contido na *string* s2.	s1 in s2
not in	*Não-Filiação*	Oposto do operador anterior. Retorna verdadeiro se a string s1 não estiver contida na *string* s2.	s1 not in s2

Vejamos alguns exemplos dos operadores in e not in:

```
>>>   s1 = 'consolação'
...   s2 = 'sola'
...
...   print(s1 in s2)
...   print(s2 in s1)
...   print('solar' in s1)
...   print('solar' not in s2)
```

```
False
True
True
True
```

Métodos das *Strings*

O Python oferece diversos métodos e funções nativas para a manipulação das *strings*. Dentre destes, talvez o que mais se destaque é a função len(string), que retorna o tamanho da *string*. Ou seja, a função retorna o número de caracteres da string:

```
>>>  palavra = 'consolação'
...  print(len(palavra))
10
```

A seguir serão apresentados, na forma de exemplos, outros métodos e funções úteis, nativas do Python. Uma referência completa de todas as funcionalidades pode ser encontrada na documentação oficial do Python (PSF, 2022).

```
>>>  s1 = 'Belo Horizonte'
...  s2 = 'Esta é uma frase, com uma vírgula.'
...
...  print(s1.upper()) # todas as letras maiúsculas
...  print(s1.lower()) # todas as letras minúsculas
...  print(s2.title()) # primeira letra de cada palavra em maísculo
...
...  print(s1.replace('Horizonte', 'Monte')) # substitui 'Horizonte' por 'Monte'
...
...  print(s1.startswith('Belo')) # verifica se a string começa com 'Belo'
...  print(s1.endswith('Monte')) # verifica se a string termina com 'Monte'
...  print(s2.find('frase')) # retorna o índice da primeira ocorrência da palavra frase
...
...  print(s2.split()) # particiona a string em outras, que são separadas por um espaço
...  print(s2.split(',')) # particiona a string em outras, que são separadas por ','
...
...  s3 = '    palavra com espaços    '
...  print(s3.strip()) # remove os espaços extras no início e fim da string
BELO HORIZONTE
belo horizonte
Esta É Uma Frase, Com Uma Vírgula.

Belo Monte

True
False
```

```
11

['Esta', 'é', 'uma', 'frase,', 'com', 'uma', 'vírgula.']
['Esta é uma frase', ' com uma vírgula.']

palavra com espaços
```

Os métodos das *strings* podem, também, ser utilizados diretamente na declaração do valor, e não somente com as variáveis do tipo *string*. Vejamos os exemplos abaixo que ilustram esta utilização:

```
>>>  print('nome'.upper())
...  print('a, b, c, d'.split(','))
...  print('Belo Horizonte'.replace('Horizonte', 'Monte'))

NOME
['a', 'b', 'c', 'd']
Belo Monte
```

Conversão e Formatação dos Tipos de Dados

Até agora vimos que os *tipos de dados* determinam quais operações e transformações podem ser realizadas com os dados em Python, assim como o comportamento destas operações. Entretanto, ao desenvolvermos um código, podemos deparar com situações em que seja necessário manipular estes tipos de dados de uma forma diferente daquelas em que foram convencionadas. Por exemplo, se tentarmos concatenar uma *string* com um número inteiro, teremos o seguinte erro de execução:

```
>>>  nascimento = 1986
...  ano_atual = 2022
...  idade = ano_atual - nascimento
     print('Eu tenho ' + idade + ' anos')

TypeError                          Traceback (most recent call last)
<ipython-input-57-0f581e55303b> in <module>()
    2 ano_atual = 2022
    3 idade = ano_atual - nascimento
----> 4 print('Eu tenho ' + idade + ' anos')

TypeError: can only concatenate str (not "int") to str
```

Isso ocorre porque a concatenação de *strings* com outros tipos de dados não é permitida. Felizmente, o Python oferece duas formas de lidarmos com esta situação: a primeira delas é a *conversão dos tipos dados* e a segunda é a *formatação de strings*. A seguir serão apresentados estes conceitos, assim como exemplos de utilização.

Conversão dos tipos de Dados

A *conversão dos tipos de dados* é um conceito que nos permite transformar um tipo de dado em outro, por meio de uma função de conversão. Por exemplo, a limitação anterior poderia ser facilmente contornada se convertêssemos a variável `idade` do tipo `int` para o tipo `str`, por meio da função `str(idade)`:

```
>>> nascimento = 1986
... ano_atual = 2022
... idade = ano_atual - nascimento
    print('Eu tenho ' + str(idade) + ' anos')

Eu tenho 36 anos
```

A tabela abaixo discute o comportamento das conversões possíveis entre os tipos de dados primitivos do Python e, a seguir, exemplos dos resultados das operações de conversões são apresentados.

CONVERSÃO			FUNÇÃO	COMPORTAMENTO
int	→	**float**	float(i)	O inteiro `i` será convertido para um número racional.
int	→	**bool**	bool(i)	`True` se o inteiro `i` for diferente de `0`, `False` se o contrário.
int	→	**str**	str(i)	O inteiro `i` será convertido para uma string, mantendo o mesmo formato do número.
float	→	**int**	int(f)	Apenas a parte inteira do número racional `f` será conservada.
float	→	**bool**	bool(f)	`True` se o racional `f` for diferente de `0.0`, `False` se o contrário.

float	→	**str**	str(f)	O número racional `f` será convertido para uma *string*, com uma precisão de 15 casas decimais.
bool	→	**int**	int(b)	Se o valor de `b` for `True` será retornado `1`. Se `False`, retorna `0`.
bool	→	**float**	float(b)	Se o valor de `b` for `True` será retornado `1.0`. Se `False`, retorna `0.0`.
bool	→	**str**	str(b)	Retorna as strings `'True'` ou `'False'`, de acordo com o valor de `b`.
str	→	**int**	int(s)	Converte a string `s` para um número inteiro, se possível.
str	→	**float**	float(s)	Converte a string `s` para um número racional, se possível.
str	→	**bool**	bool(s)	Retorna `False` se `s` for uma string vazia (`''`), ou caso contrário retorna `True`.

É importante ressaltar que a conversão de uma *string* para os tipos de dados numéricos (`int` ou `float`), pode resultar em um erro de execução. Por exemplo, a conversão de `'0.98'` para `int` resultará em um erro, pois `0.98` não é um número inteiro válido. Da mesma forma, não é possível converter a *string* `'consolação'` para um tipo numérico, pois, por motivos óbvios, *consolação* não é um número válido.

A seguir são apresentados alguns exemplos das funções de conversão de tipos:

```
>>>    #conversão do tipo int
...    print(float(10)) # 10.0
...    print(bool(-1)) # True
...    print(bool(0)) # False
...
...    #conversão do tipo float
...    print(int(9.999)) # 9
...    print(bool(-0.99)) # True
...    print(str(-0.99)) # '-0.99'
...
...    #conversão do tipo bool
...    print(int(True)) # 1
...    print(float(False)) # 0.0
...    print(str(True)) # 'True'
...
...    #conversão do tipo str
...    print(int('-99')) # 99
...    print(float('0.01')) # 0.00001
...    print(bool('palavra')) # True
...    print(bool('')) # False
10.0
True
False

9
True
-0.99

1
0.0
True

-99
0.01
True
False
```

Outro caso especial que precisamos entender são as operações de conversão com o tipo `NoneType`. Primeiramente, não existe funções de conversão dos outros tipos para `None`. Para tanto, basta simplesmente atribuirmos às variáveis o valor `None`:

```
>>>    x = 10
...    print('Valor:', x, '--', type(x))
...
...    # atribuição do valor None à variável x
...    x = None
...    print('Valor:', x, '--', type(x))

Valor: 10 -- <class: 'int'>
Valor: None -- <class: 'NoneType'>
```

Por sua vez, a conversão do valor None para outros tipos é possível apenas para os tipos str e bool. Enquanto a conversão de None para uma *string* resulta na *string* 'None', a conversão do valor None para o tipo bool sempre retornará False.

```
>>>    print(str(None))
...    print(bool(None))

'None'
True
```

Formatação de Strings

Uma outra forma de criarmos uma *string* a partir de diferentes tipos de dados é a utilização do conceito de formatação de *strings*. O Python possibilita três formas de formatarmos uma string:

- A utilização do operador %;
- O método str.format();
- O conceito de *f-strings*, que foi introduzido a partir da versão *3.6* do Python.

Por ser a maneira mais recomendada atualmente, enquanto as outras vão caindo em desuso pelos desenvolvedores, iremos focar apenas no conceito de *f-strings*. Apesar de não forcarmos na utilização do operador % e no método str.format(), é fortemente recomendado o estudo sobre estas técnicas em paralelo[21], pois, uma vez

[21] Recomendação de materiais sobre o operador % e o método str.format():

que o *f-strings* é recente, muitos códigos encontrados em repositórios, outros cursos e artigos ainda podem utilizar estes conceitos.

As *f-strings*, abreviação de *formatted string literals* (strings literais formatadas, em português), são identificadas pela letra "f" antes da declaração, no início, da string e um par de chaves são utilizados para interpolar os valores e expressões desejadas. Portanto, sua sintaxe é a seguinte:

```
f'Exemplo de f-string com um valor {valor} e uma expressão {expr}'
```

Lembram do exemplo nosso que gerou um erro quando tentamos concatenar *string* com um valor do tipo `int`? Utilizando *f-strings*, poderíamos contornar o problema da seguinte forma:

```
>>>  nascimento = 1986
...  ano_atual = 2022
...  idade = ano_atual - nascimento
...  print(f'Eu tenho {idade} anos')
Eu tenho 36 anos
```

Neste caso, o valor da variável `idade` foi utilizado no lugar da declaração `{idade}`. E não precisamos nos limitar apenas à utilização de variáveis, podemos utilizar também expressões que irão resultar em valores a serem interpolados na *string*. Poderíamos simplificar o código acima, suprimindo o uso da variável `idade`, da seguinte maneira:

```
>>>  nascimento = 1986
...  ano_atual = 2022
...  print(f'Eu tenho {ano_atual - nascimento} anos')
Eu tenho 36 anos
```

- https://realpython.com/python-f-strings/
- https://programadorviking.com.br/metodo-format-em-python/
- https://pyformat.info/

Também podemos utilizar métodos e funções dentro de *f-strings*, como por exemplo:

```
>>>  palavra = 'consolação'
...  print(f'A palavra {palavra.upper()} possuí {len(palavra)} letras')
A palavra CONSOLAÇÃO possuí 10 letras
```

Por fim, é possível realizar alguns tipos de formatações nos valores antes da interpolação na *string* resultante. Por exemplo, considere que queremos exibir a porcentagem gasta de um determinado orçamento:

```
>>>  orcamento = 5000
...  vlr_gasto = 430
...  pct = (vlr_gasto/orcamento) * 100
...  print(f'Porcentagem já gasta do orçamento: {pct}%')
Porcentagem já gasta do orçamento: 6.142857142857143%
```

Geralmente, quando tratamos de valores percentuais, não estamos interessados em mais do que duas casas decimais. Portanto, para atender esta convenção, podemos simplesmente utilizar a seguinte notação `{pct:.2f}`, onde foi declarado que o valor da variável `pct` é do tipo `float` (por isso o `f` no final) e que deverá ser utilizado apenas duas casas após o separador de decimais (`.2`).

```
>>>  orcamento = 5000
...  vlr_gasto = 430
...  pct = (vlr_gasto/orcamento) * 100
...  print(f'Porcentagem já gasta do orçamento: {pct:.2f}%')
Porcentagem já gasta do orçamento: 6.14%
```

Por fim, um outro exemplo de formatação. Observe que multiplicamos o valor da divisão (`vlr_gasto/orcamento`) por `100`, porque gostaríamos de um resultado em termos de porcentagem. Entretanto, com *f-strings* isso não é necessário, pois ela permite uma notação que permite declarar que o valor é uma porcentagem, e inclusive já adiciona o símbolo *%* no valor formatado:

```
>>>  orcamento = 5000
```

```
...     vlr_gasto = 430
...     pct = (vlr_gasto/orcamento)
...     print(f'Porcentagem já gasta do orçamento: {pct:.2%}')

Porcentagem já gasta do orçamento: 6.14%
```

O conceito de *f-strings* simplifica, e muito, a vida do desenvolvedor em Python e existem vários outros tipos de formatação e operações que podem ser realizadas. Uma relação completa pode ser encontrada na documentação oficial do Python (PSF, 2022), ou o artigo de (BRITO, 2020), em inglês, que contempla vários exemplos.

Capítulo 4. Fluxos de Controle em Python

Nos capítulos anteriores, foram apresentadas apenas instruções simples do Python que podem ser executadas em sequência, uma após a outra, como se fosse uma lista de tarefas. Entretanto, em diversas situações, apenas executar uma sequência de instruções não é suficiente para a criação de um programa funcional. Será necessário escrever códigos capazes de selecionar quais instruções devem ser executadas, ignoradas e/ou repetidas.

Uma das maiores vantagens do computador em relação ao cérebro humano é a sua capacidade de realizar tarefas repetitivas e de análise de valores em uma velocidade bem maior, como por exemplo:

- Calcular a média de um conjunto gigante de números.
- Indicar quais são os números primos entre 0 e um trilhão.
- Encontrar uma transação, entre as milhões de transações diárias, de um banco.
- Calcular o melhor caminho (menor distância) entre dois endereços.

Estes são apenas alguns exemplos, sendo que existem diversos outros problemas em que é necessário projetarmos soluções com tarefas de seleção e repetição, em que muitas das vezes estão condicionadas aos valores de um certo conjunto de dados.

As estruturas de *fluxos de controle* de código são as responsáveis por possibilitar a execução destas tarefas de seleção e repetição em uma linguagem de programação. De certa forma, será por meio delas que iremos especificar a ordem na qual o código do programa deverá ser executado. Em Python, o fluxo de controle é

realizado por meio de *estruturas condicionais*, *estruturas de repetição* e funções (MARTELLI, RAVENSCROFT e HOLDEN, 2017).

Enquanto as funções serão apresentadas em detalhes no Capítulo 6, neste capítulo iremos discutir as estruturas condicionais e as estruturas de repetição.

Estruturas Condicionais

Em diversas situações, ao invés de uma execução sequencial de instruções, é necessário executar um trecho de código específico com base em uma determinada condição. As *estruturas condicionais* são os recursos da linguagem que permitem realizar esta tarefa.

São estas estruturas de código que determinam quais instruções serão executadas a partir do momento em que uma condição é verdadeira, e quais serão executadas no caso contrário. Alguns exemplos em que podemos utilizar as estruturas condicionais: decidir se um número é par ou não; se a temperatura é alta o suficiente para ligar o ar-condicionado; decidir se a nota de um aluno foi suficiente para a aprovação ou não.

Em outras palavras, estas condições deverão ser verificadas com base nos dados de entrada e em seguida as ações (instruções) serão executadas de acordo com a condição ser verdadeira ou não. Em Python existem três tipos de estruturas condicionais, que veremos a seguir.

Estrutura Condicional **if**

A estrutura condicional `if` permite avaliar uma condição (expressão) e, se o seu resultado for verdadeiro, executar um determinado bloco de código. A sua sintaxe é a seguinte:

```
if <expressão>:
  <bloco de código se a expressão é verdadeira>
```

Capítulo 5. A <expressão> pode ser qualquer condição baseada em uma combinação de uma ou mais variáveis, operadores e até mesmo outras expressões. O seu valor será avaliado como verdadeiro (True) ou falso (False).

Capítulo 6. Se a <expressão> for avaliada como verdadeira, o <bloco de código> será executado, caso contrário, será ignorado.

Capítulo 7. Como discutido na seção Indentação, os dois pontos (:) após a <expressão> é obrigatório, assim como a indentação do <bloco de código>. Se estas regras não forem observadas, um erro de sintaxe irá ocorrer.

A seguir veremos alguns exemplos de aplicação da instrução if. No primeiro exemplo verificamos se a condição idade > 18 é verdadeira. Caso positivo, é exibida uma mensagem que a idade é suficiente para CNH.

```
>>>  # Exemplo em que a condição é verdadeira
...  idade = 35
...  if idade >= 18:
...      print('Idade suficiente para CNH!')
Idade suficiente para CNH!
```

Caso contrário, se a condição não é verdadeira, a mensagem não é exibida.

```
>>>  # Exemplo em que a condição é falsa
...  idade = 15
...  if idade >= 18:
...      print('Idade suficiente para CNH!')
```

Observe que a condição idade > 18 é formada por uma expressão que utiliza o operador condicional maior que (>=). Como mencionado anteriormente, é possível combinarmos diferentes expressões para criar uma condição:

```
>>>   # Verifica se uma pessoa está apta para se aposentar
...     idade = 70
...     tempo_contrib = 20
...     if idade >= 65 or tempo_contrib >= 35:
            print('Habilitado para solicitar aposentadoria!')

Habilitado para solicitar aposentadoria
```

Neste último exemplo utilizamos o operador lógico `or`, para verificar se pelo menos uma das expressões (`idade >= 65` ou `tempo_contrib >=35`) eram verdadeiras. Como a `idade` indicada é maior ou igual que 65, a condição é verdadeira e a mensagem foi exibida.

Podemos utilizar a imaginação (e consequentemente o uso correto da lógica) para criar vários tipos de estruturas condicionais. Por exemplo, o código anterior poderia ser modificado para exibir uma mensagem de "não habilitado", da seguinte forma:

```
>>>   # Verifica se uma pessoa não está apta a se aposentar
...     idade = 35
...     tempo_contrib = 20
...
...     if idade < 65 and tempo_contrib < 35:
            print('Não habilitado para solicitar aposentadoria!')

Não habilitado para solicitar aposentadoria
```

Estrutura Condicional **if – else**

Na seção anterior aprendemos sobre a instrução `if`, que é utilizada para executar um bloco de código caso uma condição seja verdadeira. No entanto, nenhum comportamento específico foi definido para o caso de a condição ser falsa. Para tanto, podemos fazer o uso da instrução `else` que, em conjunto com o `if`, forma uma estrutura condicional completa, conforme a sintaxe:

```
if <expressão>:
   <bloco de código se a expressão é verdadeira>
else:
   <bloco de código se a expressão é falsa>
```

Podemos observar, pela sintaxe, que essa estrutura é semelhante à instrução `if` simples, mas adiciona outro bloco de código que será executado caso a condição não seja verdadeira. A seguir veremos alguns exemplos de utilização da instrução `if – else`, a partir de variações dos códigos da seção anterior.

```
>>>  # Exemplo em que a condição é verdadeira
...  idade = 35
...
...  if idade >= 18:
...      print('Idade suficiente para CNH!')
...  else:
         print('Idade não suficiente para CNH!')
Idade suficiente para CNH!
```

Neste exemplo, foi adicionado uma nova mensagem após a instrução `else`. Se a condição `idade >= 18` for verdadeira, a primeira mensagem será exibida, caso contrário, a segunda mensagem que será exibida. Vejamos o mesmo exemplo, mas com a condição sendo falsa:

```
>>>  # Exemplo em que a condição é falsa
...  idade = 15
...
...  if idade >= 18:
...      print('Idade suficiente para CNH!')
...  else:
         print('Idade não suficiente para CNH!')
Idade não suficiente para CNH!
```

Podemos fazer a mesma alteração para o exemplo da aposentadoria, adicionando uma nova mensagem caso a condição seja falsa:

```
>>>  # Verifica se uma pessoa está apta para se aposentar
...  idade = 20
...  tempo_contrib = 20
...
...  if idade >= 65 or tempo_contrib >= 35:
...      print('Habilitado para solicitar aposentadoria!')
...  else:
         print('Não habilitado para solicitar aposentadoria!')
Não habilitado para solicitar aposentadoria
```

Outra possibilidade também, dependendo da aplicação, é a utilização de *estruturas condicionais aninhadas*, ou seja, if - else dentro de if – else. Por exemplo, estrutura condicional anterior poderia ser reescrita da seguinte maneira:

```
>>>  # Verifica se uma pessoa está apta para se aposentar
...  idade = 20
...  tempo_contrib = 20
...
...  if idade >= 65:
...      print('Habilitado para aposentadoria por idade!')
...  else:
...      if tempo_contrib >= 35:
...          print('Habilitado para aposentadoria por tempo de contribuição!')
...      else:
...          print('Não habilitado para solicitar aposentadoria!')
...
Não habilitado para solicitar aposentadoria
```

Instrução Condicional **if – elif – else**

Adicionalmente, existem situações em que existem mais de uma condição a ser verificada. Por exemplo, considerem o problema de classificar a faixa etária de uma pessoa, segundo a sua idade[22]:

- **Criança:** menor que 12 anos.
- **Adolescente:** maior ou igual que 12 anos e menor que 18 anos.
- **Adulto:** maior ou igual que 18 anos e menor que 60 anos.
- **Idoso:** maior ou igual que 60 anos

Dadas as regras de classificação, é possível escrevermos um código, como este a seguir, que utiliza estruturas condicionais para, dada a idade da pessoa, classificar qual

[22] Exemplo adaptado de (DEVMEDIA, 2022).

a sua faixa etária. Observe que a cada instrução else, uma nova condição é verificada a seguir com outra instrução if. Isso deixa o código confuso, diminuindo a sua legibilidade.

```
>>>  # idade 22 anos, faixa etária: Adulto
...  idade = 22
...
...  if idade < 12:
...     faixa_etaria = 'Criança'
...  else:
...     if idade < 18:
...        faixa_etaria = 'Adolescente'
...     else:
...        if idade < 60:
...           faixa_etaria = 'Adulto'
...        else:
...           faixa_etaria = 'Idoso'
     print('Faixa Etária: ', faixa_etaria)
Faixa Etária: Adulto
```

Para estes casos, podemos utilizar a estrutura condicional if - elif – else. Essa é uma variação da estrutura condicional if – else, onde a instrução elif é empregado para avaliar as expressões intermediárias, sendo adicionada entre as instruções if e else, de tal forma que podem ser acrescentados quantas condições forem necessárias.

A sua sintaxe desta estrutura condicional é:

```
if <expressão 1>:
   <bloco de código se a expressão 1 é verdadeira>
elif <expressão 2>:
   <bloco de código se a expressão 2 é verdadeira>
elif <expressão 3>:
   <bloco de código se a expressão 2 é verdadeira>
else:
   <bloco de código se nenhuma expressão é verdadeira>
```

Desta forma, o exemplo anterior (aquele da classificação de faixa etária) poderia ser reescrito, de uma forma bem mais legível, conforme o código a seguir:

```
>>>  # idade 22 anos, faixa etária: Adulto
...  idade = 22
...  if idade < 12:
...     faixa_etaria = 'Criança'
...  elif idade < 18:
```

```
...         faixa_etaria = 'Adolescente'
...     elif idade < 60:
...         faixa_etaria = 'Adulto'
...     else:
...         faixa_etaria = 'Idoso'
...     print('Faixa Etária: ', faixa_etaria)
Faixa Etária: Adulto
```

Neste exemplo, é definido a primeira condição (`idade < 12`). Caso essa condição seja falsa, o programa seguirá para a avaliação da próxima condição (`elif idade < 18`), que se for verdadeira fará com que o bloco logo abaixo (`faixa_etaria = 'Adolescente'`) seja executado. Caso essa condição ainda não seja atendida, a próxima condição (`elif idade < 60`) será verificada. Por fim, se nenhuma das condições for verdadeira, o programa seguirá para o bloco de código definido pelo `else` (`faixa_etaria = 'Idoso'`).

Estruturas de Repetição

Ao desenvolver uma aplicação, é comum depararmos com a necessidade de executar uma mesma instrução por repetidas vezes. Para tanto, as linguagens de programação possuem as chamadas *estruturas de repetição*, que são estruturas de código que nos permitem executar um bloco de código por uma determinada quantidade de repetições.

Em Python, temos dois tipos de estruturas de repetição. A primeira delas, o `while`, nos permite repetir a execução de um bloco de código enquanto uma determinada condição for verdadeira. A segunda estrutura de repetição é o `for … in`, que permite executar um bloco de código para cada elemento de uma sequência.

A seguir, iremos aprender como utilizar as estruturas de repetição do Python, conhecendo as particularidades de cada uma, para então conseguirmos escolher a estrutura adequada para a nossa solução.

Estrutura de Repetição **while**

O while é uma estrutura de repetição do Python que é utilizada quando não sabemos, de antemão, quantas vezes um bloco de código deverá ser repetido. Por exemplo, podemos escrever um programa que realiza a mesma tarefa de processamento para as transações bancárias de um cliente, mas não sabemos quantas transações serão processadas.

Esta estrutura de repetição possuí a seguinte sintaxe, que pode ser interpretada de uma maneira simples: *enquanto a* <expressão> *for verdadeira o* <bloco de código> *será repetido*.

```
while <expressão>:
  <bloco de código a ser repetido enquanto a expressão for verdadeira>
```

Em outras palavras, o while é uma estrutura de repetição que permite executar um determinado bloco de código enquanto uma determinada condição for verdadeira (True), criando assim o chamado *loop de repetição*. Sua sintaxe é similar à instrução condicional if, com a diferença que o bloco será executado enquanto a condição for verdadeira, e não se a condição for verdadeira.

Após a declaração do while é sempre necessário definir uma condição que será verificada antes de cada execução do bloco de código, e o *loop* somente será interrompido quando essa condição não for mais verdadeira.

Por exemplo, considere o problema de determinar a soma dos primeiros *n* números inteiros. Podemos implementar uma solução para esse problema com o seguinte código:

```
>>>  # exemplo com n = 15
...  n = 15
...  soma = 0
```

```
...     contador = 0
...
...     while contador <= n:
...         soma = soma + contador
...         contador = contador + 1
...
...     print(f'A soma dos primeiros {n} inteiros é {soma}')
A soma dos primeiros 15 inteiros é 120
```

Observe que foi utilizado uma variável auxiliar, chamada `contador`, responsável por controlar quais inteiros já foram somados ou não. Assim, a condição `contador <= n` será verdadeira enquanto o valor de `contador` não for maior que `n`. Enquanto esta condição for verdadeira, a variável `soma` vai acumulando o somatório dos números e o `contador` vai sendo incrementado em uma unidade. Uma vez que o `contador` seja maior que `n`, o *loop* é interrompido. É possível reescrever o código, sem o uso da variável auxiliar `contador`, utilizando o próprio `n` como variável de controle.

```
>>>   # exemplo com n = 15
...     n = 15
...     soma = 0
...
...     while n >= 0:
...         soma = soma + n
...         n = n - 1
...
...     print(f'A soma dos primeiros inteiros é {soma}')
A soma dos primeiros inteiros é 120
```

Estrutura de Repetição **for … in**

Em Python, a estrutura de repetição `for … in` permite executar um bloco de código repetidas vezes, sendo uma repetição para cada elemento em uma sequência. Em outras palavras, ela permite percorrer (iterar) uma *sequência iterável*.

Antes de prosseguirmos, é importante entendermos o conceito de *sequência iterável*. Basicamente, uma sequência é dita iterável quando ela tem a capacidade de retornar cada um de seus elementos de forma individual. Até o momento,

apresentamos apenas uma sequência iterável: o tipo de dados *string*. Entretanto, outros tipos de dados não-primitivos, que também são iteráveis, serão apresentados no Capítulo 5.

Diferentemente do `while`, a estrutura de código `for … in` repetirá um determinado bloco de código por um número definido de vezes, que é dado pelo tamanho da sequência que será percorrida. A sintaxe do `for … in` é da dada a seguir, onde `<item>` corresponde a cada elemento da `<sequência>` a ser iterada.

```
for <item> in <sequência>:
   <bloco de código a ser repetido para cada item da sequência>
```

Para ilustrar o funcionamento desta instrução, considere o problema de contar o número de ocorrências da letra 'a' em uma *string*. Podemos resolver este problema iterando a *string* (podemos iterar, porque a *string* é uma sequência de caracteres) e verificando, para cada caractere (elemento), se ele corresponde à letra 'a'. Se sim, incrementamos em um o nosso contado. Veja a implementação no código abaixo:

```
>>>   # exemplo de palavra: araraquara
...   p = 'araraquara'
...   contador = 0
...
...   for letra in p:
...      if letra == 'a':
...         contador = contador + 1
...   print(f"A palavara {p} possuí {contador} letras 'a'")
A palavara araraquara possuí 5 letras 'a'
```

Sequências que contém inteiros consecutivos são muito comuns e, por esta razão, o Python fornece uma maneira bem simples para a sua criação: a função `range`, que possui a seguinte sintaxe:

```
range(inicio, fim, incremento)
```

Essa função permite criar uma faixa (*range*) de números inteiros, começando por inicio e finalizando com fim – 1 (o valor de fim não é incluído na sequência resultante). O valor de incremento indica o incremento que será adicionado entre um item e outro. Os argumentos inicio e incremento são opcionais (ou seja, não precisam ser declarados), e por padrão a sequência irá iniciar no número 0 e ser incrementada em 1. Vejamos alguns exemplos de utilização da função range.

```
# 0, 1, 2, 3, 4
range(5)
# 3, 4, 5, 6, 7
range(3, 8)
# 0, 5, 10, 15, 20
range(0, 21, 5)
# 2, 4, 6, 8
range(2, 10, 2)
```

Em conjunto com a estrutura de repetição for … in, a função range pode ser utilizada para executar um determinado bloco de código, pela quantidade de vezes indicadas na função range. Por exemplo, o problema de determinar a soma dos primeiros *n* números inteiros, poderia ser solucionado da seguinte forma:

```
>>>  # exemplo com n = 15
...  n = 15
...  soma = 0
...
...  for num in range(n + 1):
...      soma = soma + num
...
...  print(f'A soma dos primeiros {n} inteiros é {soma}')
A soma dos primeiros 15 inteiros é 120
```

Observe que foi necessário utilizar a função range com n+1 (range(n + 1). Se fosse utilizado apenas n como argumento (range(n)), o resultado seria a soma de apenas os 14 primeiros inteiros, e não os 15 primeiros como pretendido.

Interrupção da Estruturas de Repetição

Por padrão, a estrutura de repetição `for … in` só será interrompida depois que o último elemento da sequência for processado. Entretanto, em alguns casos, pode ser necessário interrompermos o *loop* antes de chegar ao fim da sequência. Para tanto, existe o comando `break`, que encerra a instrução de repetição ao verificar se uma condição específica é verdadeira. A sintaxe deste conceito é apresentada abaixo. O comando `break` deve ser utilizado em conjunto com uma instrução condicional (`if – else`).

```
for <item> in <sequência>:
   <bloco de código a ser repetido>
   if <expressao>
      <outro bloco de código>
      break
```

Por exemplo, considere o problema de encontrar o primeiro número divisível por *21* e que esteja entre *250* e *300*. Como mencionado, o número precisa estar em um intervalo definido, então pode-se utilizar a função `range(250, 301)`, para gerar a sequência de valores a serem verificadas. Além disso, uma vez encontrado o número, não é mais necessário prosseguir com a busca. Desta forma, o problema pode ser resolvido pelo seguinte código:

```
>>>    # Realiza a busca na faixa de 250 e 300
...    for num in range(250, 301):
...       # verifica se o número é divisível por 21
...       if num % 21 == 0:
...          print('Número encontrado:', num)
...          # se for divísivel por 21, interrompe a busca
...          break

Número encontrado: 252
```

Também podemos utilizar esse mesmo conceito para a estrutura de repetição `while`, com a seguinte sintaxe:

```
while <condição de repetição>:
   <bloco de código a ser repetido>
```

```
if <condição de parada>
    <outro bloco de código>
    break
```

Capítulo 8. Estruturas de Dados

No Capítulo 3 aprendemos sobre os principais tipos de dados primitivos do Python. Por meio da atribuição de valores destes tipos primitivos a variáveis simples, vimos que é possível realizar diversas tarefas como operações matemáticas e o tratamento de *strings*.

Entretanto, durante o desenvolvimento de um software, é bem comum a necessidade de realizar operações com uma coleção de dados, e não mais com valor simples, apenas. Por exemplo, podemos ter situações em que seja necessário ordenar uma lista de pessoas pelo nome delas, procurar se um determinado número está presente em uma sequência ou transformar todas as palavras de um texto em letras maiúsculas.

Vejamos uma situação prática do mundo real. Considere que estamos criando um software para processar o IPTU de todos os imóveis da cidade de Belo Horizonte e para isso criaremos uma variável simples para representar cada imóvel. Os mais atentos já devem ter percebido que seria uma tarefa quase impossível criar centenas de milhares de variáveis, pois esta é a dimensão de imóveis em Belo Horizonte.

Para contornar este problema, as linguagens de programação oferecem recursos para manipular uma coleção de dados, com tamanhos variados. Estes recursos, que são chamados *estruturas de dados*, permitem organizar e estruturar os dados de uma forma eficiente, oferecendo diferentes maneiras de acessar e armazenar os valores da coleção na memória do computador.

Em Python, as estruturas de dados são sequências que são chamadas de *tipos não-primitivos de dados* (MARVIN, NG'ANG'A e OMONDI, 2018). Assim como os tipos

primitivos, as estruturas de dados possuem operadores e métodos que podemos utilizar para acessar e manipular os valores que estão contidos nelas.

Neste capítulo aprenderemos um pouco sobre os tipos de estruturas de dados em Python e, consequentemente, suas principais funcionalidades, para então selecionarmos qual estrutura devemos utilizar em nossas soluções.

Estruturas de Dados em Python

Antes de introduzirmos quais são as estruturas de dados que o Python oferece nativamente, vamos primeiro entender a importância de se utilizar um tipo de dado que possibilite o armazenamento de uma coleção extensa de valores. Para exemplificar, vamos utilizar uma das tarefas mais recorrentes na vida de um desenvolvedor: a busca pelo maior (ou menor) valor em uma sequência, não ordenada, de números. Portanto, considere o seguinte problema: *Dada as idades de um grupo de pessoas, qual a idade da pessoa mais velha?*

Considere primeiramente um grupo de apenas três pessoas. Como são poucas pessoas, podemos criar uma variável p*n* para cada pessoa e encontrar o maior valor comparando as variáveis entre si, conforme o exemplo:

```
>>>  # cria as variáveis com as idades
...  p1 = 27
...  p2 = 49
...  p3 = 12
...
...  # verifica qual a maior idade
...  if p1 >= p2:
...      if p1 >= p3:
...          print('Maior idade:', p1)
...  elif p2 >= p3:
...      print('Maior idade:', p2)
...  else:
...          print('Maior idade:', p3)
Maior idade: 49
```

No exemplo anterior foram utilizados então, para três pessoas, três variáveis, e foram realizadas três comparações. Parece simples, certo? Vamos avançar o problema então e agora vamos considerar um grupo maior de quatro pessoas.

Seguindo a mesma lógica anterior, após declarar quatro variáveis, seria necessário também apenas quatro comparações? A **resposta é não, pois seriam necessárias sete comparações**, conforme o trecho de código a seguir.

```
if p1 >= p2:
   if p1 >= p3:
      if p1 >= p4:
         print('Maior idade:', p1)
      else:
         print('Maior idade:', p4)
   elif p3 >= p4:
      print('Maior idade:', p3)
   else:
      print('Maior idade:', p4)
if p2 >= p3:
   if p2 >= p4:
      print('Maior idade:', p2)
   else:
       print('Maior idade:', p4)
elif p3 >= p4:
    print('Maior idade:', p3)
else:
     print('Maior idade:', p4)
```

Ao continuar aumentando o número de pessoas no grupo (n), a quantidade de comparações que deverão ser declaradas irá crescer exponencialmente, em uma ordem de $2^{n-1} - 1$, conforme a ilustra a tabela abaixo:

Quantidade de Pessoas	Número de Comparações
3	3
4	7
5	15
6	31
10	511
20	524.287
35	17.179.869.183

Podemos observar que, mesmo com um pequeno número de pessoas, é totalmente inviável implementar esta estratégia de utilizar variáveis simples. Por exemplo, ao adotar esta estratégia para um grupo de 35 pessoas, um desenvolvedor habilidoso (que consegue escrever uma comparação por segundo) gastaria mais de 550 anos. Ou seja, mesmo se ele tivesse iniciado a escrita código no dia que Cabral descobriu o Brasil, ele ainda não teria terminado! E nem estamos considerando o tempo necessário para executar o programa, que também demoraria muito.

Imaginem então fazer a mesma análise para todos os habitantes do Brasil? Felizmente existem estratégias bem mais simples e elegantes do que essa, e elas incluem o uso de estrutura de dados. Muita coisa evoluiu nas últimas décadas e diferentes estruturas e algoritmos para manipulação eficiente dos dados foram desenvolvidas (CORMEN, LEISERSON, *et al.*, 2012).

Em Python não poderia ser diferente, pois contamos com estruturas de dados que, juntamente com algoritmos já implementados na linguagem, oferecem formas otimizadas de acesso e processamento de uma coleção extensa de valores. As quatro estruturas de dados principais do Python são:

Estrutura	Tipo	Exemplo	Características
Lista	list	[1, 2, 3, 4]	Ordenada e mutável.
Tupla	tuple	(1, 2, 3, 4)	Ordenada e imutável.
Conjunto	set	{1, 2, 3, 4}	Não-ordenada, mutável e valores únicos.
Dicionário	dict	{'a':1, 'b':2, 'c':3}	Mapeamento (key, value), não-ordenado e mutável.

A tabela acima cita as características das estruturas, o conceito de *mutabilidade.* Em termos gerais, mutabilidade é um conceito técnico para indicar que o estado interno do valor de uma variável pode ser modificado. Assim, a definição mais simples é: um

valor cujo estado interno pode ser alterado é *mutável*. Por outro lado, *imutável* não permite nenhuma alteração no valor depois de criado.

Para concluir, as estruturas de dados se referem ao conjunto de operações e estratégias que são oferecidas ao desenvolvedor. A escolha da estrutura ideal dependerá do problema que iremos resolver, sendo que o objetivo é sempre escolher aquela que ofereça a funcionalidade desejada e que minimize o custo computacional das operações. As principais estruturas de dados em Python serão apresentadas nas próximas seções, juntamente com exemplos de utilização, para que possamos entender as particularidades a serem considerada no momento da escolha.

Listas

Em Python, uma lista (list) é uma sequência ordenada de valores, que são identificados por um índice. Estes valores, que compõem uma lista, são chamados de elementos ou itens. Listas são estruturas de dados muito similares às *strings*, que por sua vez são sequência de caracteres, apenas. A diferença é que os elementos de uma lista podem ser de qualquer tipo, ou seja, podem ser *homogêneos* (todos os valores do mesmo tipo) e *heterogêneos* (valores com tipos distintos).

Diferentemente das *strings*, uma lista é também uma sequência mutável e dinâmica. Uma vez que sejam criadas, é possível:

- Alterar o valor de um ou mais elementos.
- Os elementos podem ser adicionados, removidos ou substituídos.
- A ordem dos elementos pode ser alterada.

A seguir será apresentada formas de criar listas e acessar seus elementos, assim como os principais métodos que este tipo de dados oferece.

Criação de Listas e Acesso dos Elementos

Listas podem ser criadas por meio da declaração de valores separados por vírgulas e delimitados por colchetes. Por exemplo, o código abaixo cria uma lista dos primeiros números primos, a lista é homogênea visto que todos elementos são do tipo `int`.

```
>>>     [2, 3, 5, 7]
[2, 3, 5, 7]
```

A seguir veremos exemplos que ilustram a criação de listas com valores heterogêneos (diferentes tipos). Podemos criar listas em que os elementos são de qualquer tipo, inclusive com valores nulos (`None`). Podemos também criar listas com outras listas dentro[23]. E por fim, podemos também criar listas vazias.

```
>>>     # Lista com valores heterogêneos
...     [2, 'a', 5.44, True]
[2, 'a', 5.44, True]
>>>     # Lista com elementos nulos
...     ['um', 'dois', None, 4]
['um', 'dois', None, 4]
>>>     # Listas com outras listas dentro (listas aninhadas)
...     l = [0, 1]
...     [l, 'dois', 'três', [4, 5], 'seis']
[[0, 1], 'dois', 'três', [4, 5], 'seis']
>>>     # Lista vazia
...     l = []
...     print(l)
[]
```

[23] Quando isso ocorre, pode-se dizer que são listas aninhadas.

De forma semelhante ao acesso dos caracteres de uma *string* (Seção Operador de Acesso das *Strings*), a sintaxe para acessar os elementos de uma lista é por meio do operador ([]), especificando o índice (posição) do elemento a ser acessado (ex.: [0] e l[1]) e sempre com o primeiro elemento no índice *0*. Também podemos utilizar índices negativos e o conceito de *slicing* (faixa de índices), que nos retorna uma *sub-lista*.

```
>>>    lista = [2, 'a', 5.44, True, None, 'casa']
...    # acesso por índices
...    print(lista[0])
...    print(lista[3])
...    print(lista[-1])
...    # acesso por slices
...    print(lista[1:4])
...    print(lista[-2:])
...    print(lista[:])
2
True
casa
['a', 5.44, True]
[None, 'casa']
[2, 'a', 5.44, True, None, 'casa']
```

Agora que aprendemos a criar listas, e acessar seus elementos, que tal reescrevermos o exemplo da seção anterior utilizando uma lista? Para um exemplo inicial, considere o mesmo problema apresentado inicialmente: encontrar o maior idade entre um grupo de três pessoas. Podemos então criar uma lista com três elementos e, em seguida utiliza a estrutura de repetição for … in para encontrar o maior valor.

```
>>>    lista = [2, 'a', 5.44, True, None, 'casa']
...    # acesso por índices
...    print(lista[0])
...    print(lista[3])
...    print(lista[-1])
...    # acesso por slices
...    print(lista[1:4])
...    print(lista[-2:])
...    print(lista[:])
2
True
casa
['a', 5.44, True]
[None, 'casa']
```

```
[2, 'a', 5.44, True, None, 'casa']
```

Observe que fizemos uso da estrutura de repetição `for … in` para iterar a lista, pois, o tipo `list` também é uma sequência iterável e a esta estrutura de repetição pode percorrer qualquer sequência iterável, como aprendemos na Seção Estrutura de Repetição **for … in**. O mesmo código pode ser utilizado, também, para um grupo maior de pessoas. Por exemplo, para 15 pessoas seria necessário apenas modificar a lista inicial, acrescentando as idades das novas pessoas.

```
>>>  # cria uma lista para armazenar as 10 idades
...  idades = [27, 49, 12, 67, 21, 32, 18, 45, 84, 53, 22, 56, 80, 35, 18]
...
...  # inicialmente assume que a primeira idade é a maior
...  maior_idade = pessoas[0]
...
...  # percorre a lista verificando a maior idade
...  for idade in idades:
...      if idade > maior_idade:
...          maior_idade = idade
...
...  # exibe a maior idade encontrada
...  print('Maior idade:', maior_idade)
Maior idade: 85
```

Outra vantagem é que escrevemos apenas uma comparação (`if idade > maior_idade`), que será executada n vezes, onde n é o tamanho da lista a ser processada. Por exemplo, para encontrarmos a maior idade dentre os habitantes do Brasil (aproximadamente 212 milhões de pessoas), seriam necessários apenas 4,5 segundos! Será que existe uma maneira mais legível e eficiente de resolver o problema do exemplo anterior? A resposta é sim e veremos como na próxima seção!

Recursos Úteis das Listas

Nesta seção, veremos diversos recursos úteis do tipo `list`, que são compostos por operadores, métodos e funções que manipulam valores destas estruturas de dados. Iniciaremos com exemplos de como *adicionar*, *alterar* e *remover* elementos das listas.

```
>>>  # Criar uma lista inicial, com elementos heterogêneos
...  lista = [2, 'a', 'b', 'c', 5.44, True]
...  print(lista)
[2, 'a', 'b', 'c', 5.44, True]
>>>  # Adiciona um novo elemento na lista: Método append
...  lista.append(999)
...  print(lista)
[2, 'a', 'b', 'c', 5.44, True, 999]
>>>  # Altera o valor do quarto e o do último elemento
...  lista[3] = 'a'
...  lista[-1] = lista[-1] + 1
...  print(lista)
[2, 'a', 'b', 'a', 5.44, True, 1000]
>>>  # Remove a primeira ocorrência do elemento 'a'
...  lista.remove('a')
...  print(lista)
[2, 'a', 'b', 'a', 5.44, True, 1000]
```

Vamos analisar o código anterior e entender o que foi realizado:

1. Primeiro criarmos uma lista com seis elementos.

2. Após a criação, adicionamos um elemento, utilizando o método `append(elem)`, que é responsável por adicionar um novo elemento `elem` no final da lista.

3. Em seguida alteramos os valores do terceiro e do último elemento da lista, por meio do operador de acesso[24].

4. Por fim removemos a primeira ocorrência do elemento 'a', por meio do método `remove(elem)`. Este método remove a primeira ocorrência do elemento `elem` na lista. Um erro de execução acontecerá se o elemento `elem` não pertencer à lista.

[24] Apesar de semelhantes, listas e *strings* possuem uma diferença muito importante. Enquanto as listas são tipos de dados mutáveis, as *strings* são imutáveis. Portanto, conseguimos alterar o valor de um elemento na lista utilizando o operador `[]`, mas não conseguimos fazer o mesmo com aos caracteres de uma string.

De forma semelhante às *strings*, podemos utilizar os operadores de concatenação (+), repetição (*) e de filiação (in) com as listas. Também podemos utilizar funções e métodos. Vejamos a seguir alguns exemplos de operações úteis que são implementados no Python e que os desenvolvedores utilizam frequentemente.

```
>>>  l1 = [30, 10, 20]
...  l2 = [2, 'a', 5.44, True]
...
...  # Operações de concatenação (+), repetição (*) e filiação (in)
...  print(l1 + l2)
...  print(l1 * 3)
...  print(10 in l1)
...
...  # Funções úteis
...  print(len(l2)) # len: retorna a quantidade de elementos da lista.
...  print(sum(l1)) # sum: retorna a soma dos elementos de uma lista.
...  print(max(l1)) # max: retorna o maior elemento da lista (!!!!)
...
...  # Métodos que alteram os valores internos da lista
...  l2.reverse() # reverse: inverte a ordem dos elementos
...  print(l2)
...  l1.extend([10, 20, 30, 40, 10]) # extend: adiciona elementos de outra sequência
...  print(l1)
...  l1.sort() # sort: ordena os valores da lista
...  print(l1)
...  l2.insert(2, 'novo valor') # insert: adiciona um elemento em um índice especifico
...  print(l2)
...  l2.pop() # pop: remove o último elemento da lista
...  print(l2)
...  l2.clear() # clear: limpa a lista, removendo todos os elementos
...  print(l2)
...
...  # Métodos que retornam valores e não alteram a lista
...  print(l1.index(40)) # index: retorna o índice da primeira ocorrência do elemento
...  print(l1.count(10)) # count: conta as ocorrências do elemento
[30, 10, 20, 2, 'a', 5.44, True]
[30, 10, 20, 30, 10, 20, 30, 10, 20]
True

4
60
30

[True, 5.44, 'a', 2]
[30, 10, 20, 10, 20, 30, 40, 10]
[10, 10, 10, 20, 20, 30, 30, 40]
[True, 5.44, 'novo valor', 'a', 2]
[True, 5.44, 'novo valor', 'a']
[]
```

```
7
3
```

Calma! Apesar de tantos exemplos não é necessário decorar todos estes operadores, funções e métodos. Precisamos apena saber que eles existem e que, a medida do necessário, podemos sempre consultar a documentação do Python e relembrá-los (PSF, 2022).

É importante apenas ressaltar que algumas destas funções e métodos somente podem operar listas homogêneas e com tipos numéricos (`int` e `float`) ou tipo *string* (`str`). Isso ocorre porque estes métodos, como `sort()`, `min()`, `max()`, fazem comparações e cálculos aritméticos internos entre os elementos, e como vimos no Capítulo 3, os operadores aritméticos e de comparação não são implementados para operandos de diferentes tipos. Por exemplo, se tentássemos aplicar a função `max()` na lista `l2` do exemplo anterior, teríamos o seguinte erro:

```
>>>  l2 = [2, 'a', 5.44, True]
...  print(max(l2))

TypeError                          Traceback (most recent call last)
<ipython-input-52-3f5555a227f8> in <module>()
    1 l2 = [2, 'a', 5.44, True]
----> 2 print(max(l2))

TypeError: '>' not supported between instances of 'str' and 'int'
```

E por falar na função `max()`, lembra do problema de encontrar a maior idade em um grupo de pessoas? Pois então, com esta função podemos resolver o problema utilizando apenas uma linha! Veja o código, por exemplo, para o grupo de 15 pessoas. Primeiros criamos a lista com as 15 idades e em seguidas já podemos aplicar a função `max()` na lista.

```
>>>  idades = [27, 49, 12, 67, 21, 32, 18, 45, 84, 53, 22, 56, 80, 35, 18]
...  print('Maior idade:', max(idades))

Maior idade: 84
```

E a melhor parte é que esta função é implementada de uma forma tão eficiente que faz com que encontremos a maior idade dentre os habitantes do Brasil em apenas 7 microssegundos, ou seja, 0,000007 segundos. Bem melhor que os 4,5 segundos da nossa implementação anterior!

Tuplas

As tuplas são estruturas de dados ordenadas do tipo `tuple` e são semelhantes às listas, com uma diferença fundamental: tuplas são imutáveis. Pelo conceito de imutabilidade, uma vez que criarmos uma tupla, não será mais possível adicionar, alterar ou remover seus elementos. Este tipo de estrutura é utilizado, frequentemente, para armazenar sequências de diferentes informações, porém, com a quantidade de elementos definidos e que não irá requerer alterações ao longo da execução do código.

A criação das tuplas é semelhante às listas, entretanto, são utilizados parênteses e não colchetes, para delimitar os elementos. Tuplas também armazenam elementos de diferentes tipos de dados, podendo ser homogêneas ou heterogêneas. O acesso é realizado da mesma forma que as listas.

```
>>>  # Criação de uma tupla homogênea
...  tupla = (0, 1, 2, 3, 4)
...  print(tupla)
...
...  # Tupla heterogênea
...  tupla2 = (2, 'a', 5.44, True, None)
...  print(tupla2)
...
...  # Tupla vazia
...  tupla3 = ()
...  print(tupla3)
...
...  # acesso por índices
...  print(tupla[0])
...  print(tupla[3])
...  print(tupla[-1])
...
...  # acesso por slices
...  print(tupla2[1:4])
...  print(tupla2[-2:])
```

```
...     print(tupla2[:])

(0, 1, 2, 3, 4)
(2, 'a', 5.44, True, None)
()

0
3
4

('a', 5.44, True)
(True, None)
(2, 'a', 5.44, True, None)
```

Também existem operadores, funções e métodos capazes de operar e transformar os dados armazenados em um tupla, que seguem a mesma sintaxe e padrão de nomes das operações em listas. Vejamos alguns exemplos, muito semelhantes aos utilizados na seção anterior, que ilustram a utilização de operadores, funções e métodos em tuplas:

```
>>>   t1 = (30, 10, 20)
...     t2 = (2, 'a', 5.44, True)
...
...     # Operações de concatenação (+), repetição (*) e filiação (in)
...     print(t1 + t2)
...     print(t1 * 3)
...     print(10 in t1)
...
...     # Funções úteis
...     print(len(t2)) # len: retorna a quantidade de elementos da tupla
...     print(min(t1)) # min: retorna o menor elemento da tupla
...     print(max(t1)) # max: retorna o maior elemento da tupla
...     print(sum(t1)) # sum: retorna a soma dos elementos da tupla
...
...     # Métodos que retornam valores
...     print(t1.index(20)) # index: retorna o índice da primeira ocorrência do elemento
        print(t2.count('a')) # count: conta as ocorrências do elemento

(30, 10, 20, 2, 'a', 5.44, True)
(30, 10, 20, 30, 10, 20, 30, 10, 20)
True

4
10
30
60

2
1
```

Entretanto, uma vez que as tuplas são imutáveis, não existem métodos capazes de alterar os valores internos dela, como por exemplo `append`, `remove`, `sort`, `reverse` e `pop` e `clear`. Da mesma forma, a substituição de elementos, por meio do operador de atribuição (=), não é permitida, como por exemplo `tupla[0] = 99`. Ao tentar realizar essas operações, teremos um erro de execução alertando que o tipo `tuple` não as suportam

```
>>>   t1 = (1, 2, 3)
...   t1[0] = 4

TypeError                    Traceback (most recent call last)
....
TypeError: 'tuple' object does not support item assignment
```

Ok, mas se as tuplas são tão semelhantes às listas, ao mesmo tempo que as listas oferecem uma maior flexibilidade (por serem mutáveis), porque então deveríamos utilizar tuplas? Mark Pilgrim, em seu livro (PILGRIM, 2009), enumera três pontos interessante sobre as tuplas, que são transcritas a seguir:

- "Tuplas são mais rápidas que listas. Se você está definindo uma sequência constante de valores e você vai ter que iterar sobre ele, utilize uma tupla ao invés de uma lista."

- "Tuplas tornam o seu código mais seguro, uma vez que eles protegem contra gravações, os dados que não precisam ser alterados. Usar uma tupla em vez de uma lista é como ter uma declaração implícita de que esses dados são constantes e que uma função específica será necessária para sobrescrevê-los."

- "Tuplas podem ser utilizadas como chaves de dicionários[25]. As listas nunca podem ser utilizadas como chaves de dicionário, porque as listas não são imutáveis."

Com os pontos apresentados, conseguimos entender que existem situações adequadas para cada tipo de estrutura de dados (lista ou tupla), e por isso devemos tomar a decisão, da escolha da estrutura ideal, de acordo com o nosso caso de uso.

Conjuntos

Em Python os *conjuntos* (*sets*) são estruturas de dados do tipo `set`, não-ordenadas, que representam uma coleção de itens únicos, ou seja, itens sem repetições. Assim como a lista, o *conjunto* também é uma estrutura mutável, suportando operações de inserção, alteração e remoção de elementos.

A diferença fundamental entre as listas e os conjuntos é que os elementos dos conjuntos são únicos e a sua ordem dentro da estrutura não importa. Por isso eles são ditos como estruturas não-ordenadas. Adicionalmente, o tipo `set` suporta operações matemáticas entre conjuntos como *união*, *interseção* e *diferença*.

Os conjuntos também são criados de forma semelhante às listas, mas com chaves sendo utilizadas ao invés dos colchetes. Outra diferença é que, nos conjuntos, a ordem de declaração dos itens não importa.

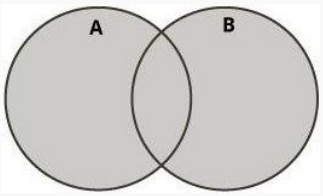
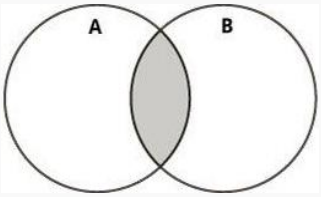
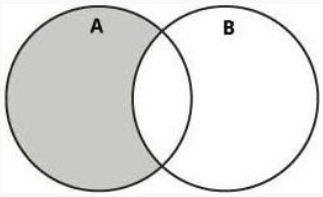
```
>>>  # Criação de um conjunto homogêneo
...  c1 = {3, 0, 1, 4, 3}
...  print(c1)
...
...  # Criação do mesmo conjunto, porém com uma ordenação diferente dos itens
...  c2 = {2, 1, 4, 3, 0}
```

[25] Para os novatos, talvez o último ponto seja confuso, mas vocês entenderão daqui a pouco quando conhecermos as estruturas de dados do tipo dicionário.

```
...     print(c2)
...
...     # Conjunto heterogêneo
...     c3 = {2, 'a', 5.44, True, None}
...     print(c3)
{0, 1, 2, 3, 4}
{0, 1, 2, 3, 4}
{True, 2, 5.44, None, 'a'}
```

Os conjuntos não possuem uma maneira direta de acessar os seus elementos. Isso pode não fazer sentido agora, mas o tipo `set` é uma estrutura projetada para não fornecer acesso aos elementos, e sim para representar os dados em forma de conjuntos e oferecer suas operações matemáticas tradicionais.

Para tornar mais clara a utilidade das estruturas de dados deste tipo, iremos conhecer as principais operações dos conjuntos. Mas, antes, vale a pena revisitarmos como essas operações matemáticas são realizadas, conforme a figura[26] abaixo:

União - $A \cup B$	**Interseção –** $A \cap B$	**Diferença –** $A - B$
		
O resultado será o conjunto formado por todos os elementos de **A** e **B**	O resultado será o conjunto formado apenas pelos elementos que estejam em **A** e **B**, simultaneamente	O resultado será o conjunto formado pelos elementos que estejam em **A**, mas que não estejam em **B**

Em Python, essas operações podem ser realizadas por meio de operadores ou métodos, conforme os exemplos a seguir.

```
>>> # Criação dos conjuntos A e B
...  A = {1, 2, 3, 4, 5}
```

[26] Imagens adaptadas de (PAREWA LABS PVT. LTD., 2022)

```
...     B = {4, 5, 6, 7, 8}
...     print('A:', A)
...     print('B:', B)
...
...     # Operação de União: (A ∪ B)
...     print('A | B =>', A | B)
...     print('A.union(B) =>', A.union(B))
...
...     # Operação de Interseção: (A ∩ B)
...     print('A & B =>', A & B)
...     print('A.intersection(B) =>', A.intersection(B))
...
...     # Operação de Diferença: (A - B) e (B - A)
...     print('A - B =>', A - B)
...     print('A.difference(B) =>', A.difference(B))
...     print('B - A =>', B - A)
...     print('B.difference(A) =>', B.difference(A))

A: {1, 2, 3, 4, 5}
B: {4, 5, 6, 7, 8}

A | B => {1, 2, 3, 4, 5, 6, 7, 8}
A.union(B) => {1, 2, 3, 4, 5, 6, 7, 8}

A & B => {4, 5}
A.intersection(B) => {4, 5}

A - B => {1, 2, 3}
A.difference(B) => {1, 2, 3}
B - A => {8, 6, 7}
B.difference(A) => {8, 6, 7}
```

Adicionalmente, existem outros operadores, funções e métodos que operam as estruturas do tipo `set`. Algumas destas operações nós já conhecemos, como as funções `len()`, `sum()`, `min()` e `max()`, os operadores `in` e `not in` e os métodos `set.clear()`, `set.pop()` e `set.remove()`.

Uma vez que nós já sabemos como estas operações trabalham, vamos focar em conhecer os principais métodos exclusivos do tipo `set`, que possuem o objetivo de manipular e realizar diferentes verificações com essa estrutura de dados. Existem outros métodos, menos utilizados, que podem ser consultados diretamente na documentação do Python (PSF, 2022).

Vejamos a seguir exemplos dos principais métodos do tipo `set`:

```
>>>  # Criação dos conjuntos A e B
...  c1 = {1, 2, 3, 4, 5}
...  c2 = {4, 5}
...  c3 = {91, 92, 93}
...
...  # Adiciona um elemento ao conjunto
...  c1.add(6)
...  print(c1)
...
...  # Adiciona os elementos de uma sequência iterável
...  c1.update([2, 4, 6, 8])
...  print(c1)
...
...  # Descarta um elemento do conjunto,
...  c1.discard(8)
...  print(c1)
...  # Diferentemente do set.remove(), o discard não gera um erro
...  # se o elmento a ser removido não existir
...  c1.discard(99)
...  print(c1)
...
...  # Verifica se os conjuntos são disjuntos, ou seja,
...  # se não possuem nenhum elemento em comum
...  print(c1.isdisjoint(c2))
...  print(c1.isdisjoint(c3))
...
...  # Verifica se o conjunto é subconjunto de outro
...  print(c1.issubset(c2))
...  print(c2.issubset(c1))
...
...  # Verifica se o conjunto contém outro conjunto (superset)
...  print(c1.issuperset(c2))
...  print(c2.issuperset(c1))
{1, 2, 3, 4, 5, 6}

{1, 2, 3, 4, 5, 6, 8}

{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}

False
True

False
True

True
False
```

Antes de encerrarmos esta seção, não podemos deixar de ilustrar um exemplo prático da utilização do tipo `set`. Para tanto, considere que estamos desenvolvendo um programa para gerenciar uma escola de línguas estrangeiras que possui três turmas de

alunos para os seguintes idiomas: inglês, espanhol e francês. Considere também que um mesmo aluno pode estar matriculado em uma única turma ou em múltiplas turmas, conforme as listagens abaixo:

Turma	Alunos
ING – Inglês	Gabriel, Caio, Maria, Ana, Juliano, Flávia, Rubens e Bruna
ESP – Espanhol	Caio, Artur, Beatriz, Carolina, Maria, Juliano, Bruna e Rui
FRA – Francês	Pedro, Bruna, Paula, Tiago, Maria, Flávia, Rui, Carolina

Neste momento estamos interessados em resolver dois problemas desta escola:

1. Criar uma relação com todos os alunos da escola, sem repetições;

2. Identificar os alunos que estão matriculados em pelo menos duas turmas para oferecermos um desconto;

Primeiro, vamos focar no *problema 1*. Se simplesmente concatenássemos as listas de alunos das turmas, teríamos uma lista com todos os alunos. Entretanto, podemos observar na saída do código que teríamos alunos repetidos como *Bruna* e *Rui*.

```
>>>  # Criação das listas de alunos das turmas
...  ING = ['Gabriel', 'Caio', 'Maria', 'Ana', 'Juliano', 'Flávia', 'Rubens', 'Bruna']
...  ESP = ['Caio', 'Artur', 'Beatriz', 'Carolina', 'Maria', 'Juliano', 'Bruna', 'Rui']
...  FRA = ['Pedro', 'Bruna', 'Paula', 'Tiago', 'Maria', 'Flávia', 'Rui', 'Carolina']
...
...  # Concatenação de todas as listas de alunos
...  ALL = ING + ESP + FRA
...
...  # Ordenação para melhor visualização
...  ALL.sort()
...
...  # Exibição do resultado
...  print('Relação de todos os alunos da escola:')
...  print(ALL)

Relação de todos os alunos da escola:
['Ana', 'Artur', 'Beatriz', 'Bruna', 'Bruna', 'Bruna', 'Caio', 'Caio', 'Carolina', 'Carolina', 'Flávia', 'Flávia', 'Gabriel', 'Juliano', 'Juliano',
'Maria', 'Maria', 'Maria', 'Paula', 'Pedro', 'Rubens', 'Rui', 'Rui', 'Tiago']
```

Se ao invés de listas utilizarmos conjuntos, podemos resolver o problema sem o inconveniente dos nomes repetidos. Basta utilizarmos a operação de união de conjuntos, em que o resultado será outro conjunto, com todos os alunos. Porém, o conjunto resultante será sem repetições (pois esta é a definição de um conjunto). Observe, pela saída do código abaixo, que agora temos uma relação dos nomes de todos os alunos, sem repetições.

```
>>>   # Criação dos conjuntos de alunos das turmas
...     ING = {'Gabriel', 'Caio', 'Maria', 'Ana', 'Juliano', 'Flávia', 'Rubens', 'Bruna'}
...     ESP = {'Caio', 'Artur', 'Beatriz', 'Carolina', 'Maria', 'Juliano', 'Bruna', 'Rui'}
...     FRA = {'Pedro', 'Bruna', 'Paula', 'Tiago', 'Maria', 'Flávia', 'Rui', 'Carolina'}
...
...     # Operação de união dos conjuntos (união dos alunos de todas as turmas)
...     # Também poderia ser: ALL = ING.union(ESP).union(FRA)
...     ALL = ING | ESP | FRA
...
...
...     # Exibição do resultado
        print('Relação de todos os alunos da escola:')
        print(ALL)

Relação de todos os alunos da escola:
{'Ana', 'Juliano', 'Gabriel', 'Tiago', 'Beatriz', 'Maria', 'Rubens', 'Caio', 'Artur', 'Rui', 'Pedro', 'Bruna', 'Flávia', 'Carolina', 'Paula'}
```

O *problema 2* também pode ser resolvido utilizando o tipo `set` e suas operações. Para tanto, precisamos primeiro descobrir quais são os alunos que estão, simultaneamente, em cada par de turmas (*ING* e *ESP*), (*ING* e *FRA*) e (*ESP* e *FRA*). Ou seja, precisamos calcular a interseção entre estes pares de conjuntos. Em seguida podemos realizar a união destas interseções, gerando a lista final de alunos que estão em pelo menos duas turmas. O código abaixo ajuda a entender essa lógica.

```
>>>   ING = {'Gabriel', 'Caio', 'Maria', 'Ana', 'Juliano', 'Flávia', 'Rubens', 'Bruna'}
...     ESP = {'Caio', 'Artur', 'Beatriz', 'Carolina', 'Maria', 'Juliano', 'Bruna', 'Rui'}
...     FRA = {'Pedro', 'Bruna', 'Paula', 'Tiago', 'Maria', 'Flávia', 'Rui', 'Carolina'}
...
...     # 1 – Interseção entre os pares de turmas: (ING & ESP), (ING & FRA) e (ESP & FRA)
...     # 2 – Calcula a união das interseções
...     ALUNOS_DESCONTO = (ING & ESP) | (ING & FRA) | (ESP & FRA)
...
...     # Exibição do resultado
...     print('Relação de dos alunos com desconto:')
...     print(ALUNOS_DESCONTO)
```

```
Relação de todos os alunos da escola:
{'Rui', 'Juliano', 'Bruna', 'Flávia', 'Maria', 'Carolina', 'Caio'}
```

É importante ressaltar que a união das interseções é diferente de simplesmente realizar a interseção entre todas as turmas. Se fizermos essa operação, teremos a relação de alunos que estão matriculados em todas as turmas, conforme o exemplo abaixo:

```
>>>  ING = {'Gabriel', 'Caio', 'Maria', 'Ana', 'Juliano', 'Flávia', 'Rubens', 'Bruna'}
...  ESP = {'Caio', 'Artur', 'Beatriz', 'Carolina', 'Maria', 'Juliano', 'Bruna', 'Rui'}
...  FRA = {'Pedro', 'Bruna', 'Paula', 'Tiago', 'Maria', 'Flávia', 'Rui', 'Carolina'}
...
...  # 1 – Interseção entre os pares de turmas: (ING & ESP), (ING & FRA) e (ESP & FRA)
...  # 2 – Calcula a união das interseções
...  ALUNOS_DESCONTO = (ING & ESP) | (ING & FRA) | (ESP & FRA)
...
...  # Exibição do resultado
...  print('Relação de dos alunos com desconto:')
...  print(ALUNOS_DESCONTO)
Relação de todos os alunos da escola:
{'Bruna', 'Maria'}
```

Para concluir, se fossemos resolver estes problemas utilizando apenas listas, teríamos que escrever uma porção de blocos de códigos com estruturas condicionais e estruturas de repetição, além de utilizar outros métodos auxiliares. Portanto, o tipo `set` tem utilidade para determinados problemas, como acabamos de ver.

Dicionários

As outras estruturas de dados que vimos até o momento são formadas por coleções de itens, ordenadas ou não. Em Python, um dicionário, que é do tipo de dados `dict`, também é uma coleção de itens, entretanto cada elemento é um par key – value (*chave – valor*). Estes pares indicam que cada elemento possuí um *valor*/*value* atrelado à uma *chave/key*.

Nas listas, tuplas e *strings*, acessamos os elementos da sequência por meio de um índice (posição). Nos dicionários isso ocorre de forma diferente, pois o acesso aos valores (*value*) dos itens é realizado por meio de uma chave (*key*) que o identifica. O nome dicionário é uma analogia aos dicionários da vida real, que mapeiam, para cada termo (*key*) um significado (*value*). Dessa forma, em Python, os dicionários são estruturas de dados que nos permitem mapear chaves à valores.

Assim como os conjuntos (`set`), a criação de estruturas do tipo `dict` também utiliza chaves ({ e }), entretanto cada elemento deve possuir uma chave (`key`) e um valor (`value`) correspondente declarados como `key: value`. Enquanto os valores podem ser de qualquer tipo de dado suportado pelo Python, as chaves (`key`) precisam ser únicas e imutáveis, como os tipos numéricos *strings* e tuplas[27]. Vejamos alguns exemplos de criação de dicionários.

```
>>>  # Dicionário onde as chaves são do tipo string
...  d1 = {'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
...  print(d1)
...
...  # Dicionário onde as chaves são do tipo inteiro
...  d2 = {2: 'dois', 1: 'um', 4: 'quatro', 3: 'três', 0: 'zero'}
...  print(d2)
...
...  # Dicionário com chaves de tipos mistos
...  d3 = {2: 'a', 5.44: True, 'key': None}
...  print(d3)
...
...  # Dicionário vazio
...  d4 = {}
...  print(d4)
{'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
{2: 'dois', 1: 'um', 4: 'quatro', 3: 'três', 0: 'zero'}
{2: 'a', 5.44: True, 'key': None}
{}
```

[27] Como citado anteriormente, uma das vantagens da utilização das tupla em relação às listas é o fato delas poderem ser utilizadas com chaves em dicionários, uma vez que são imutáveis.

O acesso aos elementos é realizado de forma semelhante às listas e tuplas, por meio do operador de acesso com colchetes ([ ]). Mas ao invés de indicarmos o índice do elemento, indicaremos a chave do elemento desejado.

```
>>>    # Criação dos dicionários
...    d1 = {2: 'dois', 1: 'um', 4: 'quatro', 3: 'três', 0: 'zero'}
...    d2 = {'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
...
...    # Acesso aos elementos
...    print(d1[0])
...    print(d1[2])
...
...    print(f'Meu nome é {d2["nome"]} e tenho {d2["idade"]} anos')
zero
dois
Meu nome é Antônio e tenho 36 anos
```

Se tentarmos acessar um elemento por meio de uma chave inválida, teremos um erro de execução alertando que a chave não existe. Existe também um método de acesso (dic.get(key)), que ao invés de retornar um erro ele retorna um valor nulo para chaves inexistentes.

```
>>>    # Criação do dicionário
...    d1 = {'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
...
...    # Acesso por meio do operador []
...    print(d1['endereço'])
KeyError                          Traceback (most recent call last)
<ipython-input-9-7ef8f119ebea> in <module>()
      1 d1 = {2: 'dois', 1: 'um', 4: 'quatro', 3: 'três', 0: 'zero'}
----> 2 print(d1['endereço'])

KeyError: 'endereço'
>>>    # Criação do dicionário
...    d1 = {'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
...
...    # Acesso por meio do método get()
...    print(d1.get('endereço'))
None
```

Os dicionários são tipos mutáveis e por isso podemos adicionar novos itens ou alterar o valor dos itens atuais, utilizando também o operador [ ]. Se a chave indicada já

estiver presente na estrutura, o valor mapeado para esta chave será atualizado. No caso de não estar presente, um novo par (key: value) será adicionado ao dicionário.

```
>>>  # Criação do dicionário
...  d1 = {'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}
...  print(d1)
...
...  # Atualização do valor da chave 'nome'
...  d1['nome'] = 'Antônio Carlos'
...  print(d1)
...
...  # Adição da chave 'endereço'
...  d1['endereço'] = 'Rua 123'
...  print(d1)
{'nome': 'Antônio', 'idade': 36, 'sexo': 'masculino'}

{'nome': 'Antônio Carlos', 'idade': 36, 'sexo': 'masculino'}

{'nome': 'Antônio Carlos', 'idade': 36, 'sexo': 'masculino', 'endereço': 'Rua 123'}
```

Assim como todas as outras estruturas de dados, o tipo dict também possuí operadores, funções e métodos que operam seus elementos. Enquanto algumas destas operações se comportam da mesma forma que nas demais estruturas, outras possuem pequenas diferenças:

- As funções min() e max() continuam retornando o *mínimo* e o *máximo*, respectivamente. Entretanto, a operação é realizada somente nas chaves da estrutura, ignorando os seus valores.

- Os operadores de filiação in e not in também operam sobre as chaves, e não sobre os valores.

- O método pop(key) remove o elemento com a chave key. Para remover um elemento arbitrário existe o método popitem().

Vejamos exemplos destas e outras operações do tipo dict:

```
>>>  # Cria o dicionário
```

```
...     d1 = {'zero': 0, 'um': 1, 'dois': 2, 'três': 3, 'quatro': 4}
...     print(d1)
...
...     # Encontra a maior e menor chave
...     print('Maior e menor chave:', max(d1), min(d1))
...
...     # Adiciona elementos de um outro dicionario
...     d1.update({'cinco': 5, 'seis': 6})
...     print(d1)
...
...     # Verifica se o dicionário possui as seguintes chaves
...     print("A chave 'dois' está no dicionário?", 'dois' in d1)
...     print("A chave 'cinco' não está no dicionário?", 'dois' in d1)
...
...     # Remove o elemento com chave 'zero'
...     d1.pop('zero')
...     print(d1)


{'zero': 0, 'um': 1, 'dois': 2, 'três': 3, 'quatro': 4}

Maior e menor chave: zero dois

{'zero': 0, 'um': 1, 'dois': 2, 'três': 3, 'quatro': 4, 'cinco': 5, 'seis': 6}

A chave 'dois' está no dicionário? True
A chave 'cinco' não está no dicionário? True

{'um': 1, 'dois': 2, 'três': 3, 'quatro': 4, 'cinco': 5, 'seis': 6}
```

Da mesma forma que os outros tipos de estruturas, também é possível utilizar a instrução `for … in` para iterar sobre os elementos do dicionário. Entretanto, devemos nos atentar sobre o que queremos iterar e escolher a opção correta. Nos dicionários podemos iterar sobre as chaves (`dict.keys()`), sobre os valores (`dict.values()`) ou sobre os itens (`dict.items()`), no caso dos pares `key:value`. Quando não especificamos, a iteração será sobre as chaves por padrão.

Vejamos os exemplos a seguir para entendermos melhor este conceito.

```
>>>     # Cria o dicionário
...     d1 = {'zero': 0, 'um': 1, 'dois': 2, 'três': 3, 'quatro': 4}
...
...     # Itera sobre as chaves
...     for key in d1:
...         if key == 'três':
...             print('Chave três encontrada!')
...
...     for key in d1.keys():
```

```
...         if key == 'dez':
...             print('Chave quatro encontrada!')
...
...     # Itera sobre os valores
...     soma = 0
...     for value in d1.values():
...         soma = soma + value
...     print('Soma dos valores do dicionário:', soma)
...
...     # Itera sobre os itens
...     for key, value in d1.items():
...         print(key, value)
...
Chave três encontrada!

Chave quatro encontrada!

Soma dos valores do dicionário: 10

zero 0
um 1
dois 2
três 3
quatro 4
```

Capítulo 9. Funções em Python

Nos capítulos anteriores apresentamos diversos exemplos simples em Python, onde os resultados das computações eram armazenados em variáveis ou exibidos na tela. Entretanto, durante o desenvolvimento de um programa completo, iremos deparar com situações em que será necessário utilizar uma mesma computação múltiplas vezes, em diferentes partes do código.

Considere, por exemplo, um programa de gerenciamento financeiro para uma empresa, que para seu funcionamento necessitamos realizar uma série de operações de soma, como por exemplo: a soma das vendas diárias, a soma dos valores dos produtos em estoque e a soma dos valores devidos aos fornecedores. Também precisaremos ordenar a lista de vendas por cliente ou por valor, assim como ordenar a lista de fornecedores por nome ou por data de cadastro. Portanto é possível identificar duas tarefas recorrentes neste programa: somar valores e ordenar itens de uma lista.

Em casos como este do exemplo, o ideal é que tenhamos um procedimento capaz de somar uma série de valores e um outro que ordene os itens de uma lista, de acordo com algum critério. Desta forma esses procedimentos podem ser utilizados diversas vezes, sem a necessidade de repetir o mesmo trecho de código em diferentes etapas do programa. Ou seja, desejamos ter partes do código que sejam reutilizáveis!

Em Python, assim como em outras linguagens de alto nível, a reutilização de códigos é realizada por meio de *funções*. Uma função é basicamente um bloco de código, que realiza uma determinada tarefa e que pode ser reutilizado várias vezes. As funções auxiliam na divisão do programa em partes menores e modulares, e à medida que o código cresce cada vez mais, as funções o tornam mais organizado e gerenciável.

Apesar de estarem sendo introduzidas agora no curso, nós já utilizamos funções diversas vezes nos capítulos anteriores, como por exemplo a função `print()`, para exibição de mensagens, a função `range()`, que gera uma sequência de inteiros, e a função `len()`, que retorna o tamanho de uma sequência. Entretanto, neste capítulo serão discutidos os aspectos fundamentais das funções, assim como a sua criação e utilização.

Declaração de Funções

Nesta seção serão discutidos os aspectos da declaração das funções em Python. As funções são declaradas utilizando a seguinte sintaxe:

```
def nome_da_funcao(argumento1, argumento2, ...):
    <bloco de código a ser executado>
    return resultado
```

Sua sintaxe é formada pelos seguintes componentes:

- A palavra-chave `def`, que define o início da declaração de uma função.
- O identificador da função (`nome_da_funcao`), que deve ser único e que será utilizado para chamar esta função. Os nomes de funções devem seguir as mesmas regras dos nomes de variáveis, apresentados na Seção Regras para Nomeação de Variáveis.
- Os argumentos (`argumento1, argumento2, ...`), que são opcionais, são utilizados para repassar valores paras as funções. Mesmo que não for passado nenhum argumento para a função, é necessário declarar os parênteses (`nome_da_funcao()`).
- O tradicional dois pontos (`:`) para marcar o fim da declaração e o início de um bloco indentado.

- O <bloco de código>, que irá compor o corpo da função, é o conjunto de instruções responsável por realizar as computações que a função irá desempenhar.

- Por fim, uma instrução de retorno (return), que é responsável por retornar o resultado da função. Uma vez que existem funções que não retornam valores, sua declaração é opcional.

Para exemplificar a definição e a utilidade das funções, vamos considerar o problema de somar um conjunto de números e que o mesmo procedimento deve ser realizado para três sequências de números diferentes. Tradicionalmente, como exemplificado no Capítulo 4 e no Capítulo 5, podemos criar um programa que utiliza as instruções de repetição (for ... in) em conjunto com a estrutura de dados do tipo lista (list):

```
>>>  # define as listas com os números a serem somados
...  l1 = [1, 2, 3, 4, 5]
...  l2 = [3, 1, 5, 9, 0, 8, 2, 3, 4]
...  l3 = [12, 43, 23, 12, 789]
...
...  # itera sobre cada lista e soma seus elementos
...  soma_l1 = 0
...  for item in l1:
...      soma_l1 = soma_l1 + item
...
...  soma_l2 = 0
...  for item in l2:
...      soma_l2 = soma_l2 + item
...
...  soma_l3 = 0
...  for item in l3:
...      soma_l3 = soma_l3 + item
...
...  # exibe os resultados
...  print(f'Resultado: l1={soma_l1}, l2={soma_l2}, l3={soma_l3}')
Resultado: l1=15, l2=35, l3=879
```

Observe que para cada lista utilizamos o mesmo trecho de código para computar a sua soma, variando apenas as variáveis envolvidas na computação. Imagine

agora se a mesma tarefa fosse necessária para computar a soma de mil listas? Nós teríamos que escrever o mesmo bloco de código mil vezes! É justamente este retrabalho que as funções evitam. Desta forma, podemos reescrever o mesmo código utilizando funções:

```
>>>  # define as listas com os números a serem somados
...  l1 = [1, 2, 3, 4, 5]
...  l2 = [3, 1, 5, 9, 0, 8, 2, 3, 4]
...  l3 = [12, 43, 23, 12, 789]
...
...  # declaraca a função que soma os elementos da lista
...  def soma_lista(lista):
...      soma = 0
...      for item in lista:
...          soma = soma + item
...      return soma
...
...  # chama a função para cada lista
...  soma_l1 = soma_lista(l1)
...  soma_l2 = soma_lista(l2)
...  soma_l3 = soma_lista(l3)
...
...  print(f'Resultado: l1={soma_l1}, l2={soma_l2}, l3={soma_l3}')
Resultado: l1=15, l2=35, l3=879
```

Utilização de Funções

A seguir serão apresentados mais alguns exemplos de utilização, juntamente com alguns outros princípios sobre funções. Vamos considerar o seguinte código, onde as funções são utilizadas diversas vezes e na ordem que desejarmos:

```
>>>  # exibe mensagem de boas vindas à uma pessoa
...  def boas_vindas(nome):
...      print(f'Olá {nome}. Seja bem-vindo (a)!')
...
...  # calcula a área de um quadrado: l x l
...  def area_quadrado(lado):
...      return lado * lado
...
...  # calcula a área de um triângulo: (b x h) / 2
...  def area_triangulo(base, altura):
...      return (base * altura)/2
...
...  # Realiza as chamadas das funções
```

```
...     boas_vindas('Priscila')
...     print(area_triangulo(4, 5))
...
...     boas_vindas('Maria')
...     boas_vindas('Joana')
...
...     print(area_quadrado(4))
...     print(area_quadrado(10))
...
...     boas_vindas('Antônio')
...     print(area_quadrado(10))
...
...     print(area_triangulo(5, 2))
...     print(area_triangulo(4, 5))
Olá Priscila. Seja bem-vindo (a)!
10.0
Olá Maria. Seja bem-vindo (a)!
Olá Joana. Seja bem-vindo (a)!
16
100
Olá Antônio. Seja bem-vindo (a)!
100
5.0
10.0
```

A primeira função (boas_vindas(nome)) é um exemplo de função que não retorna um valor e por isso não faz uso da palavra-chave return. Esta função apenas exibe uma mensagem, utilizando como parte do texto uma string (nome) fornecida como argumento da função.

As outras duas funções do exemplo, area_quadrado(lado) e area_triangulo(base, altura), são capazes de calcular e retornar a área do quadro e do triangulo, respectivamente. Para a área do quadrado basta apenas um argumento (o tamanho do lado), enquanto, para a área do triangulo são necessárias duas medidas, a base e a altura do triangulo (ou seja, dois argumentos).

Vamos considerar agora outro exemplo, que apresenta diferentes conceitos das funções em Python.

```
>>>   # Realiza uma divisão. Se o divisor é zero, retorna uma mensagem de erro.
...     def div(dividendo, divisor):
...         if divisor == 0:
```

```
...             print('ERRO: Divisor igual à zero!')
...             return
...         return dividendo / divisor
...
...     # Função similar à função div, mas que retorna o dividendo e o resto da divisão.
...     def div_qr(dividendo, divisor):
...         if divisor == 0:
...             print('ERRO: Divisor igual à zero!')
...             return
...         quociente =  dividendo // divisor
...         resto = dividendo % divisor
...         return (quociente, resto)
...
...     print('div(10, 4) ==>', div(10, 4)) # dividento=10 e divisor=4
...     print('div_qr(10, 4) ==>', div_qr(10, 4)) # dividento=10 e divisor=4
...     print('div(10, 0) ==>', div(10, 0)) # dividento=10 e divisor=0
div(10, 4) ==> 2.5

div_qr(10, 4) ==> (2, 2)

ERRO: Divisor igual à zero!
div(10, 0) ==> None
```

A primeira função `div(dividendo, divisor)`, realiza a operação de divisão do valor do `dividendo` pelo valor do `divisor`. Antes de efetuar a operação, a função verifica se o divisor é igual à zero, algo que o operador de divisão (`/`) por si só não faz. Portanto, se o divisor for igual a zero (ou seja, condição `divisor == 0` verdadeira), a função exibe uma mensagem de erro e “aborta” em seguida por meio do comando `return`.

Observe que mesmo sem retornar um valor, o comando `return` pode ser utilizado para abortar a execução da função, semelhante ao comando `break` das instruções de repetição.

Apesar de não ter um valor explicitamente declarado, o comando `return`, quando utilizado sozinho, retorna `None`. Assim, ele equivale ao mesmo que declarar `return None`. Quando a função foi utilizada com os argumentos `dividendo=10` e `divisor=0`, a mensagem de erro foi exibida e ela foi abortada, mas o valor `None` foi retornado. Por este motivo foi exibido o valor `None` (`'div(10, 0) ==> None'`) na saída do programa:

```
>>>    print('div(10, 0) ==>', div(10, 0)) # dividento=10 e divisor=0

ERRO: Divisor igual à zero!
div(10, 0) ==> None
```

A segunda função, div_qr(dividendo, divisor), é similar à primeira, com a diferença que ela retorna dois valores: o quociente e o resto da divisão. Isto ilustra um conceito bem útil das funções em Python, que é o retorno de múltiplos valores, em forma de uma tupla (tuple). É possível, também, retornar os múltiplos valores da função em uma única variável tipo tuple, ou então "desempacotar" os múltiplos valores em diferentes variáveis, cada uma com o tipo correspondente ao valor retornado.

```
>>>    # atribuição dos múltiplos valores em uma variável única do tipo tupla
...    resultado = div_qr(21, 5)
...    print('resultado:', resultado, type(resultado))
...
...    # atribuição dos múltiplos valores em variáveis separadas
...    quociente, resto = div_qr(21, 5)
...    print('quociente:', quociente, type(quociente))
...    print('resto:', resto, type(resto))

resultado: (4, 1) <class 'tuple'>

quociente: 4 <class 'int'>
resto: 1 <class 'int'>
```

Para finalizar a análise das funções div e div_qr, observe que as duas possuem um trecho de código em comum:

```
if divisor == 0:
   print('ERRO: Divisor igual à zero!')
   return
```

Se o objetivo das funções é reaproveitar código, por que não transformar estes trechos repetidos em uma nova função? Desta forma, pode-se ter funções utilizando outras funções:

```
# Verifica se é um divisor inválido (divisor == 0)
def divisor_invalido(divisor):
   if divisor == 0:
      print('ERRO: Divisor igual à zero!')
```

```
        return True
    return False

# Retorna o resultado de uma divisão
def div(dividendo, divisor):
    if divisor_invalido(divisor):
        return
    return dividendo / divisor

# Retorna o quociente e o resto de uma divisão
def div_qr(dividendo, divisor):
    if divisor_invalido(divisor):
        return
    quociente =  dividendo // divisor
    resto = dividendo % divisor
    return (quociente, resto)
```

Argumentos das Funções

Nas seções anteriores aprendemos como criar uma função e como utilizá-la. Vimos também a passagem de valores para as funções são realizadas por meio de *argumentos*. Iremos agora conhecer algumas particularidades sobre os tipos de argumentos que o Python suporta. Para tanto, antes consideremos a seguinte função:

```
>>>    # calcula a área de um triângulo: (b x h) / 2
...    def area_triangulo(base, altura):
...        return (base * altura)/2
...
...    print(area_triangulo(5, 10))
25
```

Essa função possuí dois argumentos e a utilizamos com dois valores (`5` e `10`). Desta forma, como passamos o número exato de argumentos, ela executou normalmente, sem erros.

Observe que ao chamar a função com os valores *5* e *10* (`area_triangulo(5, 10)`), estamos atribuindo o valor `5` ao argumento `base` e o valor `10` ao argumento `altura`. Isso ocorre porque estamos passando os valores desejados na mesma ordem que os argumentos foram declarados na função.

Seria possível também passar os valores dos argumentos em uma ordem diferente, mas neste caso precisamos declarar o nome dos argumentos na chamada da função, conforme o exemplo a seguir.

```
>>>    # calcula a área de um triângulo: (b x h) / 2
...    def area_triangulo(base, altura):
...        return (base * altura)/2
...
...    print(area_triangulo(altura=10, base=5))
25
```

Entretanto, se passarmos um número diferente de argumentos, o interpretador do Python irá retornar um erro. Por exemplo, vamos tentar utilizar a função com apenas um argumento.

```
>>>    # calcula a área de um triângulo: (b x h) / 2
...    def area_triangulo(base, altura):
...        return (base * altura)/2
...
...    print(area_triangulo(5))
TypeError                          Traceback (most recent call last)
<ipython-input-42-175e1116e603> in <module>()
    3    return (base * altura)/2
    4
----> 5 print(area_triangulo(5))

TypeError: area_triangulo() missing 1 required positional argument: 'altura'
```

Até o momento aprendemos a declarar funções com um número fixo de argumentos. Entretanto, o Python provê outras maneiras de definirmos uma função que pode ter um número variável de argumentos. Vejamos a seguir uma maneira muito utilizada pelos desenvolvedores, que são os argumentos predefinidos, também conhecidos como argumentos *default*.

Esse conceito permite que os argumentos de uma função possam ter um valor *default*. Podemos atribuir um valor padrão para um argumento por meio do operador de atribuição (=), conforme o exemplo a seguir.

```
>>>    def exibe_pessoa(nome, idade=30):
...        print(f'Meu nome é {nome} e tenho {idade} anos.')
...
...    exibe_pessoa('Antônio')
...    exibe_pessoa('Antônio', 36)

Meu nome é Antônio e tenho 30 anos.
Meu nome é Antônio e tenho 36 anos.
```

Observem que definimos uma função com dois argumentos (`nome` e `idade`), e definimos um valor padrão para o segundo argumento (`idade=30`). Desta forma, podemos utilizar a função passando apenas o argumento obrigatório (`nome`), como na primeira chamada da função. Neste caso, será atribuído o valor padrão (*30*) ao valor do argumento `idade`. Ou podemos utilizar a função com os dois argumentos definidos, como foi o caso da segunda chamada.

Capítulo 10. Módulos

Até agora utilizamos apenas conceitos nativos do Python, ou seja, funcionalidades que são oferecidas por padrão pela linguagem, sem a utilização de códigos externos. Também aprendemos como utilizar funções para a reutilização de nossos códigos em formas de rotinas já predefinidas.

Agora chegou a hora de aprendermos a utilizar funções e bibliotecas que foram escritas por outros desenvolvedores. Já ouviram a expressão "reinventar a roda"? Pois é, no mundo do desenvolvimento não existe nada menos produtivo do que reimplementar funcionalidades que já existem e que poderiam lhe atender muito bem, principalmente quando se trata de uma biblioteca que está em desenvolvimento há muito tempo e que já foi testada e otimizada por vários desenvolvedores.

Em Python, podemos reaproveitar estes códigos por meio de *módulos*. Portanto, veremos a seguir do que se tratam estes módulos e como podemos utilizá-los. Adicionalmente, conheceremos os módulos já embutidos do Python, assim como instalar novos módulos.

Criação e Importação de Módulos

Um módulo em Python nada mais é do que um arquivo texto contendo códigos com declarações de variáveis e ou funções. Estes arquivos de módulos devem sempre ser salvos com a extensão .py, que simboliza um arquivo do Python.

Nós utilizamos módulos para poder organizar (modularizar) os nossos códigos de acordo com as funcionalidades de cada conjunto de funções. Por exemplo, se estivéssemos desenvolvendo um programa para geometria, poderíamos ter um módulo para cada figura geométrica (quadrado, triangulo, círculo etc.) ou então um módulo para

cada tipo de cálculo geométrico (área, volume, perímetro etc.). A organização e a divisão dos códigos em módulos dependerão do seu objetivo com o programa. O importante é ter em mente que a criação dos módulos é essencial para a modularização dos programas, principalmente aqueles com milhares de linhas de código.

Por exemplo, vamos criar um módulo, chamado *areas* que contém funções básicas para o cálculo da área de diferentes figuras geométricas. Para tanto, primeiro precisamos criar um arquivo e em seguida salvá-lo com o nome `areas.py`. Em seguida, basta adicionarmos o conteúdo do módulo, que são as funções que irão realizar os cálculos:

```
# Define o valor de PI
PI = 3.141592

# Calcula a área do quadrado
def quadrado(l):
    return l ** 2

# calcula a área de um triângulo
def triangulo(b, h):
    return (b * h)/2

# calcula a área de um círculo
def circulo(r):
    return PI * (r ** 2)

# calcula a área de uma elipse
def circulo(a, b):
    return PI * a * b

# calcula a área de um trapézio
def trapezio(B, b, h):
    return  (B + b) * h / 2
```

Voila! Já temos o nosso primeiro módulo e agora basta utilizarmos ele para calcular as áreas de nossas figuras geométricas. Em Python, a utilização de códigos externos é realizada por meio da *importação de módulos*, com a sintaxe: `import <modulo>`. Portanto, para importamos o nosso módulo de exemplos, devemos utilizar a instrução:

```
>>> import areas
```

Entretanto, a importação não significa que já importamos todos os nomes de funções e variáveis declaradas no módulo diretamente. Nós apenas importamos o nome do areas, e a partir deste nome podemos acessaras as declarações contidas no módulo:

```
>>> import areas
... areas.triangulo(5, 8)
20.0
```

Se desejássemos importar todas as declarações, seria necessário a importação do código abaixo. Desta forma, teremos acesso direto a todos as declarações do módulo, inclusive a variável PI.

```
>>> from areas import *
... triangulo(5, 8)
20.0
>>> quadrado(6)
36
>>> PI
3.141592
```

O comando from … imort possibilita a importação de todas as declarações de um módulo, como o exemplo anterior, ou a importação de algumas declarações específicas, como o exemplo a seguir. Desta forma, apenas as funções quadrado() e circulo() estarão disponíveis para utilização no código.

```
>>> from areas import quadrado, circulo
```

Entretanto, existem situações em que a importação de declarações diretamente não é recomendada, pois, ao fazer isso, perdemos a rastreabilidade de qual módulo está provendo uma função. Considere, por exemplo, que tivéssemos dois módulos para figuras geométricas, um que calcula áreas e outro que calcula perímetros,

e dentro destes módulos as funções são nomeadas de acordo com as figuras (da mesma forma que fizemos no arquivo areas.py). Se fizermos a importação direta das funções dos dois módulos, perderemos a referência de qual função faz parte de cada módulo.

Também é possível atribuirmos um novo nome para o módulo quando realizamos a importação, da seguinte forma:

```
>>>  import areas as ar
...  ar.triangulo(5, 8)
20.0
>>>  ar.quadrado(6)
36
```

Módulos Embutidos do Python

O Python possuí vários módulos já embutidos na linguagem, que oferecem diversas funcionalidades úteis ao desenvolvedor. Estes módulos são desenvolvidos e mantidos pela comunidade responsável pela linguagem e são exaustivamente testados e otimizados antes de serem lançados aos usuários finais. A seguir iremos exemplificar alguns dos módulos embutidos mais conhecidos e utilizados pelos desenvolvedores[28].

```
>>>  # modulo com funções matemáticas para cálculos mais complexos
...  import math
...  print('Função cosseno:', math.cos(100))
...  print('Função log:', math.log(10))
Função cosseno: 0.8623188722876839
Função log: 2.302585092994046

>>>  # modulo para construção de sequências elaboradas
...  import itertools
...  print(list(itertools.combinations('ABCD', 3))) # combinação de 3 em 3 print(list(itertools.permutations(['a', 'b', 'c'], 2))) #
...  permutação de 2 em 2
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'C', 'D'), ('B', 'C', 'D')]
```

[28] A lista completa de todos os módulos embutidos pode ser encontrada na documentação oficial do Python (PSF, 2022): https://docs.python.org/3/py-modindex.html

```
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c', 'a'), ('c', 'b')]


>>>   # modulo para manipulação de timestamps (datas, horários, deltas etc.)
...   from datetime import datetime, timedelta
...   print('Datetime atual:', datetime.now())
...   print('Datetime após 7 dias:',  datetime.now() + timedelta(days=7))

Datetime atual: 2022-03-07 12:51:28.646004
Datetime após 7 dias: 2022-03-14 12:51:28.646878


>>>   # modulo para criação de números e sequências randômicas
...   import random
...   print('Numero aleatório entre 0 e 1:', random.random())
...   print('Inteiro aleatório entre 50 e 100:', random.randint(50, 100))

Numero aleatório entre 0 e 1: 0.7383173846470767
Inteiro aleatório entre 50 e 100: 91


>>>   # modulo para funcionalidades que dependem do sistema operacional
...   import os
...   os.mkdir('pasta') # cria um diretório chamado pasta
...   print('Caminho completo:', os.path.join('/home/antonio', 'pasta', 'arquivo.txt'))

Caminho completo: /home/antonio/pasta/arquivo.txt
```

Instalação de Novos Módulos

Além dos módulos embutidos do Python, existem diversos outros módulos que são desenvolvidos e mantidos por terceiros. Até você pode criar o seu próprio módulo e em seguida disponibilizá-lo para que os outros desenvolvedores possam utilizá-lo. O módulo, ou um conjunto deles, pode ser disponibilizado na forma de *packages*[29], "empacotando" os códigos em um arquivo e em seguida fazendo o upload para algum repositório de bibliotecas em Python.

O repositório oficial de pacotes do Python é o *pypi*[30], que atualmente possuí mais de 360 mil pacotes disponíveis para download. Entretanto, durante o

[29] Tutorial oficial com um guia sobre o empacotamento de códigos em Python: https://packaging.python.org/en/latest/tutorials/packaging-projects/

[30] Repositório oficial de pacotes Python: https://pypi.org/

desenvolvimento de uma aplicação do mundo real e dependendo do tipo desta aplicação, não iremos utilizar uma média de 10 a 30 pacotes que precisarão ser instalados destes repositórios oficiais.

No curso estamos utilizando o *Google Colab* e ele já possui vários pacotes extras instalados por padrão. Entretanto, se você estiver utilizando o Python instalado em seu computador ou se estiver escrevendo uma aplicação real, muito provavelmente você precisará instalar pacotes extras no seu ambiente *Python*.

A principal forma de instalar um pacote é utilizando o programa *pip*, que é capaz de instalar qualquer pacote do repositório *pypi* por meio do comando a seguir:

```
pip install <pacote>
```

Por exemplo, um pacote muito utilizado pelos cientistas de dados é o *pandas*[31], que é uma biblioteca para manipulação e análise de dados tabulares. Podemos instalar a versão mais recente do pacote da seguinte forma:

```
pip install pandas
```

Se necessário podemos também especificar uma versão específica do pacote para a instalação:

```
pip install pandas==1.3.5
```

Também é possível atualizar um pacote para a sua versão mais atual, utilizando o pip:

```
pip install --upgrade pandas
```

[31] Página oficial do pacote *pandas*: https://pandas.pydata.org/

É importante ressaltar que o *pip* é um programa a parte do Python. Ou seja, este comando deve ser executado no seu terminal de comandos e não dentro de uma execução interativa do Python. Geralmente, as instalações padrões do Python já trazem consigo o *pip* instalado, entretanto se for necessário instalar o *pip*, basta seguir o tutorial[32] de instalação dele.

Por fim, se for necessário instalar um pacote no *Google Colab*, podemos fazer uma chamada de comando do sistema direto das células dele. Para isso, basta indicarmos que o comando é do sistema com um ponto de exclamação no início:

```
>>>  ! pip install igpu

Collecting igpu
  Downloading igpu-0.1.2-py3-none-any.whl (15 kB)
Collecting pynvml
  Downloading pynvml-11.4.1-py3-none-any.whl (46 kB)
     |████████████████████████████████| 46 kB 1.7 MB/s
Requirement already satisfied: psutil in /usr/local/lib/python3.7/dist-packages (from igpu) (5.4.8)
Installing collected packages: pynvml, igpu
Successfully installed igpu-0.1.2 pynvml-11.4.1
```

[32] Tutorial de instalação pip: https://pip.pypa.io/en/stable/installation/

Capítulo 11. Manipulação de Arquivos

Até o momento os dados utilizados nos exemplos anteriores foram manipulados diretamente na memória do computador. Entretanto, em muitas situações teremos que ler e gravar dados que estarão armazenados em um arquivo do disco. Existem diversas soluções que exigem a manipulação dos dados em disco, como por exemplo:

- **Análise inteligente de imagens:** as imagens poderão ser geradas por meio de uma câmera e em seguida armazenadas no disco rígido do computador, para uma posterior análise.

- **Gerenciador financeiro:** pode ser necessária a leitura de arquivos no disco que contém uma lista de transações e/ou planilhas de cálculos do balanço financeiro de uma empresa.

- **Processador de texto:** poderíamos criar um editor de texto, capaz de abrir, alterar e salvar as edições em um arquivo no disco.

Neste capítulo será apresentada as principais operações do Python para a manipulação de arquivos. O conteúdo aqui apresentado é apenas introdutório, para que possamos entender os conceitos básicos e possibilitar, assim, um estudo mais amplo sobre os tópicos acerca da manipulação de arquivos em Python.

Existem diferentes formatos de arquivos como *CSV*, *JSON*, *XML*, *YML* e vários outros. Para cada um destes formatos o ideal é utilizar um pacote Python desenvolvido especificamente para a manipulação de seus dados. Entretanto, todos esses pacotes, muito provavelmente, devem utilizar os conceitos básicos de manipulação de arquivos que iremos apresentar a seguir.

Criação, Abertura e Fechamento de Arquivos

A criação de arquivos (e, consequentemente, a abertura deles), é realizada por meio da função `open(arquivo, modo)`, que permite criar e/ou abrir um arquivo com o nome (`arquivo`) especificado como argumento.

Também é possível determinar o modo de uso, por meio de um segundo parâmetro (`modo`). Existem diferentes modos de uso, conforme a tabela abaixo:

Modo	Descrição
'r'	Modo somente leitura (modo padrão).
'w'	Modo de escrita. Cria um arquivo, caso ainda não exista, ou substitui o arquivo atual.
'x'	Modo de escrita. Cria um arquivo e, se o arquivo já existir, retorna um erro.
'a'	Modo de escrita. Cria um arquivo, caso ainda não exista. Se o arquivo já existir, novas escritas serão adicionadas ao final dele.
't'	Abre o arquivo no modo texto (modo padrão).
'b'	Abre o arquivo no modo binário.

Por exemplo, para abrir um arquivo chamado `valores.txt`, em modo leitura apenas, utiliza-se o seguinte comando:

```
arquivo_valores = open('valores.txt', 'r')
```

Entretanto, para abrir o mesmo arquivo, em modo de escrita e como binário, o comando será:

```
arquivo_valores = open('valores.txt', 'wb')
```

Ao final da utilização, é necessário fechar o arquivo. Para isso, é utilizado o método `close()` do arquivo aberto anteriormente. Se o arquivo não for fechado corretamente, poderá causar a perda de dados, assim como corromper o arquivo.

```
arquivo_valores.close()
```

Leitura de Arquivos

A seguir, será utilizado como exemplo um arquivo contendo dados das maiores cidades do Brasil. Cada linha conterá os dados de uma cidade, no seguinte formato:

```
<estado>; <nome da cidade>; <população>;
```

Apesar de ser possível digitarmos estes dados no código do programa, podemos imaginar que, além de ser uma tarefa trabalhosa, também seria uma fonte de erros. Adicionalmente, esta lista pode ser alterada conforme os anos se passem e seria necessário atualizarmos o código do programa a cada nova alteração. Portanto a importância de se utilizar arquivos externos.

O nome do arquivo de exemplo é `cidades.txt` e possuí a lista das 15 maiores cidades do país[33], conforme ilustrado a seguir:

```
SP; São Paulo; 11895893
RJ; Rio de Janeiro; 6453682
BA; Salvador; 2902927
DF; Brasília; 2852372
CE; Fortaleza; 2571896
MG; Belo Horizonte; 2491109
AM; Manaus; 2020301
PR; Curitiba; 1864416
PE; Recife; 1608488
RS; Porto Alegre; 1472482
PA; Belém; 1432844
GO; Goiânia; 1412364
SP; Guarulhos; 1312197
SP; Campinas; 1154617
MA; São Luís; 1064197
```

[33] Dados extraídos de: https://exame.com/brasil/as-200-cidades-mais-populosas-do-brasil/

Existem três maneiras simples de lermos os dados de um arquivo. A primeira é a leitura de todas as linhas do arquivo, de uma única vez, e armazenando o conteúdo em uma variável do tipo *string*. Para tanto, utiliza-se o método `read()`:

```
>>>  arquivo = open('cidades.txt', 'r')
...  linhas = arquivo.read()
...  arquivo.close()
...  print(linhas)

SP; São Paulo; 11895893
RJ; Rio de Janeiro; 6453682
BA; Salvador; 2902927
DF; Brasília; 2852372
CE; Fortaleza; 2571896
MG; Belo Horizonte; 2491109
...
SP; Guarulhos; 1312197
SP; Campinas; 1154617
MA; São Luís; 1064197
```

Outra maneira é a leitura das linhas do arquivo por meio do método `readlines()`. Este método retorna uma lista de *strings*, onde cada elemento da lista corresponde à uma linha do arquivo:

```
>>>  arquivo = open('cidades.txt', 'r')
...  linhas = arquivo.readlines()
...  arquivo.close()
...  print(linhas)

['SP; São Paulo; 11895893\n', 'RJ; Rio de Janeiro; 6453682\n', 'BA; Salvador; 2902927\n', 'DF; Brasília; 2852372\n', 'CE; Fortaleza; 2571896\n', 'MG; Belo Horizonte; 2491109\n', 'AM; Manaus; 2020301\n', 'PR; Curitiba; 1864416\n', 'PE; Recife; 1608488\n', 'RS; Porto Alegre; 1472482\n', 'PA; Belém; 1432844\n', 'GO; Goiânia; 1412364\n', 'SP; Guarulhos; 1312197\n', 'SP; Campinas; 1154617\n', 'MA; São Luís; 1064197']
```

Observe que este método inclui a *quebra de linha* (`'\n'`) para cada item da lista, sendo necessário o tratamento posterior para removê-las. Por exemplo, pode-se criar outra lista (`novas_linhas`) onde, para cada elemento da lista anterior, fosse aplicado o método `rstrip()`[34]:

[34] Como visto no Capítulo 3, o método `rstrip()` remove caracteres em branco e quebras de linha ao final de uma *string*.

```
>>>    novas_linhas = []
...    for linha in linhas:
...        novas_linhas.append(linha.rstrip())
...    print(novas_linhas)
```

```
['SP; São Paulo; 11895893', 'RJ; Rio de Janeiro; 6453682', 'BA; Salvador; 2902927', 'DF; Brasília; 2852372', 'CE; Fortaleza; 2571896', 'MG; Belo Horizonte; 2491109', 'AM; Manaus; 2020301', 'PR; Curitiba; 1864416', 'PE; Recife; 1608488', 'RS; Porto Alegre; 1472482', 'PA; Belém; 1432844', 'GO; Goiânia; 1412364', 'SP; Guarulhos; 1312197', 'SP; Campinas; 1154617', 'MA; São Luís; 1064197']
```

Por fim, a última maneira é iterar sobre as linhas do arquivo, da seguinte forma:

```
>>>    arquivo = open('cidades.txt', 'r')
...    for linha in arquivo:
...        print(linha.rstrip())
...    arquivo.close()
```

```
SP; São Paulo; 11895893
RJ; Rio de Janeiro; 6453682
...
SP; Guarulhos; 1312197
SP; Campinas; 1154617
MA; São Luís; 1064197
```

Escrita de Arquivos

Considere novamente o mesmo arquivo das maiores cidades do Brasil (`cidades.txt`), apresentado na seção anterior. Desejamos agora adicionar as informações das próximas cidades no final deste arquivo. Para tanto, podemos utilizar dois métodos. Mas antes é necessário abrirmos o arquivo em modo de escrita e de adição de dados no seu final (modo `'a'`)

O primeiro método é o `write(texto)`, que recebe como parâmetro uma *string* contendo o texto a ser inserido no final do arquivo. Este método será utilizado para inserir a próxima cidade da lista: `RJ; São Gonçalo; 1031903`.

```
arquivo = open('cidades.txt', 'a')
arquivo.write('RJ; São Gonçalo; 1031903\n')
arquivo.close()
```

O segundo método, writelines(linhas), possibilita a escrita de diversos textos de uma só vez, utilizando como parâmetro uma estrutura de dados que seja iterável (ex.: list ou tuple). Por exemplo, o trecho abaixo adiciona as próximas quatro cidades da lista:

```
linhas = [
    'AL; Maceió; 1005319\n',
    'RJ; Duque de Caxias; 878402\n',
    'RN; Natal; 862044\n',
    'MS; Campo Grande; 843120\n'
]
arquivo = open('cidades.txt', 'a')
arquivo.writelines(linhas)
arquivo.close()
```

Observe que, para ambos os métodos, foi necessário adicionar a *quebra de linha* ('\n') ao final de cada nova linha a ser adicionada no arquivo.

Capítulo 12. Recursos Úteis da Linguagem

Estamos chegando ao final do curso e até agora aprendemos todos os conceitos fundamentais da linguagem Python, desde a declaração de variáveis, passando pelas estruturas de fluxo de controle e estruturas de dados, até a manipulação de arquivos.

A partir destes conceitos fundamentais, estaremos aptos a prosseguir com os estudos de conceitos intermediários e avançados, pois, ainda existem outros conceitos que não foram abordados aqui no curso e que poderão ser bastante úteis na nossa trajetória como desenvolvedores Python.

Não poderíamos finalizar o nosso curso sem antes conhecer um pouco destes recursos úteis que o Python oferece. Portanto, vamos conhecer alguns conceitos mais complexos da linguagem, mas que serão possíveis de compreendermos com todo o conhecimento que adquirimos até aqui. Desta forma poderemos deixar nossos códigos mais *pythonicos*[35] ainda.

Compreensão de Listas

Um dos recursos mais "queridos" dos desenvolvedores Python é o chamado compreensão de listas (*list comprenhesion*). É um conceito que permite otimizar a criação de novas listas e, de quebra, diminuir o número de linhas de código.

[35] A expressão *código pythonico* (derivado do inglês *pythonic*), apesar de ser vaga, é comumente utilizada na comunidade de desenvolvedores Python para referenciar um código legível, limpo e que faz uso frequente e correto dos recursos da linguagem. Apesar de não existir um guia oficial, muitos desenvolvedores consideram que um código *pythonico* deve seguir as regras do PEP 20 (também conhecido como *The Zen of Python*).

Este recurso é mais eficiente do que o modo tradicional de criação, pois requer um uso menor de memória e a operação é realizada em menos tempo.

A sintaxe básica da compreensão de listas é:

```
[expr(item) for item in sequencia]
```

Ou seja, para cada item da sequencia iterável aplica uma transformação expr. Por exemplo, tradicionalmente, para criar uma lista dos números de 1 a 10 elevado à potência 2, seria utilizado o seguinte bloco de código:

```
>>>   potencias = []
...   for item in range(1, 11):
...       potencias.append(item ** 2)
...   print(potencias)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A mesma operação pode ser reescrita da seguinte forma, utilizando compreensão de listas:

```
>>>   potencias = [item ** 2for item in range(1, 11)]
...   print(potencias)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Abaixo são apresentados mais alguns exemplos:

```
>>>   print( [n * 10 for n in range(1, 16)] ) # Multiplica por 10 os números de 1 a 15
...   print( [c.upper() for c in 'antonio'] ) # Cria lista com os caracteres em maísculo
...   print( [(n % 2 == 0) for n in range(0, 10)] ) # Indica se n é par ou não
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150]
['A', 'N', 'T', 'O', 'N', 'I', 'O']
[True, False, True, False, True, False, True, False, True, False]
```

Considere agora que, também, deverá ser criado uma lista dos números de 1 a 10 elevado à potência 2. Entretanto, apenas os números ímpares devem ser considerados e os números pares descartados. Ou seja, existe uma condição a ser

verificada (se o número é ímpar). Tradicionalmente, pode-se resolver este problema com o seguinte trecho de código:

```
>>>   potencias = []
...   for item in range(1, 11):
...     if item % 2 != 0:
...       potencias.append(item ** 2)
      print(potencias)
[1, 9, 25, 49, 81]
```

Observe que o código tem um fluxo condicional que verifica se o número é ímpar ou não (`if item % 2 != 0`). Por sorte, o recurso de compreensão de listas também permite que seja realizado essa verificação, com a seguinte sintaxe:

```
[expr(item) for item in sequencia if condicao]
```

Ou seja, caso `condicao` seja verdadeira, aplique a `expr` em cada `item` da `sequencia`. Desta forma, o código do problema anterior pode ser reescrito da seguinte forma:

```
>>>   potencias = [item ** 2 for item in range(1, 11) if item % 2 != 0]
...   print(potencias)
[1, 9, 25, 49, 81]
```

Compreensão de Dicionários

Também é possível aplicar o conceito de compreensão em dicionários. É o chamado *dict comprenhesion*. Com uma sintaxe bem semelhante à compreensão de listas, podem ser utilizados com ou sem a verificação condicional:

```
# dict comprenhesion sem condicional
{chave: valor for item in sequencia}

# dict comprenhesion com  condicional
{chave: valor for item in sequencia if condicao}
```

Observe que é utilizada chaves ({ }) ao invés de colchetes ([ ]), por serem o símbolo de definição dos dicionários em Python.

Considere o mesmo problema abordado anteriormente (números de 1 a 10 elevados a potência 2). Entretanto, deseja-se criar um dicionário onde a chave é o próprio número e o valor é o número elevado à potência 2. Abaixo, os dois trechos de código apresentam a solução tradicional, para todos os números e para apenas os números ímpares:

```
>>>  # todos os números elevado à potência 2
...  dict_todos = {}
...  for item in range(1, 11):
...      dict_todos[item] = item ** 2
...  print('Todos numeros:', dict_todos)
...
...  # apenas números ímpares elevado à potência 2
...  dict_impares = {}
...  for item in range(1, 11):
...      if item % 2 != 0:
...          dict_impares[item] = item ** 2
...  print('Números ímpares:', dict_impares)
Todos numeros: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
Números ímpares: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Utilizando compreensão de dicionários, o mesmo código anterior poderia ser escrito da seguinte forma:

```
>>>  # todos os números elevado à potência 2
...  dict_todos = {item: item ** 2 for item in range(1, 11)}
...  print('Todos numeros:', dict_todos)
...
...  # apenas números ímpares elevado à potência 2
...  dict_impares = {item: item ** 2 for item in range(1, 11) if item % 2 != 0}
...  print('Números ímpares:', dict_impares)
Todos numeros: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
Números ímpares: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

É unanimidade entre os desenvolvedores Python que os recursos de compreensão de listas e dicionários tornam o código mais elegante e mais *pythonico*.

Funções Anônimas (Funções *Lambda*)

Uma função anônima em Python é uma função definida sem nome (por isso o termo anônimo). Enquanto as funções tradicionais são definidas utilizando a palavra-chave `def`, as funções anônimas são definidas usando a palavra-chave `lambda`. Portanto, este tipo de função é conhecido popularmente como *funções lambda*. Estas funções possuem a seguinte sintaxe:

```
lambda argumentos: <expressão>
```

Uma particularidade destas funções é que elas podem ter qualquer número de argumentos, mas podem terá apenas uma expressão. Desta forma, a expressão será computada e o seu resultado será retornado. Por isso, nas funções lambdas não é necessário declarar a palavra-chave return.

Vejamos a seguir um exemplo de declaração e utilização das funções lambda.

```
>>>  # Declaração da função
...  area_quadrado = lambda lado: lado ** 2
...
...  # Utilização
...  print(area_quadrado(4))
16
```

Neste exemplo, a declaração `lambda lado: lado ** 2` é a função lambda, onde `lado` é o argumento e `lado ** 2` é a expressão que será avaliada e o seu resultado retornado. Observe que não definimos um nome para esta função, mas atribuímos ela à uma variável chamada `area_quadrado`, que se tornou do tipo `function`. Desta forma, agora podemos utilizá-la como uma função normal. Podemos verificar, utilizando a função `type()`, que a variável `area_quadrado` é do tipo `function`:

```
>>>  # Verificação do tipo da variável
...  print(type(area_quadrado))
```

```
<class 'function'>
```

As funções lambda são úteis quando precisamos de uma função específica por um curto período. Geralmente utilizamos estas funções em conjunto com outras funções que possuem como argumento outra função, como por exemplo a função `map()`, que permite que apliquemos uma função em todos os elementos de uma lista, como no exemplo a seguir.

```
>>>  # Função que calcula o triplo de número
...  triplo = lambda x: x * 3
...
...  # Calcula o triplo dos números de uma lista
...  lista = [4, 5, 9, 7, 0, 1, 8]
...  print(list(map(triplo, lista)))
[12, 15, 27, 21, 0, 3, 24]
```

Ou poderíamos fazer de um jeito mais *pythonico* ainda, em apenas uma linha:

```
>>>  print(list(map(lambda x: x * 3, [4, 5, 9, 7, 0, 1, 8])))
[12, 15, 27, 21, 0, 3, 24]
```

Atribuição Condicional em Uma Linha

Considere um código que cria uma variável, chamada `var`, e atribui a ela um valor de acordo com uma das seguintes condições: Se o nome do programador tiver mais de 5 letras, o valor de `var` deverá ser `100`. Caso contrário (o nome tem 5 ou menos letras), o valor de `var` deverá ser `0`.

Tal código pode ser escrito da seguinte forma:

```
>>>  nome = 'antonio'
...  if len(nome) > 5:
...      var = 100
...  else:
...      var = 0
...  print('O valor de var é:', var)
```

```
100
```

O Python fornece um operador ternário que é utilizado para atribuição de um valor a uma variável, condicionado à uma verificação de uma expressão. Este operador tem a seguinte sintaxe:

```
valor_se_verdadeiro if condicao else valor_se_falso
```

Ou seja, o código anterior pode ser reescrito da seguinte forma:

```
>>>    nome = 'antonio'
...    var = 100 if len(nome_program) > 5 else 0
...    print('O valor de var é:', var)
100
```

Este recurso faz com que seja possível transformar quatro linhas de código em apenas uma. Outro recurso *pythonico*!

Referências

BRITO, M. Python F-String: 73 Examples to Help You Master It. **miguendes's blog**, 2020. Disponível em: <https://miguendes.me/73-examples-to-help-you-master-pythons-f-strings>. Acesso em: 6 março 2022.

CORMEN, T. et al. **Algoritmos:** Teoria e Prática. 2. ed. [S.l.]: LTC, 2012. 944 p. ISBN 8535236996.

DEVMEDIA. Python: Estrutura condicional if-else. **DEVMEDIA**, 2022. Disponível em: <https://www.devmedia.com.br/python-estrutura-condicional-if-else/38217>. Acesso em: 6 março 2022.

FANGOHR, H. **A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering**. International Conference on Computational Science. [S.l.]: Springer. 2004.

GORELICK, M.; OZSVALD, I. **High Performance Python:** Practical Performant Programming for Humans. 2. ed. [S.l.]: O'Reilly Media, 2020. 468 p. ISBN 978-1492055020.

MARTELLI, A.; RAVENSCROFT, A. M.; HOLDEN, S. **Python in a Nutshell**. [S.l.]: O'Reilly Media, Inc., 2017. 772 p. ISBN 144939292X.

MARVIN, R.; NG'ANG'A, M.; OMONDI, A. **Python Fundamentals**. [S.l.]: Packt Publishing Ltd, 2018. 324 p. ISBN 1789807328.

OVERTON, M. L. **Numerical Computing with IEEE Floating Point Arithmetic**. [S.l.]: Society for Industrial and Applied Mathematics, 2001. 97 p. ISBN 978-0-89871-482-1.

PAREWA LABS PVT. LTD. Python Sets. **Programiz**, 2022. Disponível em: <https://www.programiz.com/python-programming/set>. Acesso em: 6 março 2022.

PILGRIM, M. **Dive Into Python 3**. 2. ed. [S.l.]: Apress, 2009. 360 p. ISBN 1430224150.

PSF. Documentação Python 3.9.10. **Documentação Python 3.9.10**, 2022. Disponível em: <https://docs.python.org/pt-br/3.9/index.html>. Acesso em: 6 Março 2022.

PYTHON BRASIL. Python Brasil. **Python Brasil**, 2022. Disponível em: <https://python.org.br/>. Acesso em: 7 março 2022.

REAL PYTHON. Python 3 Installation & Setup Guide. **Real Python**, 2020. Disponível em: <https://realpython.com/installing-python>. Acesso em: 7 março 2022.

ROMANO, F. **Learn Python Programming**. [S.l.]: Packt Publishing, 2018. ISBN 9781788996662.

SALES, L. O que é e como funciona o Garbage Collector em Python. **Medium**, 2020. Disponível em: <https://larissacroco.medium.com/o-que-%C3%A9-e-como-funciona-o-garbage-collector-em-python-87dbf6453d64>. Acesso em: 12 Fevereiro 2022.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. 11. ed. Colorado Springs: Bookman, 2018. 758 p.

SHAW, A. Why is Python so slow? **Hackernoon**, 2018. Disponível em: <https://hackernoon.com/why-is-python-so-slow-e5074b6fe55b>. Acesso em: 12 Fevereiro 2022.

SLATKIN, B. **Python Eficaz**. 1. ed. São Paulo: Novatec, 2016. 296 p. ISBN 978-85-7522-510-3.

SLONNEGER, K.; KURTZ, B. L. **Formal Syntax and Semantics of Programming Languages**. 1. ed. [S.l.]: Pearson, 1995. 592 p. ISBN 0201656973.

TIOBE. The Python Programming Language. **TIOBE Index**, 2022. Disponível em: <https://www.tiobe.com/tiobe-index/python/>. Acesso em: 8 Fevereiro 2022.

VANDERPLAS, J. **A Whirlwind Tour of Python**. 1. ed. [S.l.]: O'Reilly Media, Inc., 2016. ISBN 9781491964644.