IUP - Introdução ao Universo da Programação com Python:

Um livro aberto para aprender programação

Wendel Melo

28 de julho de 2019

Sumário

Prefácio **i**

1 Preliminares **1**
1.1 Computadores, algoritmos e linguagens de programação 1
1.2 Classificação das linguagens de programação 2
1.2.1 Linguagens compiladas 2
1.2.2 Linguagens interpretadas 4
1.2.3 Linguagens híbridas 4
1.3 Mas por que programar em Python? 5

2 Começando com Python **9**
2.1 O interpretador 9
2.2 Variáveis e atribuições 10
2.3 Atribuição e criação de variáveis 11
2.4 Tipos básicos imutáveis 16
2.5 Conversão de tipos 19
2.6 Impressão (saída) de dados: a função *print* 20
2.7 Entrada de dados: a função *input* 21
2.8 Nosso primeiro programa em Python 21
2.9 Operações básicas com números 26
2.9.1 Atribuições ampliadas 28
2.9.2 Exemplo com operações aritméticas 29
2.10 Exercícios 30

3 Expressões Booleanas e Condicionais **33**
3.1 Operadores Lógicos 33
3.1.1 Operador or 33
3.1.2 Operador and 35
3.1.3 Operador not 35
3.1.4 Considerações sobre o uso de operadores lógicos 36
3.2 Operadores de Comparação 36
3.3 Condicionais *if* 39
3.3.1 Primeira forma geral da cláusula *if* 39
3.3.2 Segunda forma geral da cláusula *if* 41
3.3.3 Terceira forma geral da cláusula *if* 44
3.4 Exercícios 45

4 Repetições (laços) 51
4.1 Laço *while* . . . 51
4.1.1 Primeira forma geral . . . 51
4.1.2 Forma mais geral . . . 55
4.2 Laço *for* . . . 55
4.2.1 Função *range* . . . 56
4.3 Cláusulas *break* e *continue* . . . 58
4.4 Exemplos . . . 59
4.4.1 Resultado Acumulativo . . . 59
4.4.2 Resultado acumulativo com teste: maior termo lido . . . 61
4.4.3 Variável sinalizadora: O exemplo de detecção de número primo . . . 64
4.5 Exercícios . . . 65

5 Strings 69
5.1 Sequências de escape (constantes de barra invertida) e literais de string . . . 71
5.2 Operações com strings . . . 72
5.3 Indexação e Fracionamento . . . 75
5.4 Percorrendo uma string . . . 77
5.5 As funções ord e chr . . . 78
5.6 Alguns exemplos . . . 79
5.6.1 Removendo espaços . . . 79
5.6.2 Contando o número de caracteres maiúsculos . . . 82
5.6.3 Contando o número de vogais . . . 82
5.6.4 "Convertendo" caracteres minúsculos em maiúsculos com *ord* e *chr* . . . 83
5.7 Métodos de String . . . 84
5.7.1 Contando consoantes de uma string . . . 88
5.8 Exercícios . . . 88

6 Funções 93
6.1 Definindo suas próprias funções . . . 95
6.2 Escopo de função . . . 97
6.3 Comentários adicionais sobre funções . . . 99
6.4 Mais exemplos . . . 100
6.4.1 Exemplo: função para o cálculo do módulo de um número 100
6.4.2 Exemplo: função para o cálculo de arranjo . . . 101
6.4.3 Exemplos: funções sobre objetos sequenciais . . . 102
6.5 Variáveis locais e globais . . . 103
6.6 Recursão . . . 105
6.7 Funções com número arbitrário de argumentos de entrada . . . 108
6.7.1 Por Tupla . . . 108
6.7.2 Por Dicionario . . . 109
6.8 Exercícios . . . 110

7 Listas e Tuplas **113**
7.1 Operações com listas 115
7.2 Métodos de listas 117
7.3 Percorrendo os elementos de uma lista 120
7.4 Alguns exemplos 121
7.4.1 Ordenação de valores lidos 121
7.4.2 Média aritmética e média geométrica 122
7.5 Usando listas para processamento de texto 124
7.6 Usando listas para manusear matrizes 125
7.6.1 Função para impressão de matriz 126
7.6.2 Geração de uma matriz de zeros 126
7.6.3 Exemplo: multiplicação de matriz por fator 128
7.7 Cópia rasa e cópia profunda 128
7.8 Tuplas 132
7.9 Exercícios 134

8 Importação de Módulos **141**
8.1 Funções matemáticas com o módulo *math* 145
8.2 Geração de números aleatórios com o módulo *random* 145
8.3 Funções de tempo com o módulo *time* 149
8.4 Importando seus próprios módulos 154
8.5 Exercícios 156

Prefácio

Esse texto está sendo construído como material de apoio ao ensino e aprendizado de programação de computadores através da linguagem Python. Embora inicialmente pensado para uma disciplina introdutória de graduação, o mesmo pode vir a ser utilizado por qualquer tipo de estudante, profissional ou curioso em busca de conhecimento. Aqui, tentamos focar nossa atenção no Python 3 apontando algumas diferenças em relação a versão 2. Qualquer pessoa é incentivada a entrar em contato com possíveis críticas, sugestões, exemplos adicionais, exercícios, correção de erros, etc.

Capítulo 1

Preliminares

1.1 Computadores, algoritmos e linguagens de programação

A parte de todos os avanços no campo da inteligência artificial, o computador ainda é, essencialmente, uma máquina que apenas segue ordens a risca, sem qualquer capacidade de reflexão ou questionamento sobre as tarefas que realiza. Originalmente, era necessário especificar instruções para os computadores usando diretamente a linguagem de máquina, que é composta por códigos numéricos binários (ou hexadecimais) de compreensão e manuseio bastante complicado, algo bem rudimentar em comparação às linguagens de programação de hoje. Desse modo, com o objetivo de facilitar a tarefa de prescrever instruções para os computadores, as linguagens de programação estabelecem um conjunto de conceitos e expressões válidas que estão mais próximos às linguagens utilizadas pelos humanos para comunicação natural e escrita de expressões matemáticas. Para que este objetivo seja de fato alcançado, as linguagens de programação possuem ferramentas responsáveis por fazer a tradução entre as expressões permitidas pelas mesmas (mais amigáveis aos humanos) e os códigos numéricos em linguagem de máquina (bem menos amigável aos humanos). Programar através das linguagens de programação é tão mais simples do que através das linguagens de máquina que, uma única instrução escrita em uma linguagem de programação pode ser traduzida para dezenas, centenas, milhares ou talvez até centenas de milhares de instruções (ou mais) em linguagem de máquina.

Aqui, já começa a ficar claro no que consiste a tarefa de programar computadores. Programação de computadores se remete ao desenvolvimento de programas de computador, que, por sua vez, se remete à definição de instruções a seres seguidas por computadores para a realização de tarefas específicas. A esse conjunto de instruções bem definidas para a realização de uma tarefa é dado o nome de *algoritmo*. O conceito de algoritmo é muito anterior às linguagens de programação, e, em muitos contextos, é definido de forma clássica e trivial como sendo fundamentalmente uma "receita de bolo". Ao seguir uma receita de bolo de cenoura (um dos meus prediletos), estamos, na realidade, executando um algoritmo para a preparação de bolo de cenoura. Observe que, desse modo, a ideia de algoritmo está muito difundida no nosso dia a dia, mesmo que muitas

pessoas jamais tenham ouvido essa palavra. Existem algoritmos (receitas) para praticamente todas as tarefas do cotidiano, e não segui-los pode gerar resultados diferentes dos inicialmente desejados, e até nos colocar em apuros. Imagine, por exemplo, se alguém decide fazer uma torta especial de dia dos namorados misturando ingredientes aleatoriamente e levando a mistura ao congelador. Tal atitude poderia gerar resultados catastróficos para o relacionamento em questão! Por essa razão, o conceito de algoritmo envolve a elaboração de instruções bem definidas, isto é sem ambiguidade ou qualquer argem de dúvida, para que, assim, haja sucesso na execução da respectiva tarefa. Observe também que uma instrução estar bem definida é algo um tanto quanto subjetivo e depende do contexto em questão, e da pessoa que estará lendo ou executando o algoritmo. Uma determinada instrução específica que, para mim, é muito clara, pode não ser tão clara assim para uma outra pessoa, ou vice-versa, por uma série de razões.

É oportuno também ressaltar que o conceito de algoritmo é algo puro, no sentido de que não depende de programas, linguagens de programação ou mesmo de computadores. De certo modo, o algoritmo é uma entidade abstrata; É a receita em si para a resolução de um problema ou a execução de uma tarefa, que pode ser definida sem o uso de linguagens de programação, através da própria língua portuguesa ou de qualquer outro tipo de linguagem. Por sua vez, as linguagens de programação, fornecem um modo de implementar a execução de um algoritmo por meio de um computador. Assim, todo programa de computador é a implementação de um algoritmo para a execução de uma tarefa por uma máquina.

1.2 Classificação das linguagens de programação

Existem diversas formas de se classificar as inúmeras linguagens de programação existentes. Aqui, estamos interessados na classificação quanto à execução dos programas gerados, o que nos permite identificar as seguintes três categorias principais:

1. Linguagens compiladas;
2. Linguagens interpretadas;
3. Linguagens híbridas.

1.2.1 Linguagens compiladas

As linguagens compiladas funcionam da seguinte forma: um algoritmo é descrito em um arquivo usando a sintaxe permitida pela linguagem [1]. Esse arquivo é denominado como *arquivo fonte*, pois é partir de seu conteúdo que o programa de computador é gerado. Esse conteúdo de um arquivo fonte, por sua vez, é denominado como *código fonte* . Assim, o código fonte pode ser visto como sendo a representação de um algoritmo usando uma linguagem de programação.

[1] Na prática, é muito comum (e útil) a divisão de um programa em diversos arquivos fontes, no lugar de apenas fim. Para nossa discussão, é indiferente o número de arquivos fonte utilizado para codificar o algoritmo.

O arquivo fonte é então submetido a um programa especial denominado *compilador*, cuja missão é fazer a tradução das expressões escritas na linguagem de programação para a linguagem de máquina, gerando assim um programa executável. Observe que essa tradução só precisa ser realizada uma única vez, não sendo mais necessário o compilador para a execução do programa gerado. Assim, o programa produzido é um executável por si só, que, uma vez gerado, em princípio, não depende de outros programas para a sua execução, além é claro, do sistema operacional da máquina[2]. O processo de tradução pode ser feito de forma a otimizar a execução do algoritmo, o que pode ajudar a gerar programas com boa velocidade de execução. Em geral, o programa executável depende da arquitetura para a qual foi gerado. Um programa executável compilado para Windows, por exemplo, não funcionará nativamente em um sistema Linux. Se um programa for compilado para execução em um processador 64 bits, ele não funcionará em computadores com processador de 32 ou de 16 bits. De modo a gerar programas executáveis para diferentes arquiteturas, pode ser preciso gerar compilações específicas para cada arquitetura.

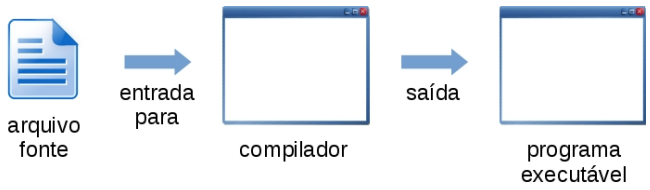

Figura 1.1: Esquema de funcionamento das linguagens compiladas.

O processo de compilação permite aos desenvolvedores venderem ou distribuírem seus programas sem a necessidade de revelar os segredos sobre seu funcionamento contidos no código fonte. Com acesso ao código fonte, torna-se mais simples a compreensão detalhada das operações sendo executadas por um programa, e sua divulgação poderia, em casos extremos, conduzir grandes empresas desenvolvedoras de programas de computador à falência. Graças ao processo de compilação, os códigos fontes podem então ser escondidos do público. A essa altura, o leitor mais atento pode se perguntar: mas não seria possível "descompilar" um programa, isto é, a partir do programa executável, chegar ao código fonte que lhe originou? A resposta seria sim, no entanto esse processo é extremamente complicado sendo praticamente impossível em muitos casos. De toda a forma, não são realizados grandes investimentos para desenvolver pesquisas na área de "descompilação", cujo nome técnico apropriado é *engenharia reversa*, pois a mesma não é considerada como de grande interesse para a sociedade nem para as organizações. Por fim, podemos citar como exemplos de linguagens compiladas dentre outras, as linguagens: C, C++, Fortran e Pascal.

[2]No mundo real, programas complexos podem ser projetados para, propositalmente, dependerem de outros programas. O que queremos dizer aqui é que não é necessário nenhum programa a parte do sistema operacional para realizar tradução para linguagem de máquina.

1.2.2 Linguagens interpretadas

Nesse tipo de linguagem , o algoritmo ainda é descrito em um ou mais arquivos fonte. Esse(s) arquivo(s) fonte são então submetidos a um programa especial denominado *interpretador*, que lê o código fonte contido no(s) arquivo(s) e executa ele próprio as instruções contidas nele. Uma vez que a execução do algoritmo fica a cargo do interpretador, não é gerado um programa executável. Portanto, a princípio, o único modo de distribuir programas feitos em linguagens interpretadas, é através do próprio código fonte, o que implica divulgar detalhes sobre o funcionamento dos programas e comprometer possíveis segredos de negócio envolvidos. Adicionalmente, para conseguir executar o programa, cada usuário precisará possuir um interpretador da respectiva linguagem disponível em sua plataforma.

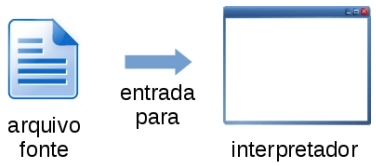

Figura 1.2: Esquema de funcionamento das linguagens interpretadas.

Como, a cada execução, o interpretador precisa ele próprio traduzir as instruções do código fonte para a linguagem de máquina e as executar logo em seguida, em geral, programas interpretados acabam executando mais demoradamente que seus equivalentes compilados. Ademais, em geral, essa tradução costuma não ser feita de forma tão otimizada quanto nas linguagens compiladas, o que também prejudica o desempenho dos programas. Por sua vez, as linguagens interpretadas apresentam vantagens no acompanhamento e na depuração da execução de programas, o que pode facilitar a correção de possíveis erros no código fonte. O fato de haver interpretação dinâmica de código permite que usuários possam fornecer comandos de linguagem de programação diretamente aos programas, o que possibilita um nível maior de interatividade. Por exemplo, um programa feito em Python pode receber do usuário comandos escritos na própria linguagem Python para serem prontamente executados. Além disso, linguagens interpretadas facilitam a portabilidade, pois interpretadores podem, em alguns casos, lidar mais facilmente com possíveis diferenças entre plataformas em comparação com compiladores. Como exemplos de linguagens que podem trabalhar como puramente interpretadas, temos, dentre outras: Python, JavaScript, Matlab, R, PHP e ASP.

1.2.3 Linguagens híbridas

As linguagens híbridas são uma espécie de meio termo entre as linguagens compiladas e as interpretadas, tentando aproveitar um pouco das vantagens de ambos esquemas. Como de praxe, tudo começa com a definição do(s) arquivo(s) fonte que contém um algoritmo codificado na linguagem. Esse(s) arquivo(s) fonte são então submetidos a um programa especial denominado *compilador de código de byte* , cuja finalidade é realizar uma espécie de compilação do código fonte

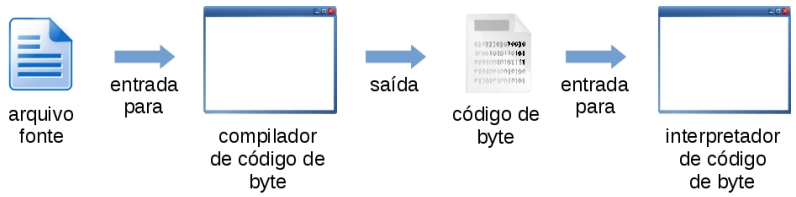

Figura 1.3: Esquema de funcionamento das linguagens híbridas.

como ocorre com as linguagens compiladas. No entanto, aqui, no lugar de gerar um programa executável, o compilador gera *código de byte* (também representado em um arquivo), que, para ser executado, precisa ser submetido a outro programa especial denominado *interpretador de código de byte* , que lerá o conteúdo do código de byte e o executará ele próprio, como um interpretador das linguagens interpretadas. Assim, para distribuir programas, é suficiente fornecer apenas o código de byte, mantendo assim em sigilo o código fonte original. Por haver uma etapa de compilação para código de byte, é possível haver alguma otimização no código gerado, o que pode ajudar no desempenho dos programas, embora essa possível otimização não seja, em geral, tão agressiva quanto nas linguagens compiladas. Observe que ainda é necessário que cada usuário, para executar o código de byte, tenha um interpretador apropriado disponível em sua plataforma de execução, o que também pode manter as vantagens de portabilidade e execução dinâmica do código. Como exemplos de linguagens que podem trabalhar nesse modelo, temos, dentre outros: Python, Java e C#. Observe que algumas linguagens podem trabalhar simultaneamente no modelo interpretado puro, ou no modelo híbrido.

1.3 Mas por que programar em Python?

Aprender a programar computadores pode trazer uma série de benefícios, como desenvolvimento do raciocínio lógico e um melhor uso da tecnologia através da automação de tarefas e sua especialização para casos particulares. O auxílio de computadores tem permitido a profissionais de diversas áreas alcançar resultados, há bem pouco tempo, inimagináveis. Cada vez mais a habilidade de construir soluções personalizadas usando as Tecnologias da Informação e Comunicação (TIC) tem sido um diferencial para profissionais de ponta em áreas relacionadas à (bio) tecnologia, ciências exatas e análise de dados. Em outras palavras, a programação já não se encontra mais restrita apenas a técnicos diretamente ligados à computação, se tornando assim uma poderosa ferramenta cada vez mais adotada por profissionais em contextos diversos. Todavia, o interesse por programação não está limitado apenas ao mundo técnico e acadêmico. Os recentes avanços em termos de dispositivos móveis como tablets e telefones celulares tem ajudado a democratizar o acesso a equipamentos computacionais com capacidade razoável de processamento. Em tempos recentes, além de programadores profissionais, pessoas sem formação tecnológica estrita têm se dedicado ao estudo e desenvolvimento de aplicações gerais para este tipo de dispositivo. Nos tempos atuais, já existem defensores da programação como disciplina escolar de ensino fundamental e médio, tanto para desenvolver a habilidade dos

estudantes em lógica, matemática e resolução de problemas genéricos, quanto para potencializar seu crescimento pessoal e profissional através da construção de projetos tecnológicos. Há ainda os que se interessam por programação como hobby, curiosidade, ou porque sonham em desenvolver jogos ou aplicativos em geral que tragam fama e/ou fortuna.

Uma vez tendo se convencido(a) da necessidade de aprender programação (ou talvez tendo sido forçado(a)), o passo seguinte é a escolha de uma linguagem de programação para iniciar o aprendizado. As linguagens de programação facilitam a elaboração de programas através de um conjunto de conceitos e expressões válidas que nos abstraem de programar computadores usando a (bem complicada) linguagem de máquina. Toda uma gama de opções está disponível para essa escolha, com linguagens de programação voltadas a fins variados. É importante compreender que não existe uma linguagem de programação superior às demais em todos os aspectos. Cada linguagem é projetada com propósitos diversos em mente, possui suas peculiaridades que se traduzem em vantagens e desvantagens dependendo do contexto em questão. Existem linguagens que são mais apropriadas à determinadas situações, e linguagens mais apropriadas a outras. A boa notícia é que, ao aprender os fundamentos da programação em qualquer que seja a linguagem, ao menos em tese, não deveria haver grandes problemas para o aprendizado e adaptação a novas linguagens. De toda a forma, a escolha da primeira linguagem pode ter grande influência na curva de aprendizado dos conceitos de programação. E é aqui que entra o Python.

Python é uma linguagem que permite a programação em altíssimo nível, o que significa que a mesma cuida de detalhes técnicos de funcionamento dos programas e nos libera para focar apenas na implementação de procedimentos (algoritmos) em si. O gerenciamento de memória, por exemplo, é realizado de forma totalmente automática pelo interpretador da linguagem. O foco da linguagem inclui, dentre outras coisas, a facilidade de aprendizado, a rapidez na elaboração de programas, a liberdade e a democratização da programação em si. Desse modo, muitas ferramentas e bibliotecas da linguagem podem ser obtidas para diversas arquiteturas de execução gratuitamente. Essas características têm tornado a comunidade de desenvolvedores Python bastante ativa, crescente e vibrante.

Python tem se tornado a linguagem de programação preferida de muitos profissionais que não possuem formação em TIC, mas têm se deparado com a necessidade de desenvolver programas para atender as suas necessidades de processamento de dados e computação. Muitos usuários de sistemas de computação algébrica bastante avançados (e caros) como Matlab, Mathemática, etc têm encontrado no Python uma alternativa livre e gratuita que fornece bibliotecas com nível semelhante de funcionalidades e facilidade de uso. Muitos profissionais que trabalham com computação científica e numérica estão virando adeptos de bibliotecas Python como Numpy, Scipy, Pandas e Matplotlib. A facilidade e rapidez no desenvolvimento em Python também têm feito a linguagem ser bastante utilizada para prototipação, que consiste em desenvolver uma espécie de pré-projeto de *software* onde funcionalidades iniciais são testadas, tanto para desenvolvimento de sistemas de informação, como para desenvolvimento de aplicações acadêmicos que implementam novas ideias ou conceitos. Além da prototipação, Python também tem sido utilizado para a construção de sistemas complexos, uma vez que pode ser utilizado para programação voltada a computadores de mesa (desktops ou laptops), internet, e até mesmo para dispositivos

móveis.

O fato de muitas arquiteturas já contarem com a disponibilidade de interpretadores Python o torna uma linguagem bastante portável. A maioria das distribuições Linux já vem com um interpretador instalado "de fábrica". É possível também instalar interpretadores Python em sistemas Unix em geral, Windows, OS, iOS e Android, o que permite executar programas Python nesses sistemas.

Python também pode ser integrado a outras linguagens de programação, como C, Java, Lua ou R. A integração com a linguagem C, em especial, permite contornar uma das maiores deficiências de Python que está na relativa lentidão com que os programas são executados. É possível, por exemplo, desenvolver partes críticas de um *software* em C para integração com Python, ou mesmo importar bibliotecas Python que já possuem núcleo implementado em C, e, assim, se valer da velocidade de execução de programas em C sem ter de abandonar o ambiente Python.

Python também oferece suporte a diferentes paradigmas de programação. É possível, por exemplo, desenvolver programas sobre o paradigma da programação estruturada, da orientação a objetos, da programação funcional, ou mesmo uma combinação de todos estes paradigmas. Recursos avançados como tratamento de exceções, herança múltipla e suporte a expressões regulares também estão disponíveis. Por fim, gostaria de apontar que, aprendendo Python, você terá em mãos uma poderosa ferramenta que poderá lhe trazer vantagens diversas, seja você um profissional de TIC, um profissional que faz uso da tecnologia, um acadêmico, um técnico de uma área qualquer ou mesmo um curioso.

Capítulo 2

Começando com Python

2.1 O interpretador

Se o seu sistema operacional não possuir uma versão mais ou menos recente do Python, você precisará instalar alguma para praticar os conceitos e exemplos discutidos aqui. Você pode procurar pelo Python nos repositórios do sistema (Unix) ou indo até o site oficial do Python `www.python.org`. Lá também é possível encontrar bastante material sobre a linguagem, incluindo tutoriais, exemplos e documentação. Conforme vimos na Seção 1.2, Python pode trabalhar como linguagem interpretada ou híbrida. Assim, o prompt da linguagem oferece um interpretador pronto a receber comandos e executá-los, como pode ser observado na Figura 2.1.

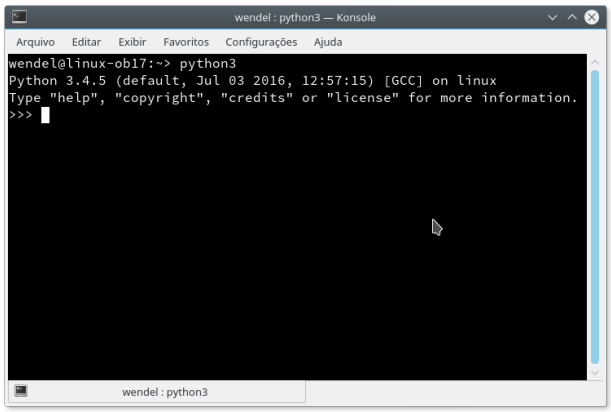

Figura 2.1: Prompt interativo do Python.

É possível abrir o interpretador por meio do menu de aplicativos do seu sistema, ou através do comando python (ou python3). Há também a possibilidade de executar comandos Python na IDLE, que é uma IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) bem básica desenvolvida na própria linguagem Python. A IDLE também traz um simulador para o prompt, conforme pode ser conferido na figura 2.2: Além de ser possível executar comandos Python um por vez diretamente no prompt,

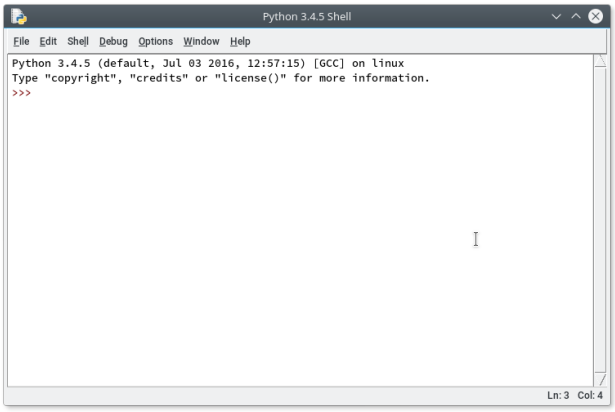

Figura 2.2: IDLE do Python.

obtendo-se assim seu resultado de imediato, também é possível a construção de arquivos fonte com uma série de comandos para serem executados de uma só vez em sequência. Mais adiante, veremos em detalhes como realizar essa tarefa. Por hora, é oportuno ressaltar que o prompt indica estar pronto para receber um comando escrevendo na tela a sequência de 3 caracteres ">>>". Assim, adotamos nesse texto a convenção de que comandos Python precedidos pelos caracteres ">>>" estão sendo executados no prompt de comando, conforme a seguir:

```
1 >>> x = 2 + 3
```

. Ao apertar enter após digitar o comando acima (sem os caracteres ">>>" que são impressos pelo próprio interpretador), o interpretador imediatamente executará a linha de código, que, no caso, atribui à variável x o resultado da operação 2 + 3, que é 5. Discutiremos o conceito e funcionamento das variáveis em Python em detalhes na Seção 2.2. Por hora, nos limitaremos a dizer que podemos ecoar (visualizar) o conteúdo da variável criada no interpretador apenas digitando seu nome:

```
1 >>> x = 2 + 3
2 >>> x
3 5
4 >>>
```

Note que, após executar um comando, o interpretador imediatamente se prontifica a receber o próximo comando para execução.

2.2 Variáveis e atribuições

De modo geral, programas de computador manipulam dados. Enquanto algum dado estiver sendo manipulado, ele precisa estar carregado na memória de trabalho da máquina. Podemos pensar nessa memória de trabalho (RAM) como uma grande tabela com células endereçadas para o armazenamento de dados. Para buscar ou atualizar um dado nessa tabela, é necessário saber o endereço

exato da(s) célula(s) que o contém. Nesse contexto, as linguagens de programação nos oferecem as variáveis como uma abstração que nos poupa a enorme complexidade de manipular diretamente os endereços das células de memória.

A linguagem Python trata as variáveis de uma forma bastante moderna e peculiar. Todo o gerenciamento de memória é feito de forma automática pelo interpretador da linguagem, liberando o programador para se preocupar com questões mais intrínsecas de seus algoritmos. Algumas características marcantes das variáveis em Python são descritas a seguir:

- Variáveis em Python são encaradas como referências, isto é, apontadores, para objetos na memória. Isso significa que, na prática, toda variável é um ponteiro, isto é, aponta para algum objeto na memória. Antes que programadores C comecem a roer suas unhas, é válido destacar que é muito mais simples manipular variáveis em Python em comparação com os ponteiros em C;

- Variáveis são criadas e destruídas dinamicamente ao longo da execução dos programas em Python. Também não há a declaração "formal" de variáveis no início de cada função como ocorre em linguagens como C. A criação de uma variável ocorre quando se realiza a primeira atribuição de dado sobre a mesma;

- Ao contrário de linguagens tradicionais como C e Fortran, em Python as variáveis não possuem tipo. São os objetos para os quais as variáveis apontam (referenciam) é que possuem tipo. Desse modo, qualquer variável em Python pode apontar para qualquer tipo de objeto. Inclusive é possível fazer com que uma variável que aponte para um objeto do tipo X passe a apontar para outro objeto de um tipo diferente Y.

2.3 Atribuição e criação de variáveis

Em Python, as variáveis podem ser criadas através da atribuição de objetos (dados). A operação de atribuição é feita com o operador =, conforme o exemplo abaixo, rodado no prompt do Python (note os caracteres ">>>" do próprio prompt antes do nosso comando):

```
1 >>> a = 1
```

A operação acima cria uma variável chamada `a` e lhe atribui o valor 1. É importante ressaltar desde já que, no contexto da programação em Python, **o operador = não tem o significado matemático ao qual estamos acostumados**. Aqui o operador = significa atribuição, e não a igualdade propriamente dita. Mais formalmente, interpretamos o operador = como sendo: "pegue o resultado da expressão a direita e atribua este resultado à variável que está a esquerda". Avaliando essa operação de forma mais minuciosa, podemos dividi-la em três passos:

- **1°Passo:** Gera-se na memória o objeto (dado) resultante do lado direito da atribuição (o número 1);

- **2°Passo:** Se a variável indicada do lado esquerdo da atribuição não existir no escopo (contexto) atual, cria-se esta variável (a variável `a`);

- **3°Passo:** A variável indicada no lado esquerdo da atribuição (a variável `a`) passa a apontar para o objeto gerado no 1°Passo (o número 1).

Após a execução da operação, podemos gerar um esquema (simplificado) do estado corrente de variáveis e objetos, representado na Figura 2.3. No prompt

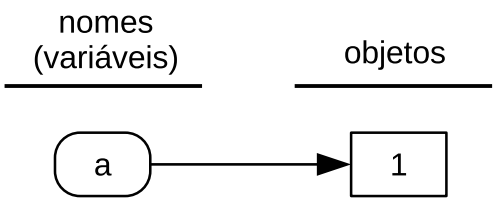

Figura 2.3: Estado corrente das variáveis e objetos na memória.

do Python, é possível ecoar (exibir) o conteúdo de uma variável, isto é, objeto para o qual ela aponta, a partir de seu nome:

```
1 >>> a
2 1
```

Observe que a variável `a` passa a apontar para o número 1. Por questões de praticidade, dizemos que a variável `a` está com o valor 1, ou que a variável `a` está armazenando o valor 1, ou ainda, que o conteúdo da variável `a` é 1.

É importante ressaltar que a **atribuição sempre ocorre da direita para a esquerda**, o que implica que a variável que receberá o resultado **sempre** deve estar na esquerda. Assim, a expressão a seguir estaria equivocada:

```
1 >>> 1 = a          #expressão equivocada
```

Pois ela seria lida como "pegue o dado da variável `a` e atribua este dado ao número 1, o que não faz sentido. Por outro lado, a expressão a seguir é válida:

```
1 >>> a = 1          #expressão válida (pega o número 1 e atribui à
      avariável a)
```

Avaliamos agora o resultado da expressão a seguir:

```
1 >>> b = a
```

A interpretação do comando acima seria "pegue o objeto resultante à direita e atribua à variável à esquerda". Assim, o resultado é que `b` passa a apontar para o mesmo objeto que `a` aponta, como no esquema da Figura 2.4.

Mais uma vez ressaltamos que o sinal = não possui o significado de igualdade ao qual estamos habituados. Aqui, = tem o significado de atribuição. A operação `b = a` apenas faz com que `b` aponte para o mesmo objeto para que `a` aponta. Essa operação não gera nenhum tipo de relação entre `b` e `a`, que são variáveis totalmente independentes sem qualquer tipo de amarração. Por exemplo, suponhamos que o seguinte comando seja passado ao interpretador

```
1 >>> a = 2
```

A interpretação seria "pegue o objeto do lado direito (2) e atribua à variável do lado direito (`a`)". Assim, a variável `a` passará a apontar para o objeto 2 daqui por diante, como ilustrado na Figura 2.5.

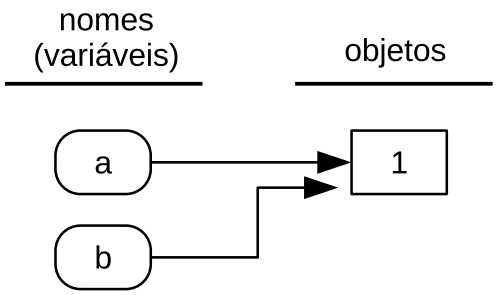

Figura 2.4: Estado corrente das variáveis e objetos na memória após atribuição $b = a$.

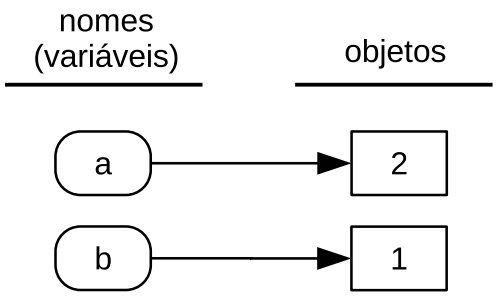

Figura 2.5: Estado corrente das variáveis e objetos na memória após a atribuição $a = 2$.

Sempre que for preciso utilizar a variável `a` para alguma operação, será resgatado o objeto para o qual ela aponta no momento da realização da operação. Nesse instante, seria o valor 2. Apontamos que apenas o valor atual da variável é armazenado pelo interpretador Python. Isso significa que, nesse momento, a variável `a` apenas "sabe" que aponta para o objeto 2. Ela sequer "sabe" que já apontou para o objeto 1 em algum instante do passado.

Note ainda que a nova atribuição sobre `a` não alterou o valor de `b`, que continua com o valor 1. Embora possa causar alguma confusão, é preciso mencionar que quando duas variáveis apontam para o mesmo objeto e esse objeto sofre alguma alteração, essa alteração se reflete no valor apontado por ambas as variáveis, afinal ambas apontam para o mesmo objeto. No entanto, objetos numéricos no Python são do tipo *imutável*, o que significa que esses objetos jamais podem ser alterados após a criação, apenas apagados da memória (discutiremos mais sobre objetos imutáveis na Seção 2.4). Assim, a operação `a = 2`, não altera o objeto 1. O que ocorre, na prática, é que um novo objeto com valor 2 é criado para que a variável `a` aponte para o mesmo. Assim, ao se trabalhar com objetos imutáveis, como por exemplo os números em geral, não corremos o risco de alterar o conteúdo de uma variável como efeito colateral de uma operação sobre outra variável. Todavia, mais adiante ao manipularmos objetos do tipo mutável (como listas, conjuntos e dicionários), será preciso tomar um cuidado especial sempre que o mesmo objeto for apontado por duas variáveis distintas. Por fim, observe que o conceito de mutabilidade ou imutabilidade se aplica aos objetos apontados pelas variáveis, a depender do seu tipo, e não às variáveis em si. Assim, sempre é possível fazer uma variável apontar para um novo objeto a qualquer momento.

Também é possível utilizar expressões aritméticas no lado direito da atribuição. Assim, o comando a seguir:

```
1 >>> c = b + 6
```

cria uma variável chamada `c` e a faz apontar para o resultado da expressão `b + 6`, que é 7, pois o valor atual de `b` é 1, conforme a Figura 2.6. Note que

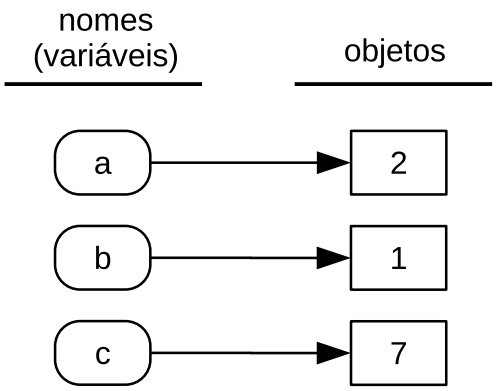

Figura 2.6: Estado corrente das variáveis e objetos na memória após a atribuição $c = b + 6$.

a variável `c` apenas aponta para o valor 7, que é o resultado da operação. No entanto, ela não saberá que esse 7 foi obtido a partir do valor em `b`, o que

significa que `b` e `c` são variáveis totalmente independentes, isto é, sem qualquer tipo de amarração. Assim, a mudança do objeto apontado por `b` não interfere em `c`, como no exemplo a seguir:

```
>>> b = 13
>>> b
13
>>> c       #apesar da ''mudança'' do valor de b, c continua
    valendo 7
7
```

O comportamento visto no exemplo anterior reforça que o operador = não possui o significado matemático ao qual estamos habituados, mas sim o de atribuição à variável à esquerda o resultado da expressão à direita. O correto entendimento do operador = é fundamental para o aprendizado da programação com Python. Um fato curioso sobre o operador = é que ele permite a escrita de expressões como a seguinte:

```
>>> c = c + 1
```

Do ponto de vista matemático, a expressão `c = c + 1` não faz sentido, pois se remeteria a um número `c` que seria igual a ele mesmo mais 1. No entanto, no contexto da programação em Python essa expressão é totalmente válida e significa: "pegue o objeto resultante do lado direito (8) e atribua à variável do lado esquerdo (c)". Assim, a variável c passará a apontar para o valor 8:

```
>>> c
8
```

Observe que o fato da mesma variável ter sido usada na geração do objeto resultante e no seu armazenamento não traz nenhum tipo de empecilho. Aproveitamos para observar que o interpretador realiza todo o gerenciamento de memória de forma automática em Python. Assim, sempre que um determinado objeto na memória não mais possuir qualquer apontamento para si, o mesmo será automaticamente deletado pelo coletor de lixo do interpretador sem que o usuário nem mesmo o veja em operação. Assim, os valores 1 (anteriormente apontado por `a` e `b`) e 7 (anteriormente apontado por `c`) são removidos da memória ao ficarem sem apontamentos. Também é possível remover uma variável do escopo atual por meio do operador `del`:

```
>>> del c       #a variável c deixará de existir
```

É oportuno destacar que o operador del remove variáveis, e não objetos. Os objetos são removidos automaticamente pelo coletor de lixo quando não existem mais apontamentos para os mesmos. Em nosso exemplo, em particular, ao remover a variável `c`, o objeto 8 ficará sem receber apontamento, e, por consequência, será removido também pelo coletor de lixo. Todavia, se houvesse outra variável apontando para o mesmo, ele ainda permaneceria na memória.

Finalmente, é válido ressaltar que o nome de uma variável não precisa ser composto de uma única letra. Na realidade, é possível usar nomes com um número arbitrário de caracteres alfanuméricos e '_' (*underline*), desde que o primeiro caracter não seja um número. **É muito importante frisar que nomes de variáveis não podem conter espaços em branco, acentos ou pontuação em geral!** A Tabela 2.1 traz alguns exemplos válidos e inválidos para nomes de variáveis. Também é relevante destacar que Python faz diferen-

Nomes válidos	Nomes inválidos
w5	5w
jessica	jéssica
ana_luiza	ana luiza
_casa	!pedra
AguiA	agui@$

Tabela 2.1: exemplos de nomes válidos e inválidos para variáveis.

ciação quanto a caixa, o que significa que o nome `karen` é tratado como sendo diferente de `KAREN` ou `KaREn` ou `kArEn`. Assim, todos esses nomes se remeteriam a variáveis distintas para o interpretador Python.

2.4 Tipos básicos imutáveis

Conforme mencionado na Seção 2.2, objetos imutáveis são objetos que não podem ser alterados na memória após a sua criação, apenas destruídos. Assim, para "modificar" um valor imutável armazenado em uma variável, é necessário gerar um novo objeto com o valor desejado e usar uma operação de atribuição para que o mesmo seja apontado pela variável. Em contrapartida, objetos do tipo mutáveis, como listas, conjuntos e dicionários, permitem sua alteração na memória, o que exige mais cuidado em sua manipulação. Nessa seção, apresentamos os tipos básicos imutáveis em Python, postergando então a discussão sobre os tipos mutáveis. Todos os tipos de dados numéricos são imutáveis em Python. Estes tipos numéricos são compostos pelas classes:

- `bool`: tipo booleano. Este tipo de dado só pode assumir dois valores `True` (verdadeiro) e `False` (falso). Em termos estritamente teóricos, valores booleanos não são numéricos. Entretanto, Python os trata associando o número 1 ao valor `True` e 0 ao valor `False`, o que permite que sejam usados até mesmo em expressões aritméticas. Exemplos de uso de valores booleano:

```
1 >>> alto = True   # atribui à variável alto o valor True
2 >>> feio = False  # atribui à variável feio o valor False
```

- `int`: tipo para números inteiros. A partir da versão 3, objetos `int` possuem precisão arbitrária, o que significa que podem representar números com quantidade arbitrária de dígitos. Antes da versão 3, o tipo `int` representava números inteiros com quantidade fixa (tipicamente 32) de dígitos binários (bits). No entanto, essas versões anteriores a 3 definem o tipo `intl` para representar inteiros de precisão arbitrária. Exemplos de uso de objetos do tipo `int`:

```
1 >>> ano = 2018 # atribui à variável ano o valor int 2018
2 >>> num = 7    # atribui a variável num o valor int 7
3 >>> v = -1     # atribui a variável v o valor int -1
```

  A capacidade de representar números inteiros com precisão arbitrária faz

com que seja simples manipular grandes números em Python. Por exemplo, podemos calcular 3^{1000}, cujo resultado é um valor bem grande:

```
>>> r = 3**1000   # atribui à variável r o resultado de 3 elevado a 1000
>>> r
1322070819480806636890455259752144365965422032752148167
6649203682268285973467048995407783138506080619639097776
9687258235595095458210061891186534272525795367402762022
5198320803878014774228964841274390400117588618041128947
8156230944380615661730540866744905061781254803444055470
5439703889581746536825491613622083026856377858229022841
6398307887896918556404084898937609373242171846359938695
5167650189405881090604260896714388641028143503856487471
6583201061436613217310276890285522000 1
```

Esse tipo de cálculo não seria tão trivial em linguagens como C.

- `float`: tipo para números reais racionais. Esta classe representa números reais racionais que podem ser representados na codificação ponto flutuante com 64 dígitos binários de precisão (precisão dupla). Exemplos de uso da dados `float`:

```
>>> altura = 1.89
>>> placar = -3.14
>>> peso = 80.0
>>> pot = 2e7  # atribui à variável pot o resultado de 2.0 vezes 10 elevado a 7
```

O que diferencia a declaração de um dado `int` e um dado `float` é a presença do caracter '.' (ponto) para separar a parte inteira da parte decimal. Dessa forma, os objetos `7` e `7.0` são de tipos diferentes pois o primeiro é um `int`, ao passo que o segundo é um `float`. Isso faz que esses objetos sejam representados de forma totalmente diferente na memória, embora se refiram a mesma entidade do mundo real. Na prática, se for detectado que uma entidade do mundo real só pode possuir valores inteiros, como por exemplo a idade de uma pessoa, é preferível o uso do tipo `int` em vez do tipo `float`.

Um erro muito comum, especialmente entre brasileiros, é realizar a separação entre a parte inteira e a parte decimal usando vírgula no lugar de ponto. Sintaticamente, a vírgula é utilizada, dentre outras coisas, para separar elementos de objetos sequenciais. Assim, ao escrever

```
>>> topo =  53,82
```

estamos, na realidade, declarando uma tupla de dois elementos.

```
>>> topo
(53, 82)
```

- `complex`: tipo para números complexos. Utiliza-se a letra `j` (maiúscula ou minúscula) para designar o número imaginário. Exemplos de uso de dados `complex`:

```
>>> h = 3 + 5j
>>> ka = 20J
>>> men = -2.1 + 1.5j
```

.

Além dos tipos numéricos, também são classes de tipos imutáveis em Python:

- `None` (`NoneType`): tipo especial de objeto, que serve como lugar reservado vazio. A primeira vista, pode parecer estranho um objeto que representa o vazio, mas a função do `None` é ser um objeto que apenas ocupa espaço. Podemos usar o `None`, por exemplo, para representar um dado que no momento não é conhecido, mas que futuramente o será, por exemplo, inicializando uma variável ou posições de objetos sequenciais como listas (*arrays*). Exemplo:

```
>>> cpf = None
```

  Há quem enxergue uma relação entre o uso do objeto `None` e o da constante NULL da linguagem C.

- `str`: sequência de caracteres (string) . Este tipo de dado é utilizado para representar texto. Em Python, não há o tipo caracter isolado como ocorre em linguagens como C. Todavia, é possível trabalhar com strings de um caracter só. Discutiremos strings em mais detalhes adiante no Capítulo 5. Por hora, nos limitaremos a dizer que strings são declaradas usando aspas simples ou duplas de modo equivalente. Exemplo:

```
>>> nome = "jessica"
>>> universidade = 'ufrj'
```

- `tuple`: tupla. Este tipo de dado funciona como uma lista (array) imutável de objetos de tipos quaisquer. São declaradas com parênteses (opcionais) e seus elementos são separados usando vírgula, conforme a seguir:

```
>>> megasena = (5, 16, 18, 39, 44, 57)
>>> notas =  6.3 , 2.9 , 8.1
>>> dados = ( 5, False, None, 2.71, "Rachel", "Diana" )
```

- `type`: objetos para manipulação de tipos. Objetos type servem, por exemplo, para verificarem o tipo de objetos em geral.

```
>>> a = 3
>>> b = type(a)   # b armazena o tipo de a nesse
    instante
>>> b
<class 'int'>
>>> b == int          #pergunta se b armazena type int
True
>>> type(a) == float #pergunta se o tipo de a é float
False
```

2.5 Conversão de tipos

O nome de cada classe (tipo) funciona como um construtor de objetos da respectiva classe. Para quem ainda não entende muito de orientação a objetos, é suficiente saber que isso serve para gerar objetos de uma classe a partir do valor de objetos de outra classe (conversão de tipos). Por exemplo, vamos supor que temos uma variável peso com o valor 60:

```
>>> peso = 60
```

para gerar objetos de outros tipo (conversão), por exemplo `float`, `complex` e `str` podemos fazer:

```
>>> outropeso = float(peso)   #gera um objeto float a partir do
    valor da variável peso
>>> outropeso
60.0
```

```
>>> maisumpeso = complex(peso) #gera um objeto complex a partir
    do valor da variável peso
>>> maisumpeso
(60+0j)
```

```
>>> ultimopeso = str(peso) #gera um objeto str (string) a partir
     do valor da variável peso
>>> ultimopeso
'60'
```

É importante destacar que o valor contido na variável peso não foi alterado pelos exemplos anteriores. A "conversão" de objetos `float` para `int` faz com que a parte fracionária do número seja desprezada.

```
>>> pi = 3.1415
>>> num = int(pi)
>>> num
3
```

Existem conversões que não podem ser realizadas, em geral, por não fazerem sentido. Por exemplo, não é possível converter um número em um objeto `tuple`:

```
>>> tuple(peso)
Traceback (most recent call last):
File "<stdin>" line 1, in <module>
TypeError: 'int' object is not iterable
```

Note que a tentativa de conversão de um objeto `int` (aquele apontado pela variável peso) em `tuple` gerou um erro de execução (exceção), já que apenas objetos sequenciais (iteráveis), como, por exemplo, os da classe `str` podem ser convertidos para `tuple`, conforme a seguir:

```
>>> nome = "leidiana"
>>> t = tuple(nome)
>>> t
('l', 'e', 'i', 'd', 'i', 'a', 'n', 'a')
```

2.6 Impressão (saída) de dados: a função *print*

A função `print` permite a impressão de informações em Python. Aqui, usamos o verbo imprimir com o significado de "escrever algo na tela". Podemos utilizá-la, por exemplo, para enviar uma saudação ao mundo imprimindo uma string (`str`):

```
>>> print("Ola mundo!")
Ola mundo!
```

Ela também pode ser utilizada, por exemplo, para verificar o conteúdo de uma variável:

```
>>> nome = "jessica"
>>> print(nome)   #aqui, como não usamos aspas, assume-se que é desejado o conteúdo da variável nome, e não a palavra "nome" em si
jessica
```

Note que, a segunda linha do exemplo anterior ordena a impressão do conteúdo da variável nome. Se a palavra nome estivesse entre aspas, seria compreendido que a impressão deveria ser da própria palavra "nome" (literalmente), e não do conteúdo da variável nome, como no exemplo a seguir:

```
>>> print("nome") #aqui como nome está entre aspas, entende-se que a palavra nome deve ser impressa, e não o conteúdo da variável nome
nome
```

Ok, realizar impressão no próprio prompt do Python pode parecer algo sem muita graça. No entanto, ao construirmos nossos programas de verdade fora do prompt, será inicialmente necessário utilizar a função `print` para que possam forneçam informações para o usuário, isto é, a função `print` se constitui numa das formas mais básicas de comunicação com este. Antes de prosseguirmos com um exemplo mais complexo, deve ser ressaltado que nas versões do Python anteriores a 3, `print` é uma instrução, e não uma função. Por conta disso, o uso adequado de `print` nessas versões requer a não colocação dos parênteses aqui colocados. Por exemplo, o mesmo exemplo anterior no Python 2.x seria:

```
#código para o Python 2.7
>>> print "nome"  #aqui como nome está entre aspas, entende-se que a palavra nome deve ser impressa, e não o conteúdo da variável nome
nome
```

Podemos realizar impressão de diversos objetos de uma só vez com `print`, separando esses objetos com vírgula:

```
>>> nome = "jessica"
>>> idade = 27
>>> print(nome, idade)  #imprime o conteúdo de nome seguido do conteudo de idade
jessica 27
```

a última linha do exemplo anterior imprimirá o conteúdo da variável nome seguido do conteúdo da variável idade [1]. Podemos construir uma impressão mais amigável ao usuário colocando mais objetos `str` para serem impressos:

[1] Nas versões do Python anteriores a 3, essa linha deve ser digitada sem o par de parênteses, pois do contrário, o interpretador Python entenderá que se deseja imprimir uma tupla.

```
>>> print(nome, "tem", idade, "anos")
jessica tem 27 anos
```

Observe que o exemplo anterior imprime quatro objetos em sequência, o que forma a mensagem amigável para o usuário "jessica tem 27 anos".

2.7 Entrada de dados: a função *input*

Programas de computador manipulam dados, que precisam ser obtidos de alguma fonte. A função `input` tem a finalidade de solicitar entrada de dados ao usuário. A partir da versão 3, esta função imprime uma string (`str`) na tela, e aguarda o usuário digitar alguma coisa e pressionar a tecla Enter. O conteúdo digitado pelo usuário é então devolvido pela função como uma nova string, que pode então ser atribuída à uma variável. Veja o exemplo:

```
>>> cidade = input("Digite o nome da sua cidade: ")
Digite o nome da sua cidade:
```

Primeiramente, definimos uma variável chamada cidade para receber a entrada digitada pelo usuário. Observe que, nessa mesma linha, passamos à função `input` a string "Digite a sua cidade: ". Essa string será impressa na tela para que o usuário saiba qual informação deve fornecer. Na segunda linha do exemplo, vemos a execução da primeira linha, onde o interpretador imprime a string passada e fica no aguardo do usuário digitar alguma coisa e pressionar Enter. Vamos supor que o usuário digite "Rio de Janeiro" e pressione Enter, conforme abaixo:

```
>>> cidade = input("Digite o nome da sua cidade: ")
Digite o nome da sua cidade: Rio de Janeiro
>>>
```

Se solicitarmos o conteúdo da variável `cidade`, veremos que ela aponta para um objeto string que possui exatamente o conteúdo digitado pelo usuário, a exceção do Enter:

```
>>> cidade = input("Digite o nome da sua cidade: ")
Digite o nome da sua cidade: Rio de Janeiro
>>> cidade
"Rio de Janeiro"
```

Por fim, é válido destacar que a função `input` do Python 3 é equivalente à função `raw_input` do Python 2. Assim, para rodar esse mesmo exemplo em versões anteriores a 3, use `raw_input` no lugar de `input`.

2.8 Nosso primeiro programa em Python

Agora que já sabemos como nos comunicar com usuários através das funções `input` (entrada de dados) e `print` (saída de dados), estamos prontos para construir nosso primeiro programa em Python. Para tal atividade, usaremos a IDLE do Python. Podemos abrir a IDLE pelo menu de aplicativos do sistema operacional, ou através do comando `idle` (ou `idle3`) na linha de comando (se você não conseguir abrir a IDLE do Python, certifique-se de que a mesma está instalada, e na versão correta). A tela da IDLE, por padrão, traz um simulador para o

prompt do Python, mas, em geral, ela é branca, conforme exibido na Figura 2.2.

Como agora desejamos construir um programa de fato, e não apenas entrar com comandos no prompt, precisaremos construir nosso arquivo fonte, que, nada mais é do que um arquivo de texto puro com o código fonte do programa. Para isso, clicamos no menu `File`, e, em seguida, em `New File`, como mostra a Figura 2.7

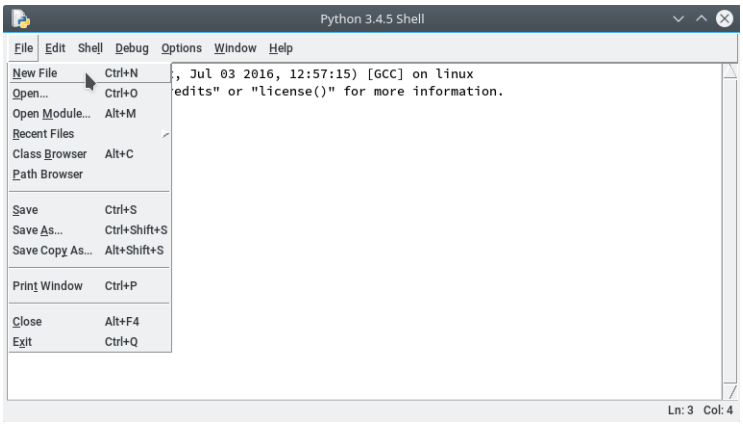

Figura 2.7: IDLE do Python.

Em seguida, se abrirá uma janela com um editor de texto bem simples, para que possamos elaborar nosso código fonte, de forma similar à Figura 2.8

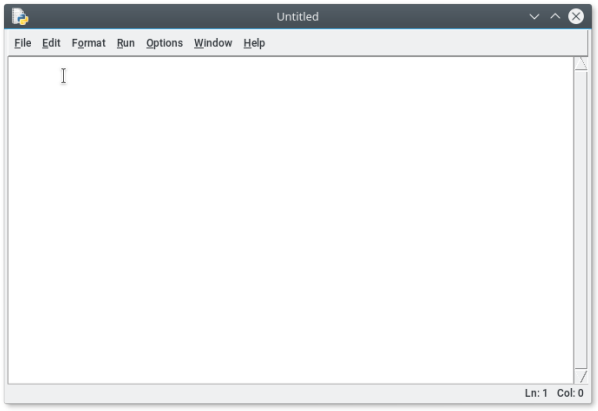

Figura 2.8: editor da IDLE do Python.

Digitaremos, nessa janela do editor da ILDE, o seguinte código

```
1 #primeiro programa em Python
2 #autor: Wendel Melo
3
4 nome = input("Digite o seu nome: ")
5 print("Ola ", nome, "!")
```

Código 2.1: primeiro programa em Python.

Inicialmente, chamamos a atenção para as duas primeiras linhas. Observe que elas contém informações voltadas à leitura por seres humanos, e não para execução do programa propriamente dita. Observe que, não por acaso, ambas as linhas são inciadas pelo caracter `#`. Este caracter tem a finalidade de definir o que chamamos de *comentários*, que são informações que devem ser ignoradas pelo interpretador Python e estão lá apenas para ajudar leitores humanos a compreenderem do que se trata o código fonte. Assim, tudo o que estiver em uma linha do código após o caracter `#` não será considerado pelo interpretador.

Salvamos o arquivo, por exemplo com o nome "primeiro.py", através da opção `Save` no menu `File`. Você está livre para escolher qualquer nome que julgar conveniente, mas é altamente recomendável escolher um nome que termine com a extensão ".py". O uso de caracteres acentuados ou espaço em branco é desencorajado aqui. O próximo passo, é executar o programa, através do menu `Run`, conforme a Figura 2.9 A partir daí, se inicia a execução do programa na janela

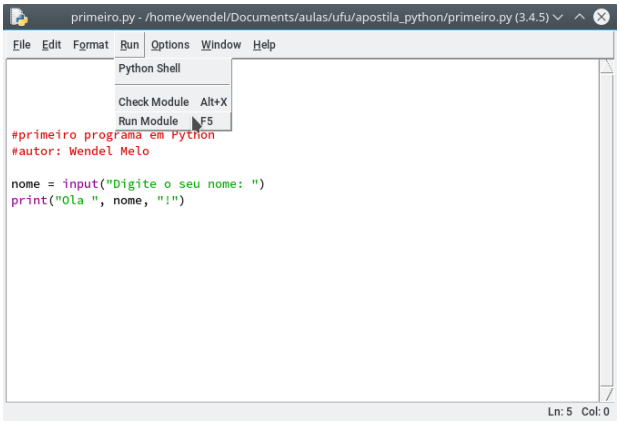

Figura 2.9: editor da IDLE do Python.

principal da IDLE (Figura 2.10). Note que o programa inicia executando a linha 4 (a primeira linha após os comentários), imprimindo na tela a mensagem "Digite o seu nome" e aguardando o usuário fornecer alguma informação e pressionar ENTER. Ao entrarmos com a informação, o programa segue a execução, executando então a linha 5, que estabelece a impressão de uma mensagem de saudação (Figura 2.11). Como não há mais linhas de código a serem executadas, o programa encerra sua execução, e o prompt da IDLE se torna imediatamente apto a receber comandos.

É importante frisar que as linhas do programa são executadas segundo a ordem em que forem escritas. Por essa razão, não faria sentido escrever a linha 5 antes da linha 4, pois a execução da linha 5 necessita do valor da variável `nome`, que só é definido na linha 4. Assim, a execução da linha 5 depende da execução anterior da linha 4. A troca de posição dentre essas duas linhas geraria um erro de execução quando o interpretador fosse tentar imprimir o conteúdo da variável `nome` (que não estará definida). Durante a construção de seus programas, é preciso ter em mente esse tipo de dependência entre as instruções para definir uma ordem adequada para as linhas de código.

Podemos ainda tornar o Código 2.1 mais elaborado, através de mais usos de `input` e `print`. Por exemplo, podemos perguntar também a idade do usuário.

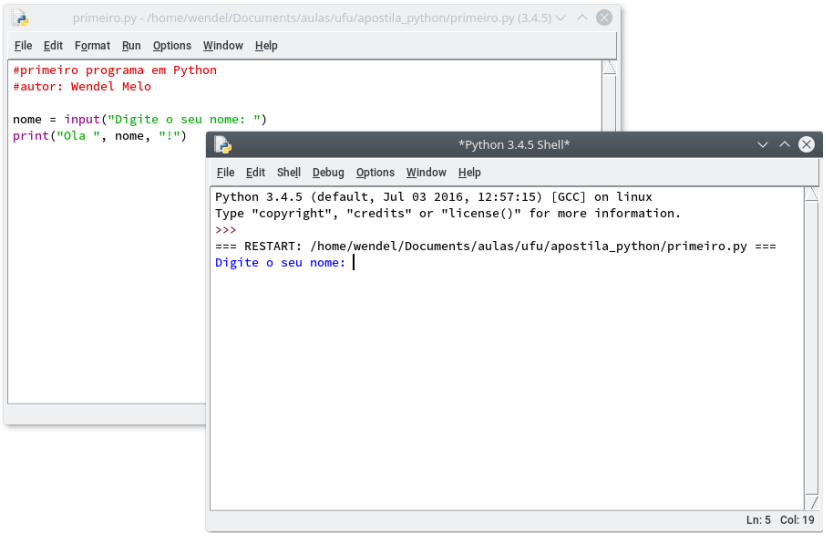

Figura 2.10: Editor da IDLE do Python durante a execução do programa.

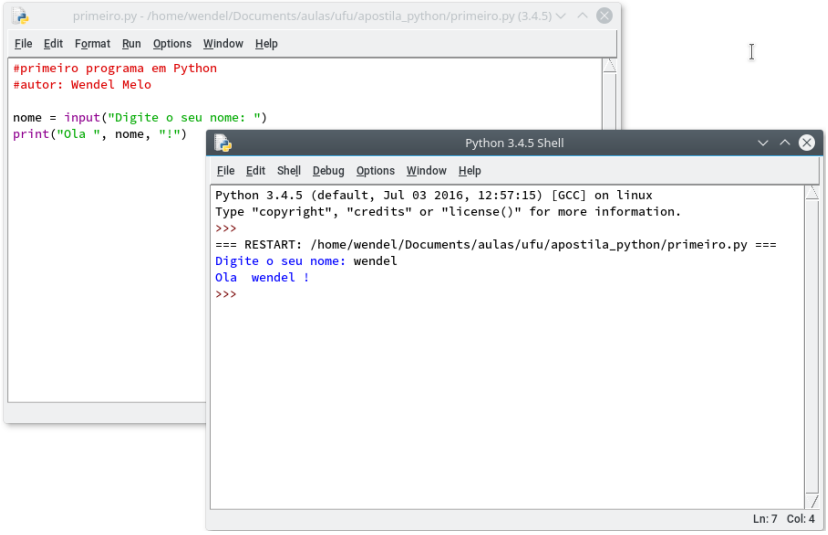

Figura 2.11: editor da IDLE do Python após a execução do programa.

Note que, nesse caso, como idade é um dado numérico (número inteiro), é recomendável realizar a conversão do objeto `str` retornado por `input` para o tipo `int`, como realizado abaixo:

```
1 idade = input("Digite a sua idade: ")
2 idade = int(idade)
```

Assim, nosso programa fica conforme exibido no Código 2.2:

```
1 nome = input("Digite o seu nome: ")
2 idade = input("Digite a sua idade: ")
3 idade = int(idade)
4
5 print("Ola ", nome, "!")
6 print("Voce tem", idade, "anos!")
```

Código 2.2: segundo programa em Python.

A seguir, temos um exemplo de execução do Código 2.2 (os textos em laranja são as entradas do usuário, ao passo que os textos em preto são impressos pelo próprio programa):

```
Digite o seu nome: Mayara
Digite a sua idade: 25
Ola  Mayara !
Voce tem 25 anos!
```

É importante destacar que, no Python 3, a função `input` **sempre** retorna um objeto do tipo `str` com o exato conteúdo digitado pelo usuário. Por conta disso, quando precisarmos trabalhar com um dado numérico lido através dessa função, será necessário realizar sua conversão para algum tipo numérico conforme o exemplo anterior, onde foi realizada a conversão do dado lido na variável `idade` (linha 3). É um erro comum, especialmente entre principiantes, o esquecimento dessa conversão.

2.9 Operações básicas com números

É oportuno salientar que quando realizamos uma operação no prompt sem atribuir o resultado a uma variável, o interpretador ecoa (exibe) o resultado da operação no próprio prompt, conforme o exemplo:

```
>>> 2 + 3
5
```

Observe que, no exemplo anterior, realizamos a operação `2 + 3`. Como não atribuímos o resultado a nenhuma variável, o interpretador nos ecoou sem resultado logo na linha abaixo (`5`).

Listamos a seguir algumas operações básicas, com seus respectivos operadores com tipos numéricos em Python:

- **soma**: `+`

```
>>> k = 7
>>> k + 3
10
```

- **subtração**: `-`

```
>>> k = 5
>>> w = k - 1
>>> w
4
```

- **multiplicação**: `*`

```
>>> 2.5 * 3
7.5
```

- **divisão**: `/` . **Atenção:** em versões anteriores a 3, assume-se que a divisão entre dois números do tipo `int` deve ter como resultado um número também do tipo `int`. Assim, o resultado da divisão é arredondado para baixo. Portanto, nessas versões, dividir 5 por 2 terá como resultado 2. Por sua vez, dividir -5 por 2 terá como resultado -3. Assim, se o objetivo é realizar uma divisão exata nesse contexto, é necessário garantir que, ao menos um dos operandos (numerador ou denominador) seja do tipo `float`.

```
#executando divisão no Python 2.7
>>> a = 7
>>> b = 2
>>> a/b     #assume-se que como os operandos são int,
    resultado deve ser int também no Python 2.x
3
>>> float(a)/float(b)  #para obter o resultado com as
    devidas casas decimais, é preciso converter ao menos um
     dos operandos para float no Python 2.x.
3.5
```

  A partir do Python 3, o resultado da divisão entre dois números `int` é sempre `float`, o que permite apresentar o resultado com as casas decimais.

```
#executando divisão no Python 3.6
>>> a = 7
>>> b = 2
>>> a/b      #A partir da versão 3, o resultado da divisão
    entre dois ints é float.
3.5
```

Por via de dúvidas, de modo a construir um código fonte que seja compatível com as versões do Python 2 e 3, é recomendável sempre converter ao menos um dos operandos para `float` para garantir o resultado com casas decimais:

```
#executando divisão no Python 3.6
>>> a = 7
>>> b = 2
>>> float(a)/b     #convertemos ao menos um dos operandos
    para float para manter compatibilidade com Python 2.
3.5
```

Por sua vez, se o objetivo é de fato obter o resultado da divisão como número inteiro, é recomendável utilizar a operação de divisão inteira (divisão na base), a seguir.

- **quociente da divisão inteira (divisão na base)**: `//` . A operação de divisão na base é voltada para dividir números inteiros, obtendo o quociente da divisão e ignorando o resto. Por exemplo, o resultado da divisão na base de 9 por 2 é 4, pois 4 é o quociente desta divisão. Observe que o resto da divisão e possíveis casas decimais do quociente serão ignorados.

```
>>> 9 // 2
4
```

- **resto da divisão inteira**: `%` . Esse operador também é voltado para a divisão de números inteiros, porém, aqui se obtém o resto da divisão no lugar do quociente.

```
>>> 11 % 3      # calcula o resto da divisão de 11 por 3
2
```

- **potenciação**: `**` .

```
>>> w = 5
>>> w ** 2       # eleva w ao quadrado
25
```

Uma vez que a radiciação também pode ser vista como uma forma de potenciação, podemos utilizar esse operador também para essa finalidade.

```
>>> 36 ** 0.5      #eleva 36 a 0,5 , isto é, calcula a raiz
    quadrada de 36.
6.0
```

Existe uma espécie de hierarquia dentre os tipos numéricos em Python. O tipo `complex` tem precedência sobre o tipo `float`, que, por sua vez, tem precedência sobre `int`. Ao se realizar uma operação entre números, o resultado terá

o mesmo tipo predominante dentre os operandos. Por exemplo, na soma entre um `complex` e um `int`, o resultado será um `complex`. Na multiplicação entre um `int` e um `float`, o resultado será um `float`. A exceção a essa regra é a operação de divisão entre dois objetos `int` a partir da versão 3, onde o resultado será sempre `float`.

Também há uma espécie de hierarquia dentre as operações. A operação de potenciação tem precedência sobre a de divisão, que, por sua vez, tem precedência sobre a de multiplicação, que, por sua vez, tem precedência sobre soma e subtração. Um erro comum entre iniciantes é não levar essa precedência em conta ao formular suas expressões. Por exemplo, vamos supor que se deseje calcular o resultado de $\frac{14+6}{2}$. Uma primeira tentativa descuidada seria:

```
>>> 14 + 6 / 2  #expressão equivocada
17.0
```

No entanto, a mesma produziria um resultado equivocado, pois, uma vez que a divisão tem precedência sobre soma, a operação $6/2$ seria realizada inicialmente, para em seguida, ter seu resultado somado a 14, o que resultaria em 17. Para indicar a ordem correta das operações, pode-se utilizar pares de parênteses:

```
>>> (14 + 6)/2
10.0
```

Os parênteses na expressão anterior indicam que a operação de soma deve ser realizada antes das operações de fora dos parênteses. **Observação:** apenas parênteses podem ser utilizados para indicar precedência de operações. Colchetes e chaves, por exemplo, têm outras finalidades em Python. É permitido, todavia, o uso de um número arbitrário de pares de parênteses, como no exemplo a seguir:

```
>>> (( a + b )* c ) ** (d - e)
```

2.9.1 Atribuições ampliadas

Na elaboração de programas, é muito comum a escrita de expressões como:

```
b = b + 1
```

isto é, expressões onde uma variável é atualizada em função do seu próprio valor corrente. Para facilitar a escrita dessas operações, Python incorporou o conceito de atribuição ampliada. Assim, usando atribuição ampliada, a expressão anterior poderia ser escrita como

```
b += 1       #equivalente a:  b = b + 1
```

Outros exemplos de atribuições ampliadas são:

```
c -= b       #equivalente a:  c = c - b
d *= 2       #equivalente a:  d = d * 2
e /= 3       #equivalente a:  e = e/3
f **= 0.5    #equivalente a:  f = f ** 0.5
g %= h       #equivalente a:  g = g % h
```

2.9.2 Exemplo com operações aritméticas

Agora que já sabemos como tratar entradas e saída e conhecemos as operações numéricas, podemos construir programas envolvendo cálculos. O exemplo a seguir solicita ao usuário o valor do raio de um círculo e, a partir deste valor, informa seu diâmetro, perímetro e área. Lembrando um pouco de geometria, dado um círculo de raio r, o diâmetro é calculado como sendo $2r$, o perímetro é dado pela expressão $2\pi r$ e a área é dada por πr^2.

```
1  #programa que lê o valor do raio de um círculo e informa:
2  #1 - O valor do seu diâmetro
3  #2 - O valor do seu perímetro
4  #3 - O valor da sua área
5
6  raio = input("Entre com o valor do raio do circulo: ")
7  raio = float(raio) #como raio é um numero real, realizamos a
       conversão para float
8
9  #definimos uma variável com o valor de pi para facilitar os
       cálculos
10 pi = 3.141592
11
12 diametro = 2*raio
13 perimetro = 2*pi*raio
14 area = pi*(raio**2)
15
16 #imprimindo os resultados:
17 print("Valor do diametro:", diametro)
18 print("Valor do perimetro:", perimetro)
19 print("Valor da area:", area)
20
21 print("Tenha um bom dia!")
```

Código 2.3: programa que calcula diâmetro, perímetro e área de um círculo a partir de seu raio.

Começamos o programa com comentários sobre sua finalidade nas linhas 1-4. Lembramos que tudo o que estiver em uma linha depois de um caracter `#` é ignorado pelo interpretador Python (comentário). O passo seguinte, é obter, do usuário o valor do raio (linha 6) e converter o objeto `str` lido para número `float` (linha 7). Em seguida, começa a etapa da realização dos cálculos. Inicialmente, definimos uma variável para armazenar o valor de pi (linha 10). Poderíamos não ter definido a variável `pi` e usado seu valor diretamente nas linhas 12-14, mas optamos por essa forma para facilitar a escrita das expressões. Desse modo, os cálculos do diâmetro, perímetro e área são realizados nas linhas 12, 13 e 14, respectivamente. A última etapa do programa é a impressão dos resultados para o usuário, junto com uma mensagem de saudação, nas linhas 17-21. As linhas em branco foram colocadas no código fonte apenas para deixá-lo visualmente mais organizado e não produzem qualquer efeito sobre a execução do programa. Pessoas acostumadas a outras linguagens de programação podem notar a não presença de caracter de fim de linha no código em Python. Na realidade, Python nos fornece a opção de terminar ou não cada linha por um caracter `;` (ponto e vírgula), o que é especialmente útil para quem está acostumado com linguagens como C e Java. O ponto e vírgula também nos permite especificar mais de uma instrução em uma mesma linha.

A seguir, temos um exemplo de execução do programa. O texto em laranja

indica informações digitas pelo usuário, ao passo que o texto em preto é impresso pelo programa.

```
Entre com o valor do raio do circulo:  5
Valor do diametro: 10.0
Valor do perimetro: 31.41592
Valor da area: 78.5398
Tenha um bom dia!
```

2.10 Exercícios

1. O Índice de Massa Corporal (IMC) é calculado dividindo-se o peso de um indivíduo pelo quadrado de sua altura. Faça um programa que leia o peso e a altura de um indivíduo e informe seu IMC.

   Exemplo:

   ```
   Entre com o peso: 115
   Entre com a altura: 2
   IMC: 28.75
   ```

2. Escreva um programa em Python que leia dois pares ordenados do plano cartesiano (x, y) e imprima:

   - a distância entre esses dois pontos;
   - o coeficiente angular da reta que passa por ambos os pontos;
   - o coeficiente linear da reta que passa por ambos os pontos.

   Exemplo:

   ```
   Entre com x no primeiro ponto: 1
   Entre com y no primeiro ponto: -4.5

   Entre com x no segundo ponto: -2
   Entre com y no segundo ponto: 10

   Distancia entre os pontos: 14.807
   Coeficiente angular da reta: -4.8333
   Coeficiente linear da reta: 0.33333
   ```

3. Escreva um programa que calcule as duas raízes da equação de segundo grau: $ax^2 + bx + c = 0$ (note que as raízes podem ser exibidas como números complexos). Seu programa deverá ler os valores de a, b e c do teclado. Dica: calcule $\Delta = b^2 - 4ac$ sempre como número complexo.

   Exemplo:

```
Entre com o valor de a: 1
Entre com o valor de b: 1
Entre com o valor de c: -2

Delta: 9+0j
Raizes: -2+0j e 1+0j
```

4. Um trem de longa distância parte de uma estação inicial em um determinado horário H_1 (hora e minuto) e chega à estação final em um determinado horário H_2. Faça um programa em Python que leia os horários de partida e chegada do trem e informe o tempo total de viagem (em hora e minuto). Assuma que as viagens sempre iniciam e terminam no mesmo dia.

   Exemplo:

```
Entre com a hora de partida: 10
Entre com o minuto de partida: 35

Entre com a hora de chegada: 11
Entre com o minuto de chegada: 25

O trem partiu as 10:35 e chegou as 11:25.
Tempo de viagem: 00:50
```

Capítulo 3

Expressões Booleanas e Condicionais

3.1 Operadores Lógicos

Nessa seção, discutimos os operadores que implementam funções lógicas básicas. Estas funções lógicas operam a partir de valores que podem ser considerados como verdadeiros ou falsos, e, a depender destes, fornecem como resposta um valor booleano, isto é, um valor que pode ser `True` (verdadeiro) ou `False` (falso).

3.1.1 Operador or

O operador `or` implementa a função lógica OU. `X or Y` retorna verdadeiro se `X` for verdadeiro ou `Y` for verdadeiro. A Tabela 3.1 enumera os resultados da operação `X or Y` para todas as diferentes possibilidades de `X` e `Y`. A esse tipo de tabela, onde enumeramos possibilidades de entrada e saída de uma expressão lógica, damos o nome de *tabela verdade*.

`X`	`Y`	`X or Y`
Falso	Falso	Falso
Falso	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Verdadeiro	Verdadeiro	Verdadeiro

Tabela 3.1: tabela verdade do operador *or* (função lógica OU).

Note que o tipo `bool` (booleano) foi criado justamente para representar verdadeiro (com o valor `True`) e falso (com o valor `False`). Desse modo, o uso do operador `or`, e dos operadores lógicos em geral, é mais comum com esse tipo de objeto:

```
>>> X = True
>>> Y = False
>>> X or Y
True
>>> Y or False
False
>>> False or True
True
```

Todavia, é válido destacar que não são apenas valores do tipo `bool` que são interpretado como verdadeiros ou falsos. Valores dos demais tipos nativos do Python também podem ser interpretados como verdadeiros ou falsos de acordo com as seguintes regras:

- Objeto `None`:
  - Sempre é considerado falso.
- Objetos numéricos:
  - São considerados falsos se estiverem com o valor zero;
  - São considerados verdadeiros se estiverem com qualquer valor diferente de zero.
- Demais objetos:
  - São considerados falsos se estiverem vazios;
  - São considerados verdadeiros se não estiverem vazios.

Assim, temos os seguintes exemplos:

```
>>> a = ""          #a é falso, pois seu valor é a string vazia
>>> b = "jessica" #b é verdadeiro, pois seu valor é uma string
    não vazia
>>> c = 0           #c é falso, pois seu valor é o numero zero
>>> d = 1991        #d é verdadeiro, pois seu valor é um número
    diferente de zero
>>> e = []          #e é falso, pois seu valor é uma lista vazia
>>> f = [1]         #f é verdadeiro, pois seu valor é uma lista
    não vazia
>>> g = None        #g é falso, pois seu valor é o None
```

De modo a economizar esforço, se o primeiro valor recebido pelo operador `or` já for verdadeiro, o operador o dá como resposta sem sequer avaliar o segundo valor. Por exemplo:

```
>>> d or b
1991
```

No exemplo anterior, pelo fato do valor em `d` já ser considerado verdadeiro, o interpretador Python já responde imediatamente seu valor sem sequer olhar o segundo operando (`b`). Note que a resposta foi o próprio valor de `d`, que deve então ser interpretado por nós como verdadeiro. Outros exemplos:

```
>>> a or c      #ambos são falsos. Então o operador or responderá
    o valor de c, que deve ser interpretado por nós como falso
0
```

```
>>> f or c
[1]
```

Assim, concluímos que o operador `or` sempre responderá como resposta um dos valores recebidos como entrada.

3.1.2 Operador and

O operador `and` implementa a função lógica E. `X and Y` retorna verdadeiro se `X` e `Y` forem ambos verdadeiros. A Tabela 3.2 enumera os resultados da operação `X and Y` para todas as diferentes possibilidades de `X` e `Y`:

X	Y	X and Y
Falso	Falso	Falso
Falso	Verdadeiro	Falso
Verdadeiro	Falso	Falso
Verdadeiro	Verdadeiro	Verdadeiro

Tabela 3.2: tabela verdade do operador *and* (função lógica E).

As mesmas regras vistas com o operador `or` para considerar um objeto como verdadeiro ou falso são válidas aqui. De modo a economizar esforço, se o primeiro valor recebido pelo operador `and` já for falso, o operador o dá como resposta sem sequer avaliar o segundo valor. Por exemplo:

```
>>> X = False
>>> Y = True
>>> X and Y
False
```

```
>>> g = None
>>> f = [1]
>>> g and f    # como o valor de g já é considerado falso, o
    operador and o dá como resposta sem sequer avaliar o valor
    de f. Esta resposta (None) deve ser encarada como falso.
None
```

```
>>> d = 1991
>>> b = "jessica"
>>> d and b    # como o valor de b é considerado verdadeiro, o
    operador and olhará o segundo operando (b). Como ele também
    é verdadeiro, seu valor será dado como resposta, a qual
    devemos encarar como verdadeiro.
"jessica"
```

3.1.3 Operador not

O operador `not` implementa a função lógica NÃO, que atua fornecendo o resultado oposto ao recebido como entrada. Assim, o operador `not` atua como um inversor do seu operando. Se `X` for verdadeiro, `not X` responderá `False`. Se `X` for falso, `not X` responderá `True`, como pode ser conferido na tabela 3.3

É possível observar que o operador `not` tem duas peculiaridades em relação aos operadores `or` e `and`. A primeira delas é que ele só recebe um único valor

X	not X
Falso	Verdadeiro
Verdadeiro	Falso

Tabela 3.3: tabela verdade do operador *not* (função lógica NÃO).

como entrada, ao passo que os demais recebem dois. A segunda, é que o operador `not` só fornece valores booleanos como resposta, isto é, só responde `True` ou `False`, conforme os exemplos:

```
>>> X = False
>>> not X
True
```

```
>>> b = "jessica"
>>> not b
False
```

```
>>> not 27
False
```

3.1.4 Considerações sobre o uso de operadores lógicos

É possível misturar diferentes operadores lógicos em uma mesma. Todavia, nesse caso, é altamente recomendável o uso de parênteses, pois os operadores tem diferentes prioridades (precedências) de modo que é muito fácil confundirmos seu uso. O operador `not` tem prioridade sobre o operador `and`, que por sua vez tem prioridade sobre o operador `or`. Veja os exemplos:

```
>>> a = True
>>> b = True
>>> c = False
>>> a or b and c  # a operação and é realizanda antes da or
True
```

Note, no exemplo anterior, que a última parte da expressão lógica (`and`) foi realizada antes da primeira (`or`), de forma equivalente à seguinte:

```
>>> a or (b and c)
True
```

Para reverter essa ordem é necessário o uso de parênteses na primeira expressão:

```
>>> (a or b) and c
False
```

3.2 Operadores de Comparação

Existem operadores que tem o objetivo de comparar os valores de objetos. Eles retornam o valor booleano `True` (verdadeiro) quando a comparação realizada de fato se verifica, e `False` (falso) caso contrário. A seguir, a lista de operadores:

- **Igualdade: `==`**

  A operação `a == b` tem como resultado `True` se o valor de `a` for considerado igual (equivalente) ao de `b`, e `False` caso contrário. Veja os exemplos a seguir:

```
>>> a = 3
>>> b = 4
>>> a == 3
True
>>> a == b
False
>>> a + 1 == b
True
```

É importante não confundir o uso dos operadores = (atribuição) e == (comparação de igualdade). O operador = é destinado a atribuição de um objeto (valor) à uma variável, podendo assim mudar o estado corrente da mesma. Usamos o operador = para criar novas variáveis ou para "modificar" o seu valor. Por sua vez, o operador == tem o objetivo apenas de comparar se os valores de dois objetos são equivalentes. O operador == é usado apenas para perguntar se um objeto tem valor equivalente a outro, e não altera o conteúdo dos objetos nativos em Python.

Os seguintes operadores de comparação têm uso semelhante:

- **Diferença**: **!=**

```
>>> c = 7
>>> d = 9
>>> c != d
True
>>> c != 3
True
>>> c != d - 2
False
```

- **Menor**: <

```
>>> e = 9
>>> f = 6
>>> 2 < 3
True
>>> e < 4
False
>>> f < e
True
```

- **Maior**: >

```
>>> g = 7
>>> h = 20
>>> g > h
False
>>> g > -2
True
```

- **Menor ou igual que**: <=

```
>>> p = 14
>>> m = 100
>>> m >= p
True
>>> p >= 15
False
```

- **Maior ou igual que**: `>=`

```
>>> q = 16
>>> r = 8
>>> q >= r
False
>>> q >= 2*r
True
```

É válido dizer que os operadores de comparação possuem precedência mais alta que os operadores lógicos. Assim, podemos testar se uma variável `n` está com valor entre 0 e 100 fazendo:

```
>>>  n >= 0 and n <= 100
```

Para a realização de diversas comparações de igualdade ou desigualdade em sequência, é interessante o uso dos operadores `in` e `not in`:

- **Participação como membro de sequência**: `in`

  O operador `in` retorna `True` se um objeto aparece em um objeto sequencial, isto, é, se o seu valor equivale ao valor de um membro de uma sequência. Por exemplo, vamos supor que queremos saber se o valor de uma variável `a` é um número primo menor que 10. Nesse caso, poderíamos, por exemplo, compará-lo com os elementos do conjunto $\{2, 3, 5, 7\}$ usando os operadores `==` e `or`:

```
>>> a = 4
>>> a == 2  or  a == 3  or  a == 5  or  a == 7
False
```

  ou, de modo mais fácil, podemos usar o operador `in` com uma sequência que inclua exatamente os números desejados. Podemos utilizar, por exemplo, uma tupla (`tuple`):

```
>>> a = 4
>>> a in (2,3,5,7)
False
```

  Outros exemplos:

```
>>> 3 in (2,3,5,7)
True
```

```
>>> "w" in "mariana"
False
```

```
>>> "j" in "jessica"
True
```

É importante ressaltar que o operador `in` só se aplica a sequências (objetos iteráveis). Assim, o objeto a direita do operador precisa ser uma sequência, como, por exemplo, strings, tuplas, listas, dicionários, conjuntos, etc. O operador não funcionará se você quiser saber, por exemplo, se o dígito 2 aparece no número 726:

```
>>> 2 in 726   # Erro, pois o objeto a direita (726) não é uma sequência
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: argument of type 'int' is not iterable
```

Todavia, a operação dará certo se você fizer conversões para string:

```
>>> "2" in "726"  #operação válida, pois string é um objeto sequencial
True
```

Note que foi preciso converter ambos os objetos para string no exemplo anterior.

- **Não participação como membro de sequência**: `not in`

  O operador `not in` funciona de forma análoga ao operador `in`, no entanto, este se destina a verificar se o valor de um objeto não é equivalente ao valor de qualquer objeto da sequência:

```
>>> 4 not in (2,3,5,7)
True
```

```
>>> "p" not in "jessica"
True
```

```
>>> 7 not in (2,3,5,7)
False
```

3.3 Condicionais *if*

Instruções condicionais são fundamentais na elaboração de programas de computador, pois permitem a escolha entre um ou mais fluxos (caminhos) de execução baseado em resultados de testes. Veremos aqui as possíveis formas da cláusula `if`, que é a cláusula para condicionais em Python.

3.3.1 Primeira forma geral da cláusula *if*

A primeira forma geral de uso da cláusula `if` está descrita a seguir:

```
if <teste>:
<tab> <instrução 1>
<tab> <instrução 2>
<tab> <instrução 3>
.
.
.
<tab> <instrução n>

<primeira instrução pós-if>
```

Código 3.1: primeira forma da cláusula *if*.

Onde `<teste>` (primeira linha) representa qualquer expressão (teste) cujo resultado possa ser interpretado como verdadeiro ou falso e `<tab>` representa um caracter de tabulação (tecla *tab* do teclado). Observe que após a primeira linha com a cláusula if, há uma sequência de n instruções precedidas por `<tab>`. O interpretador considera que cada uma dessas instruções precedidas por um `<tab>` está condicionada à clausula `if` da linha 1 (que está sem o `<tab>` na frente), o que significa que essas n instruções só podem ser executadas se `<teste>` resultar em verdadeiro. Se `<teste>` resultar em falso, essas n instruções **não** serão executadas e o programa saltará diretamente para a execução da primeira linha com instrução pós-if, que é a primeira linha após o `if` sem o caracter `<tab>` na frente (última linha).

Como exemplo, vamos fazer um programa em Python que lê uma nota de um aluno (entre 0 e 10) e informa se a nota é vermelha, isto é, se a nota está abaixo de 5.0:

```
1  #primeiro exemplo de uso de condicional if
2
3  nota = input("Digite o valor da nota: ")
4  nota = float(nota)
5
6  if nota < 5.0:
7      print("Nota vermelha!")
8      print("Precisa estudar mais!")
9      print("E se esforcar!")
10
11 print("Tenha um bom dia!")
```

Código 3.2: primeiro exemplo de uso de condicional *if*.

Após as linhas inicias de comentário e em branco, o programa começa sua execução na linha 3, solicitando que o usuário forneça o valor de uma nota. Após a leitura do dado, que virá como uma string (`str`), realizamos a conversão para número real (`float`) na linha seguinte. A seguir, vem uma cláusula `if` na linha 6, que testa a condição `nota < 5.0`. Apenas se este teste resultar em verdadeiro, as três linhas seguintes (7-9) serão executadas. Ressaltamos mais uma vez que o especifica que essas linhas estão condicionadas à cláusula `if` é a presença do caracter `<tab>` em seu início, o que faz com que as mesmas estejam mais a direita em comparação com a linha do `if` (linha 6). Se o valor da variável nota for superior a 5.0, as linhas 7-9 nãos serão executadas, e o programa saltará diretamente para a linha 11, que é a primeira linha após o `if`, pois é a primeira linha depois do `if` não precedida pelo caracter `<tab>`. Novamente, linhas em branco não causam qualquer efeito no código fonte e só estão presentes para uma melhor leitura do código por parte de seres humanos. Exemplos de execução desse código seriam:

```
Digite o valor da nota: 4.5
Nota vermelha!
Precisa estudar mais!
E se esforcar!
Tenha um bom dia!
```

Agora um exemplo de execução com nota superior a 5.0:

```
Digite o valor da nota: 6.8
```

```
Tenha um bom dia!
```

Note que a linha 12 é executada em qualquer situação onde o usuário fornece uma nota válida, pois a mesma não está vinculada a uma cláusula `if`. Você pode rodar o programa acima na IDLE do Python (veja seção 2.8) e testar sua execução para diferentes valores de nota.

É importante ressaltar que o caractere `<tab>` tem o objetivo de declarar ao interpretador Python o bloco de instruções que está condicionado ao `if`. Na prática, esse caracter fará com que esse bloco de instruções apareça mais à direita em relação a linha do `if` no código fonte, o que facilita a visualização e a compreensão do código por parte de seres humanos. A essa prática de deixar um bloco de código mais a direita é dado o nome de *indentação*, e, na linguagem Python, é o único modo de condicionar um bloco de instruções à uma cláusula. Na realidade, o interpretador Python não requer que se use exatamente um caracter `<tab>`, mas sim que o bloco de código esteja indentado em relação à cláusula subordinada, o que pode ser feito, por exemplo com quatro caracteres de espaço, dois caracteres de espaço, ou um número arbitrário de caracteres de espaço ou `<tab>` antes das instruções, desde que todo o bloco seja precedido exatamente pela mesma quantidade de espaços ou `<tab>`.

Um aluno mais caprichoso poderia se incomodar com o fato do programa exemplo anterior chamar a atenção de um aluno que tirou nota vermelha mas não dizer absolutamente nada para o aluno que tirou nota azul. Poderíamos agradar a esse aluno então com a introdução de outro `if`:

```
1  #segundo exemplo de uso de condicional if
2
3  nota = input("Digite o valor da nota: ")
4  nota = float(nota)
5
6  if nota < 5.0:
7      print("Nota vermelha!")
8      print("Precisa estudar mais!")
9      print("E se esforcar!")
10
11 if nota >= 5.0:
12     print("Nota azul!")
13     print("Parabens!")
14
15 print("Tenha um bom dia!")
```

Código 3.3: segundo exemplo de uso de condicional *if*, com aninhamento.

Note que o exemplo anterior utiliza duas cláusulas que são independentes entre si. Você está convidado a testar sua execução para diferentes valores de notas. Para facilitar a escrita de código em casos como esse, veremos a segunda forma geral da cláusula `if`.

3.3.2 Segunda forma geral da cláusula *if*

A segunda forma geral da cláusula `if` está a seguir:

```
1  if <teste>:
2  <tab> <instrução 1>
3  <tab> <instrução 2>
4  .
5  .
6  .
7  <tab> <instrução n>
8  else:
9  <tab> <instrução n+1>
10 <tab> <instrução n+2>
11 .
12 .
13 .
14 <tab> <instrução n+m>
15 <primeira instrução pós-if>
```

Código 3.4: segunda forma da cláusula *if*, com cláusula else.

Essa nova forma introduz uma nova cláusula denominada `else`, que é complementar a `if`. O objetivo dessa nova cláusula é especificar um bloco de instruções que deve ser executado no caso de `<teste>` resultar em falso na linha 1. Assim, o programa deverá executar as instruções (1)-(n) se `<teste>` resultar verdadeiro, e as instruções (n+1)-(n+m) se `<teste>` resultar em falso. Note que assim, não é possível executar ambos os blocos de instruções em uma mesma execução desse trecho de código fonte. Apenas um deles será executado, o que determinará essa escolha será o `<teste>` declarado logo após a cláusula `if`. Novamente, o que determina que cada instrução esteja vinculada a uma cláusula `if` ou `else`, é a presença do caracter de tabulação (`<tab>`) em seu início. Note que a cláusula `else` está no mesmo nível de `if`, isto é, não está precedida por `<tab>`. Desse modo, o interpretador Python saberá que esse `else` se remete a cláusula `if` imediatamente anterior que esteja no mesmo nível. Assim, toda cláusula `else` precisa estar vinculada a algum `if` [1].

Podemos então utilizar essa nova forma para construir um exemplo onde cumprimentamos um aluno que tenha obtido nota azul:

```
1  #primeiro exemplo de uso de condicional if-else
2
3  nota = input("Digite o valor da nota: ")
4  nota = float(nota)
5
6  if nota < 5.0:
7      print("Nota vermelha!")
8      print("Precisa estudar mais!")
9      print("E se esforcar!")
10 else:
11     print("Nota azul!")
12     print("Parabens!")
13
14 print("Tenha um bom dia!")
```

Código 3.5: primeiro exemplo de uso de condicional *if-else*.

Exemplos de execução:

[1] Na realidade, veremos mais adiante que *else* também pode estar vinculado à outras cláusulas como *while*, *for* e *try*. Todavia, o que queremos enfatizar é que uma cláusula *else* não pode aparecer "solta" no código. Ela precisa estar vinculada a uma dessas cláusulas.

```
Digite o valor da nota: 6
Nota azul!
Parabens!
Tenha um bom dia!
```

```
Digite o valor da nota: 2.1
Nota vermelha!
Precisa estudar mais!
E se esforcar!
Tenha um bom dia!
```

A esse momento, deve haver, em algum lugar, algum estudante cético se perguntando como nosso programa se comporta se o usuário digitar uma nota inválida, como por exemplo, um número negativo, ou um número superior a 10. Bom, poderíamos agradar a esse estudante utilizando `if` aninhados:

```
1  #segundo exemplo de uso de condicional if-else
2
3  nota = input("Digite o valor da nota: ")
4  nota = float(nota)
5
6  if 0.0 <= nota and nota <= 10.0:
7
8      if nota < 5.0:
9          print("Nota vermelha!")
10         print("Precisa estudar mais!")
11         print("E se esforcar!")
12     else:
13         print("Nota azul!")
14         print("Parabens!")
15 else:
16
17     print("Você digitou uma nota inválida")
18
19 print("Tenha um bom dia!")
```

Código 3.6: segundo exemplo de uso de condicional *if-else*.

Note que no exemplo anterior, no bloco de instruções do primeiro `if` (linha 6), existe outro `if` (linha 8), o que é totalmente válido. Observe que esse segundo `if` possui um caracter `<tab>` a sua frente, pois o mesmo se encontra dentro do bloco de instruções de outro `if`. Por consequência, o bloco de instruções vinculado a esse `if` (linhas 9-11) precisa ter dois caracteres `<tab>` a sua frente, pois este bloco precisa estar mais a direita do que seu `if` (linha 8), que já está precedido por `<tab>`. O mesmo é válido para o `else` na linha 12 e seu bloco de instruções vinculado (linhas 13-14). Como este `else` está com `<tab>` a sua frente, o interpretador Python saberá que ele está vinculado ao `if` da linha 8, que também possui um `<tab>`, e não ao `if` da linha 6, que não está precedido por `<tab>`. Por essa mesma razão, sabe-se que o `else` da linha 15 está vinculado ao `if` da linha 6.

Note então que, para que o teste da linha 8 (`nota < 5.0`) seja executado, é necessário que o teste da linha 6 (`0.0 <= nota and nota <= 10.0`) tenha resultado em verdadeiro, pois, do contrário, o programa pulará diretamente para a linha 17. Por ser uma linguagem amigável, Python permite que o teste na linha 6 seja escrito como `0.0 <= nota <= 10.0`. Mais uma vez, você está encorajado a executar esse programa e testar diferentes valores de nota. Para facilitar a codificação de casos como esse, veremos a terceira forma geral de uso de `if`, que é a forma mais genérica.

3.3.3 Terceira forma geral da cláusula *if*

A terceira forma geral de uso de `if`, que é a forma mais genérica, é dada a seguir:

```
1  if <teste 1>:
2  <tab> <bloco de instruções 1>
3
4  elif <teste 2>:
5  <tab> <bloco de instruções 2>
6
7  elif <teste 3>:
8  <tab> <bloco de instruções 3>
9  .
10 .
11 .
12 elif <teste n>:
13 <tab> <bloco de instruções n>
14
15 else:
16 <tab> <bloco de instruções n+1>
17
18 <primeira instrução pós-if>
```

Código 3.7: forma mais genérica da cláusula *if*, com cláusulas *elif* e *else*.

Para facilitar o entendimento, aqui, `<bloco de instruções x>` representa um bloco com diversas instruções. Observe que aqui, foi introduzida a cláusula `elif`, que pode aparecer um número arbitrário de vezes e permite a construção de uma hierarquia de testes, cada qual com seu bloco de instruções associado. **É preciso ressaltar que, no máximo, um dos blocos de instruções será executado.** Todos os testes serão realizados, na respectiva ordem de declaração até que o primeiro teste resulte em verdadeiro. Assim, o bloco associado ao teste verdeiro será executado, e o programa pulará a seguir para a primeira instrução após o `if` sem executar os demais testes. Assim, primeiramente, `<teste 1>` será avaliado. Se resultar em verdadeiro, `<bloco de instruções 1>` será executado. Caso contrário, `<teste 2>` será avaliado. Se resultar em verdadeiro, `<bloco de instruções 2>` será executado. Caso contrário, `<teste 3>` será avaliado, e assim sucessivamente. **Note então que, para que o teste `n` seja executado, é necessário que todos os `n-1` testes anteriores resultem em falso.** Se todos os `n` testes derem falso e houver uma cláusula `else` ao final, o bloco de instruções vinculado a esta cláusula (`<bloco de instruções n+1>`) será executado. A presença dessa cláusula `else` com seu respectivo bloco é opcional, e sua ausência pode fazer com que nenhum bloco de instruções seja executado, caso todos os `n` testes resultem em falso. Portanto, a presença de `else` garante que exatamente um bloco de instruções será executado, que será o bloco correspondente ao primeiro teste que der verdadeiro, ou o bloco correspondente ao `else` se todos os testes resultarem em falso.

No exemplo a seguir, exibiremos uma congratulação especial para os alunos que tiraram nota acima de 8.0 e exibiremos uma mensagem de erro ao usuário caso ele não digite uma nota entre 0 e 10.

```
#exemplo de condicional if-elif-else

nota = input("Digite o valor da nota: ")
nota = float(nota)

if 0 <= nota < 5.0:
    print("Nota vermelha!")
    print("Precisa estudar mais!")
    print("E se esforcar!")

elif 5.0 <= nota < 8.0:
    print("Nota azul!")
    print("Parabens!")

elif 8.0 <= nota <= 10.0:
    print("Nota super alta!")
    print("Hiper parabens!")
    print("Nerd detectado!")

else:
    print("Nota invalida!")

print("Tenha um bom dia!")
```

Código 3.8: exemplo de uso de condicional *if-elif-else*.

Observe que o exemplo anterior com cláusulas `elif` é de mais fácil construção e entendimento que o Código 3.6, que utiliza cláusulas `if` aninhadas. Exemplos de execução desse programa vêm a seguir:

```
Digite o valor da nota: -3
Nota invalida!
Tenha um bom dia!
```

```
Digite o valor da nota: 9.4
Nota super alta!
Hiper parabens!
Nerd detectado!
Tenha um bom dia!
```

3.4 Exercícios

1. Escreva um programa que leia um número inteiro entre 1 e 12 representando um mês e imprima se este mês tem 28, 30 ou 31 dias. Assuma, conforme a tabela 3.4, que fevereiro sempre tem 28 dias.

Mês	**Dias**
Jan, Mar, Mai, Jul, Ago, Out, Dez	31
Abr, Jun, Set, Nov	30
Fev	28

Tabela 3.4: número de dias em cada mês

Exemplos:

```
Digite o numero do mes: -8
Mes invalido!
```

```
Digite o numero do mes: 12
Mes com 31 dias!
```

```
Digite o numero do mes: 19
Mes invalido!
```

2. Escreva um programa que leia um número inteiro positivo do teclado e informe se ele é par ou é ímpar. Nota: um número é par se o mesmo é divisível por dois, isto é, se o resto da divisão do número por 2 é 0.

   Exemplos:

```
Digite um numero: 3
Numero impar!
```

```
Digite um numero: 18
Numero par!
```

3. Escreva um programa que leia os comprimentos dos lados de um triângulo e informe se o triângulo é equilátero, isósceles ou escaleno.

   Exemplos:

```
Digite o comprimento do primeiro lado do triangulo:  7
Digite o comprimento do segundo lado do triangulo: 9
Digite o comprimento do terceiro lado do triangulo: 7

Este triangulo e isosceles.
```

```
Digite o comprimento do primeiro lado do triangulo: 7
Digite o comprimento do segundo lado do triangulo: 5
Digite o comprimento do terceiro lado do triangulo: 5

Este triangulo e isosceles.
```

```
Digite o comprimento do primeiro lado do triangulo: 3
Digite o comprimento do segundo lado do triangulo: 4
Digite o comprimento do terceiro lado do triangulo: 5

Este triangulo e escaleno.
```

```
Digite o comprimento do primeiro lado do triangulo: 8
Digite o comprimento do segundo lado do triangulo: 8
Digite o comprimento do terceiro lado do triangulo: 8

Este triangulo e equilatero.
```

4. A *Bachatóvia* adota a Tabela 3.5 para o cálculo do seu imposto de renda. Faça um programa que peça a renda anual de um contribuinte e calcule o seu devido imposto, de acordo com a tabela.

Faixa	**Imposto**	**Sobre valor superior a**
$Renda \leq 21450.00$	15%	0.00
$21450.00 < Renda \leq 51900.00$	$3117.50 + 28\%$	21450.00
$Renda > 51900.00$	$11743.00 + 31\%$	51900.00

Tabela 3.5: tabela de imposto de renda da Bachatóvia

Exemplos:

```
Digite a sua renda anual: -20
Renda invalida!
```

```
Digite a sua renda anual: 1000.00
Imposto: 150.0
```

```
Digite a sua renda anual: 21451
Imposto: 3117.78
```

```
Digite a sua renda anual: 52000
Imposto:  11774.0
```

5. Faça um programa que calcule as raízes reais de uma equação de primeiro ou segundo grau. Assuma que a equação estará no formato:

$$ax^2 + bx + c = 0$$

Seu programa deverá receber como entrada os valores dos coeficientes a, b e c, e imprimir as raízes reais (se a equação as tiver). Note que quaisquer algarismos podem ser digitados como entrada para a, b e c, e se $a = 0$, então seu programa deverá calcular uma raiz de equação de primeiro grau.

Exemplos:

```
Digite o valor de a: 0
Digite o valor de b: 0
Digite o valor de c: 0
Equacao invalida!
```

```
Digite o valor de a: 0
Digite o valor de b: 2
Digite o valor de c: -12
Raiz: 6.0
```

```
Digite o valor de a: 1
Digite o valor de b: 0
Digite o valor de c: 9
Esta equacao nao possui raizes reais
```

```
Digite o valor de a: 1
Digite o valor de b: 1
Digite o valor de c: -2
Raizes: -2.0 e 1.0
```

6. Faça um programa para calcular a média final de um aluno da matéria de Abstração I da Universidade da Bachatóvia. As provas dessa universidade são pontuadas de 0 a 10, podendo haver casas decimais nas notas. A média final deve ser calculada segundo as regras abaixo:

   (a) O programa deve receber inicialmente dois números representando as notas da Prova 1 (P1) e da Prova 2 (P2) do aluno. Se a média $M1 = \frac{P1+P2}{2}$ for maior ou igual que 7,0, o aluno estará aprovado direto. Se essa mesma média for menor que 3,0, o aluno estará reprovado direto. Nesses dois casos, esta média $M1$ será a média final do aluno.

   (b) Caso a média $M1$ do aluno fique entre 3,0 e 7,0, o aluno deve realizar uma Prova Final (PF). Apenas nesse caso, o programa deverá pedir também a nota da PF. A média final MF será então calculada segundo a expressão $MF = \frac{M1+PF}{2}$, onde $M1$ é a média calculada entre a P1 e a P2 no item anterior. Para este último caso, se MF for maior ou igual que 5,0, o aluno estará aprovado. Caso contrário, estará reprovado.

   Exemplos:

```
Digite a nota da P1: 8.5
Digite a nota da P2: 9.5
Situacao: Aprovado direto. Media final: 9.0
```

```
Digite a nota da P1: 5
Digite a nota da P2: 0.4
Situacao: Reprovado direto. Media final: 2.7
```

```
Digite a nota da P1: 6
Digite a nota da P2: 5
Digite a nota da PF: 6.5
Situacao: Aprovado. Media final: 6.0
```

```
Digite a nota da P1: 12
Nota invalida!
```

7. Escreva um programa que leia as coordenadas do centro de um círculo (em um plano cartesiano) juntamente com o seu raio, e então informe se um determinado ponto de teste lido está dentro do círculo, no centro do círculo, na circunferência (fronteira) ou fora do círculo. Assuma que não ocorrem erros de arredondamento nos cálculos e que o usuário sempre fornece valores válidos. Apenas para lembrar, a equação da circunferência é dada por:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

, onde (x_c, y_c) são as coordenadas do centro da circunferência e r é o raio. Lembre-se de que seu programa deve informar em qual das *quatro* categorias está o ponto de teste.

Exemplo:

```
Digite a coordenada x do centro do circulo: 10
Digite a coordenada y do centro do circulo: 5
Digite o raio do circulo: 4

Digite a coordenada x do ponto de teste: 9
Digite a coordenada y do ponto de teste: 6

O ponto de teste se encontra dentro do circulo
```

Capítulo 4

Repetições (laços)

As cláusulas para repetição são de extrema importância no contexto da programação. Elas são utilizadas em situações onde é necessário executar um conjunto de instruções repetidas vezes, podendo estas repetições estarem condicionadas a algum teste. Na linguagem Python, temos duas cláusulas para essa finalidade: `while` e `for`.

4.1 Laço *while*

4.1.1 Primeira forma geral

A primeira forma geral de uso da cláusula `while` é dada por:

```
1 while <teste>:
2 <tab> <instrução 1>
3 <tab> <instrução 2>
4 .
5 .
6 .
7 <tab> <instrução n>
8 <primeira instrução pós-while>
```

Código 4.1: primeira forma geral da cláusula *while*.

A cláusula `while`, cuja tradução é *"enquanto"*, nos permite construir laços de repetição para finalidades diversas em Python. Denominamos aqui `<teste>` como *teste de repetição*. A filosofia por trás da mesma é a seguinte: primeiramente, `<teste>` é executado. Se o resultado for verdadeiro, o bloco de `n` instruções subordinadas (aquelas precedidas por `<tab>`) será executado. Denominados esse bloco de instruções como *bloco de repetição*. Após a execução de todo esse bloco, o programa saltará para a linha de declaração de `while` (a primeira linha do Código 4.1) e realizará novamente `<teste>`. Se o resultado ainda for verdadeiro, o bloco será novamente executado, e o programa saltará novamente para a linha de declaração de `while`, para uma nova realização de `<teste>`. Isso ocorrerá sucessivamente até que a avaliação de `<teste>` resulte em falso. Nessa situação, o programa saltará para a primeira linha pós-while, isto é, para a primeira linha após o bloco de instruções subordinadas a `while`. Assim, através da cláusula `while`, um bloco de instruções será executado repetidas vezes enquanto a avaliação de `<teste>` for considerada verdadeiro. A

cada execução do bloco de repetição, damos o nome de *iteração*. Observe que, é possível que o bloco não seja executado uma vez sequer se a primeira avaliação de `<teste>` já resultar em falso.

Como primeiro exemplo, faremos um programa que imprime na tela os primeiros n números naturais:

```
1  #exemplo de uso de while
2  #programa que imprime na tela os primeiros n números naturais
3
4  n = input("Digite o valor de n: ")
5  n = int(n)
6
7  contador = 1
8  while contador <= n:
9      print(contador)
10     contador = contador + 1
11
12 print("Tenha um bom dia!")
```

Código 4.2: exemplo de uso de *while*: programa que imprime na tela os primeiros n números naturais.

Note que, no Código 4.2, precisamos repetir `n` vezes a tarefa de imprimir um número na tela, onde `n` será um número fornecido pelo usuário durante a execução do programa. Para cumprir essa tarefa, definimos uma variável auxiliar denominada `contador`, que tem o objetivo de controlar o número de repetições realizadas e fornecer os números a serem impressos. Assim, esta variável é inicializada com o valor `1` (linha 7). A seguir, a linha de declaração do `while` define como teste `contador <= n` (linha 8). Isto significa que este teste dará verdadeiro para qualquer número `n` maior que zero que o usuário venha a fornecer como entrada. A seguir, vem o bloco de repetição. Primeiramente, a linha 9 imprime o valor corrente da variável contador, ao passo que a linha 10 faz com o que valor na variável contador aumente de uma unidade. Desse modo, a cada execução do bloco, temos uma impressão do valor de `contador` e um incremento de uma unidade nessa variável. Assim, é fácil visualizar que, em algum momento, mais precisamente após `n` iterações, o valor de `contador` se tornará maior que o da variável `n`, o que fará com que o teste na linha 8 resulte em falso e a repetição se acabe. A partir daí, o programa saltará para a linha 12, imprimindo assim sua saudação de despedida. Note então que a repetição das linhas 9 e 10 se encerrará exatamente quando `contador` assumir o valor `n+1`, e que até isto acontecer, todos os números de 1 até `n` serão impressos na tela devido a repetição da impressão de `contador` na linha 9 e da operação de incremento dessa mesma variável na linha 10.

Um exemplo de execução do Código 4.2 poderia ser:

```
Digite o valor de n: 3
1
2
3
Tenha um bom dia!
```

No exemplo anterior, o usuário fornece 3 como valor de `n`. Isto significa que o programa deve imprimir na tela os primeiros 3 números naturais. Após ler o valor de `n`, o programa definirá `contador` com o valor 1 na linha 7. Na linha 8, será feito o teste `contador <= n`, que, inicialmente será `1 <= 3`, o que

resulta em verdadeiro. Assim, a linha 9 imprimirá o valor de `contador` (que atualmente é 1), ao passo que a linha 10 atribuirá à variável `contador`, seu próprio valor somado de 1. Assim, `contador` passará a valer 2. Em seguida, o programa tem de voltar a linha 8, onde novamente o teste `contador <= n` deve ser avaliado. Devido aos valores correntes de `contador` (2) e `n` (3), este teste será avaliado como `2 <= 3`, o que também resultará em verdadeiro. Desse modo, o programa passará novamente a execução do bloco de repetição. A linha 9 causará a impressão de `contador` (que atualmente vale 2), ao passo que a linha 10 fará seu valor aumentar em 1, o que significa que a mesma passara a valer 3. Novamente o programa saltará para linha 8 para uma nova avaliação do teste `contador <= n`, que será realizada como `3 <= 3` e resultará em verdadeiro. Assim, uma nova execução das linhas 9 e 10 será realizada, o que fará com o que o valor de `contador` seja impresso (que atualmente é 3) e que esta variável passe a valer 4. Ao retornar para avaliação do teste na linha 8, agora o mesmo será executado como `4 <= 3`, o que resultará em falso. Assim, o programa saltará para linha 12, imprimindo a mensagem "Tenha um bom dia!". Como esta é a última linha do programa, o mesmo encerrará sua execução.

Ao se trabalhar com laços `while`, é preciso tomar cuidado ao se definir o teste de parada e o bloco de repetição de modo a evitar laços que acabem se repetindo indefinidamente (laço infinito), como o exemplo a seguir:

```
1 a = 0
2 while a >= 0:
3     print(a)
4     a += 1
5 print("Tchau!")
```

Código 4.3: exemplo de laço infinito: programa que imprime na tela os números naturais indefinidamente.

Lembramos que a linha `a += 1` equivale a `a = a + 1`. No Código 4.3, o teste de repetição é definido como `a >= 0`. No entanto, uma vez que a variável `a` é inicializada com o valor 1 na linha 1, e, no bloco de repetição, ela sempre aumenta de uma unidade, temos que o teste de repetição nunca deixará de ser satisfeito, pois Python trabalha com precisão arbitrária de dígitos para números inteiros. Assim, o laço nas linhas 2-4 se repetirá indefinidamente até o usuário abortar a execução do programa, pois o mesmo não se encerrará de modo natural. Como consequência, temos que a linha 5 jamais será executada.

No próximo exemplo, faremos um programa que lê notas de todos os alunos de uma turma. Se uma nota for superior a 5.0, diremos que o respectivo aluno está aprovado. Caso contrário, declaramos que o mesmo está reprovado.

```
1  #programa que lê notas de todos os alunos
2  #de uma turma. Se uma nota for superior a 5.0, diremos que
3  #o respectivo aluno está aprovado. Caso contrário, diremos
4  #que o mesmo está reprovado.
5
6  numAlunos = input("Digite o numero de alunos: ")
7  numAlunos = int( numAlunos )
8
9  contador = 1
10 while contador <= numAlunos:
11
12     nota = input("Digite a nota do aluno " + str(contador) + ":
           " )
13     nota = float(nota)
14
15     if nota < 5.0:
16         print("Aluno ", contador, " reprovado!")
17     else:
18         print("Aluno ", contador, " aprovado!")
19
20     contador += 1
21
22 print("Tenha um bom dia!")
```

Código 4.4: programa que lê notas de alunos e imprime situação dos mesmos.

Note que iniciamos o Código 4.4 pedindo ao usuário que informe o número de alunos da turma. Precisamos dessa informação para controlar o número de repetições de nosso `while`, pois precisamos, para cada aluno, ler sua nota e informar se o mesmo está aprovado ou reprovado. Dentro do bloco de repetições de `while`, usamos a função `input` para ler uma nota (linha 12). Como esse bloco se repetirá `numAlunos` vezes, devido às linhas 9,10 e 20, a linha 12 lerá a nota de todos os alunos da turma. Observe que passamos como argumento à essa função a string `"Digite a nota do aluno " + str(contador) + ": "` . O operador `+`, com strings, realiza a tarefa de concatenação. Assim a operação `"jessica" + "linda"` resulta na string `"jessicalinda"`. Desse modo, o uso do valor corrente de contador na mensagem impressa pela função `input` (linha 12) fará com que, na primeira execução do bloco de repetição (iteração), seja impressa a mensagem "Digite a nota do aluno 1:". Na segunda execução, será impressa a mensagem "Digite a nota do aluno 2:", e assim sucessivamente. Após ler a nota e convertê-la para `float`, realizamos um teste na linha 15 para determinar se o aluno foi aprovado ou não. Observe então que usamos uma cláusula `if` com `else` dentro do bloco de repetições de `while`. Por consequência, as instruções subordinados à essas cláusulas serão precedidas por dois caracteres `<tab>` de modo que fiquem mais a direita de sua cláusula subordinadora.

Um exemplo de execução do Código 4.4 seria:

```
Digite o numero de alunos: 3
Digite a nota do aluno 1: 7.5
Aluno  1  aprovado!
Digite a nota do aluno 2: 4.3
Aluno  2  reprovado!
Digite a nota do aluno 3: 9.1
Aluno  3  aprovado!
Tenha um bom dia!
```

Note que, embora o código 4.4 leia as notas de todos os alunos, o mesmo não as armazena de modo permanente, pois, a cada iteração, a nova nota lida sobrescreve a nota lida anteriormente, de modo que a variável `nota` armazena apenas a última nota lida pelo programa. Assim, não seria possível usar as notas lidas posteriormente para, por exemplo, calcular a mediana das mesmas. Aprenderemos, todavia, ao longo do curso a utilizar objetos sequenciais que nos permitirão realizar esse armazenamento de modo adequado, e, portanto, habilitar o uso posterior desses dados pelo programa.

Como de praxe, você está convidado a acompanhar a execução o Código 4.4. Se ficar com alguma dúvida em relação a alguma das operações, é sempre uma boa ideia colocar *prints* adicionais no código para conferir os valores que as variáveis estão assumindo a cada iteração do laço.

4.1.2 Forma mais geral

A forma mais geral de um laço `while` inclui a presença opcional de uma cláusula `else`:

```
1 while <teste>:
2 <tab> <bloco de instruções 1>
3 else:
4 <tab> <bloco de instruções 2>
5 <primeira instrução pós-while>
```

Código 4.5: forma mais geral da cláusula *while*, com cláusula *else*.

O bloco de instruções associado à cláusula `else` só será executado, no máximo, uma única vez quando `<teste>` resultar em falso. Se o programa entrar em laço infinito ou o laço for interrompido por uma cláusula `break`, o bloco de instruções associado à cláusula `else` não será executado. Ressaltamos que o uso de `else` com laços é opcional e deve ser evitado por principiantes em programação.

4.2 Laço *for*

A cláusula `for`, cuja tradução é *"para"*, funciona como iterador genérico de sequências em Python. Desse, modo, seu uso está restrito apenas à tarefa de percorrer os itens de objetos iteráveis. Sua forma geral é dada por:

```
1 for <destino> in <objeto iterável>
2 <tab> <bloco de instruções 1>
3 else:
4 <tab> <bloco de instruções 2>
5 <primeira instrução pós-for>
```

Código 4.6: forma geral da cláusula *for*.

O laço `for` funciona da seguinte forma: `<objeto iterável>` é um objeto iterável, isto é, é um objeto composto por diversos outros objetos e que pode ser percorrido. A variável `<destino>` apontará para cada um dos objetos contidos em `<objeto iterável>`, um de cada vez, na ordem definida pelo mesmo, isto é, `<destino>` percorrerá todos os valores em `<objeto iterável>`. Assim, o bloco subordinado ao `for` (`<bloco de instruções 1>`) será executado uma vez para cada valor em `<objeto iterável>` assumido por `<destino>`. Mais uma vez, o

que define esse bloco de repetição é a presença de um caracter `<tab>` a frente das instruções.

Note que o laço `for`, assim como `while`, também admite uma cláusula opcional `else`, que é executada após `<objeto iterável>` ser percorrido por completo e sem a execução de uma cláusula `break`. Reiteramos que principiantes devem evitar o uso da cláusula `else` em laços.

O Código 4.7 traz um primeiro exemplo com uso de `for`:

```
1 #primeiro exemplo de for
2
3 for x in (1, 7, -2, 4.8):
4     print("Numero: ", x)
5
6 print("Tenha um bom dia!")
```

Código 4.7: primeiro exemplo de uso de *for*.

No Código 4.7, a tupla `(1, 7, -2, 4.8)` é o nosso objeto iterável que será percorrido. A variável `x` faz o papel de `<destino>`, isto é, `x` é a variável que percorrerá os itens do objeto iterável. O bloco de repetição é composto apenas da linha 4, que imprime a mensagem "Numero:" seguido do valor corrente de `x`. Por fim, após a execução do laço, será impressa a mensagem "Tenha um bom dia!". A saída do Código 4.7 é dada por:

```
Numero:  1
Numero:  7
Numero:  -2
Numero:  4.8
Tenha um bom dia!
```

Observe que `x` assumiu, a cada iteração do laço `for`, cada um dos valores do objeto sendo iterado, ao passo que a linha 4 foi executada para `x` assumindo cada um desses valores. Podemos ler a linha 3 como *"para todo x no conjunto {1, 7, -2, -4.8 }"*.

Antes de passar aos exemplos de laço `for` é oportuno apresentar a função `range`:

4.2.1 Função *range*

A forma geral de uso da função `range` é dada por:

```
1 range( <inicio>, <fim>, <passo> )
```

A função `range` gera, no Python 3, um objeto `range` que representa um intervalo, onde os elementos são oriundos a progressão aritmética iniciada em `<inicio>`, finalizada em `<fim>` com razão `<passo>`. É preciso ressaltar desde já que `<fim>` não é incluído no intervalo. No Python 2, essa mesma funcionalidade é desempenhada pela função `xrange`, pois `range` no Python 2 é uma função que gera uma lista com as mesmas características descritas.

Para visualizar melhor os intervalos gerados, faremos a conversão dos mesmos para tupla nos exemplos a seguir:

Gerando um intervalo de 1 até 5 (sem incluir o 5) com passo 1:

```
>>> a  = tuple( range(1,5,1) )  #gera intervalo de 1 até 5 (sem
    inclui-lo), com passo (razão) 1
>>> a
(1, 2, 3, 4)
```

Vamos gerar agora um intervalo de 0 até 10 com passo 2.

```
>>> b = tuple( range(0,10,2) )  #gera intervalo de 0 até 10 (
    sem inclui-lo) com passo (razão) 2
>>> b
(0, 2, 4, 6, 8)
```

Podemos gerar intervalos decrescentes, por exemplo de -6 até -19 com passo -3:

```
>>> z = tuple( range(-6,-19,-3) )
>>> z
(-6, -9, -12, -15, -18)
```

Se `<passo>` for omitido, o valor 1 é assumido:

```
>>> c = tuple( range(6,9) )
>>> c
(6, 7, 8)
```

Se apenas um argumento for fornecido, assume-se que o início é zero, o argumento fornecido é o fim e o passo é 1.

```
>>> d = tuple( range(4) )
>>> d
(0, 1, 2, 3)
```

Vamos então reescrever o Código 4.2 que imprime na tela os primeiros n números naturais. Para isso, vamos utilizar um laço `for` e a função `range`:

```
1  #exemplo de uso de for
2  #programa que imprime na tela os primeiros n números naturais
3
4  n = input("Digite o valor de n: ")
5  n = int(n)
6
7  for contador in range(1, n+1, 1):
8      print(contador)
9
10 print("Tenha um bom dia!")
```

Código 4.8: exemplo de uso de *for*: programa que imprime na tela os primeiros n números naturais.

Observe que, na linha 7, utilizamos como objeto a ser iterável, o intervalo retornado pela função `range`. Observe que, como o segundo argumento (`<fim>`) não é incluído no intervalo, foi preciso utilizar o valor `n+1` para que o intervalo gerado fosse de 1 até `n`. Note também o uso do passo 1. Assim, a variável `contador`, declarada na própria linha 7, percorrerá todos os números naturais do intervalo [1, `n`]. Desse modo, tudo o que precisamos fazer é imprimir, no bloco de repetição, o valor corrente de `contador`. Note que não preciso utilizar a linha `contador = contador + 1`, pois o incremento do valor de `contador` acaba sendo feito de modo implícito pela uso combinado do laço `for` com a função `range`. Desta maneira, fica claro desde já que, trabalhando com laços `for`, reduzimos o risco de acidentalmente implementarmos um laço infinito (por exemplo, se esquecermos a linha 10 do Código 4.2, teremos um laço infinito).

No entanto, nem todo laço de repetição feito com `while` pode também ser feito com `for`, pois este último se destina apenas a percorrer objetos iteráveis. Por sua vez, todo laço construído com `for` pode ser reimplementado com `while`.

Um exemplo de execução do Código 4.8 poderia ser (idêntico ao do Código 4.2):

```
Digite o valor de n: 3
1
2
3
Tenha um bom dia!
```

Como exercício, você está convidado a reescrever o Código 4.4 usando `for` no lugar de `while`.

4.3 Cláusulas *break* e *continue*

As cláusulas `break` e `continue` são utilizadas no contexto de laços de repetição. `break` força a saída (interrupção) do laço mais próximo que a envolve, sem a execução das instruções associadas a uma possível cláusula `else` desse laço.

```
1  #exemplo de uso de break
2
3  a = 0
4  while 1 == 1:
5
6      print(a)
7      if a > 3:
8          break
9
10     a += 1
11 else:
12     print("Resultado do teste deu falso")
13
14 print("Tenha um bom dia")
```

Código 4.9: exemplo de uso de *break*.

O Código 4.9 se inicia declarando uma variável `a` com o valor 0 (linha 3). A seguir, utiliza-se um laço `while` com o teste `1 == 1`, cuja resposta será sempre verdadeiro, pois 1 sempre será igual a 1. Por esta razão, a depender do teste de repetição, esse laço está destinado a se repetir indefinidamente. No entanto, a presença da cláusula `break` na linha 8 interromperá a repetição do laço quando o teste na linha 7 resultar em verdadeiro, o que ocorrerá quando `a` se tornar maior que 3. Observe que a linha 10 incrementa o valor de `a` em uma unidade a cada execução do bloco de repetição. Assim, esse código imprimirá na tela:

```
0
1
2
3
4
Tenha um bom dia
```

Note que a execução da cláusula `break` na linha 8 fez com o bloco de instruções associado ao `else` (linha 12) não fosse executado, e o programa saltasse

diretamente para a linha 14, imprimindo assim sua saudação de despedida. Por fim, apontamos que a declaração `while 1 == 1:` poderia ser substituída por `while True:`, já que `1 == 1` sempre resulta em `True`.

A cláusula `continue`, por sua vez, pula para o início do laço mais próximo que o envolve (para a linha de declaração de `while` ou `for`). Na prática, `continue` se destina a fazer o laço avançar imediatamente para a próxima iteração, mesmo que o bloco de repetição não tenha sido totalmente executado na iteração corrente.

O Código 4.10 ilustra um exemplo de uso de `continue`. Na linha 3, é declarado um laço `for` no qual a variável `k` percorrerá os itens da tupla `(1, 3, 5, 7)`. Observe que, a cada iteração, a linha 8 imprime o valor corrente de `k`. No entanto, o `if` na linha 5 fará com que, quando `k` assuma o valor 5, a cláusula continue (linha 6) seja executada, o que fará com que o programa salte imediatamente para a próxima iteração do `for` e não execute a linha 8 para este valor de `k`. Assim, o valor 5 não será impresso na tela.

```
1  #exemplo de uso de continue
2
3  for k in (1, 3, 5, 7):
4
5      if k == 5:
6          continue
7
8      print(k)
9
10 print("Tenha um bom dia!")
```

Código 4.10: exemplo de uso de *continue*.

A execução do Código 4.10 terá como resultado na tela:

```
1
3
7
Tenha um bom dia!
```

4.4 Exemplos

4.4.1 Resultado Acumulativo

É muito comum o uso de laços para o cálculo de resultados acumulativos. Como exemplo, faremos um programa que calcula o somatório de termos fornecidos pelo usuário.

```
1  #programa que calcula o somatório de termos fornecidos pelo
       usuário:
2
3  numTermos = int( input("Entre com o numero de termos: ") )
4
5  soma = 0
6  for k in range(0, numTermos):
7
8      termo = float( input("Entre com o termo " + str(k+1) + ": ")
           )
9      soma += termo
10
11 print("Somatorio dos termos: ", soma)
```

Código 4.11: somatório de termos do usuário.

O Código 4.11 se inicia com a leitura do número de termos do somatório na variável `numTermos` (linha 3). Observe que aqui, realizamos a leitura com `input` e a conversão para `int` em uma única linha. A seguir, na linha 5 inicializamos uma variável chamada `soma`, cuja finalidade é armazenar o valor do somatório dos termos lidos até então, com o valor 0. Na linha 6, declaramos um laço `for` de modo que a variável `k` percorrerá os números inteiros no intervalo de 0 até `numTermos - 1`, o que fará com o bloco de repetição seja executado `numTermos` vezes. Uma alternativa seria fazer essa mesma variável percorrer o intervalo de 1 até `numTermos`. Todavia, preferimos iniciar a contagem a partir do 0, e não do 1, devido ao fato de que Python inicia a contagem de índices de objetos sequenciais a partir do 0. Assim, de modo a já nos habituarmos com essa forma de contagem dos índices, começamos desde já a contar nossos intervalos a partir do 0 também. A seguir, já no bloco de repetição, a linha 8 solicita ao usuário o valor de um termo e o converte para `float`. Note que usamos o valor de `k+1` para compor o argumento da função `input`, conforme fizemos no Código 4.4. Aqui, somamos 1 ao valor de `k`, pois `k` começa a percorrer o intervalo a partir do 0, e não do 1. Assim, na primeira iteração, a função `input` imprimirá na tela a mensagem "Entre com o termo 1:". Na segunda iteração, será impressa a mensagem "Entre com o termo 2", e assim sucessivamente. A linha 9 é responsável por pegar o termo lido, somar com o valor corrente de `soma`, e armazenar na própria variável `soma`. Como esta variável é inicializada com 0 na linha 5, após a primeira iteração, `soma` estará exatamente com o valor do primeiro termo lido (o valor anterior 0 mais o valor do primeiro termo). Após a segunda iteração, soma estará com seu valor anterior (o valor do primeiro termo) mais o valor do segundo termo, que acabou de ser lido. Após a leitura do terceiro termo, `soma` estará com o valor anterior (a soma do primeiro termo com o segundo) mais o valor do terceiro termo que foi lido prontamente. Dessa forma a variável `soma` acumulará o somatório de todos os termos lidos até então. Por fim, a linha 11 será executada após o laço `for` e imprimirá o somatório de todos os termos lidos calculado na variável `soma`.

Um exemplo de execução do Código 4.11 é dado a seguir:

```
Entre com o numero de termos: 3
Entre com o termo 1: 6
Entre com o termo 2: 10
Entre com o termo 3: 2
Somatorio dos termos:  18.0
```

Lembre-se de que, caso você tenha ficado em dúvida quanto ao funcionamento, você sempre pode rodar o código com `prints` adicionais para acompanhar os valores que as variáveis estão assumindo. Neste exemplo, para um melhor acompanhamento do programa, seria uma boa ideia passar a linha 11 para dentro do laço `for` colocando um caracter `<tab>` a sua frente. Dessa forma, seria possível a evolução dos valores da variável `soma` ao longo das iterações do laço.

4.4.2 Resultado acumulativo com teste: maior termo lido

Nesta subseção, faremos diversos exemplos de código para um programa que deve ler termos numéricos do usuário e apontar qual o maior termo lido. A filosofia por trás de todos os exemplos é manter uma variável denominada `maior` para armazenar o maior termo lido até então.

Primeiro modo

Neste primeiro exemplo (Código 4.12), lemos o número de termos na linha 3 e criamos a variável `maior` antes do laço de repetição lendo o primeiro termo separadamente para inicializar seu valor (linha 5). O próximo passo é a construção do laço de repetição `for`. Uma vez que o primeiro termo já foi lido antes do laço, esta repetição iterará com a varável `i` indo de 2 até `numTermos` (linha 7). Já no bloco de repetição, a linha 8, lê um termo do teclado e o converte para `float`. Note, novamente, o uso da variável sendo iterada, `i`, conforme o Código 4.11. A seguir, temos um `if` na linha 9 para testar se o termo recém lido na variável `termo` é maior que o termo armazenado na variável `maior`. Se sim, `maior` é atualizada com o valor de `termo` (linha 10). Dessa forma, a variável `maior` sempre armazenará o maior termo lido do usuário até então. Por fim, a linha 12 imprime o maior termo lido.

```
1  #programa que termos e informa o maior termo lido:
2
3  numTermos = int( input("Entre com o numero de termos: ") )
4
5  maior = float( input("Entre com o termo 1: ") )
6
7  for i in range(2, numTermos+1):
8      termo = float( input("Entre com o termo " + str(i) + ": ") )
9      if termo > maior:
10         maior = termo
11
12 print("Maior termo: ", maior)
```

Código 4.12: maior termo lido do usuário.

Um exemplo de execução do Código 4.11 é dado a seguir:

```
Entre com o numero de termos: 4
Entre com o termo 1: 9
Entre com o termo 2: -7
Entre com o termo 3: 100
Entre com o termo 4: 3
Maior termo:  100.0
```

Mais uma vez, caso você tenha ficado em dúvida quanto ao funcionamento, você pode rodar o código com `prints` adicionais para acompanhar os valores que as variáveis estão assumindo. Neste exemplo, para um melhor acompanhamento do programa, seria uma boa ideia passar a linha 12 para dentro do laço `for` colocando um caracter `<tab>` a sua frente. Dessa forma, seria possível a evolução dos valores da variável `maior` ao longo das iterações do laço.

Segundo modo

Embora o Código 4.12 funcione adequadamente, algumas pessoas podem se sentir incomodadas pelo fato de ter sido preciso ler o primeiro termo separadamente dos demais antes do laço de repetição. O Código 4.13 apresenta uma nova solução para este mesmo problema onde todos os termos são lidos dentro do laço de repetição. Neste exemplo, após a leitura de cada termo, a variável maior é atualizada de acordo com a seguinte estratégia:

- Se o termo lido foi o primeiro, `maior` deve ser declarando usando seu valor;
- Caso contrário, `maior` deve ser atualizado se o termo lido é maior que seu valor.

```
1  #programa que termos e informa o maior termo lido:
2
3  numTermos = int( input("Entre com o numero de termos: ") )
4
5  for i in range(1, numTermos+1):
6      termo = float( input("Entre com o termo " + str(i) + ": ") )
7
8      if i == 1:
9          maior = termo
10     else:
11         if termo > maior:
12             maior = termo
13
14 print("Maior termo: ", maior)
```

Código 4.13: maior termo lido do usuário.

Como todos os termos são lidos no laço de repetição, agora o `for` na linha 5 faz `i` iterar de 1 até `numTermos`. Dentro do bloco de repetição, após a leitura do termo corrente e sua conversão para `float` na variável `termo`, é implementada a estratégia de atualização da variável `maior` explicitada anteriormente. O teste `i == 1` visa a verificar se o termo recém lido é o primeiro (linha 8). Em caso positivo, significa que esta é a primeira iteração do laço e, portanto, nesse caso, a variável `maior` ainda não foi declarada, ou seja, não existe. Assim, a variável `maior` é criada e inicializada com o valor de `termo` (linha 9). Se o teste `i == 1` resulatr em falso, significa que esta já não é a primeira iteração do laço e, portanto, a variável maior já existe e está armazenando o valor de algum termo lido anteriormente. Assim, só é preciso testar se o termo recém lido é maior do que o termo armazenado na variável `maior` (linha 11). Em caso positivo, a variável `maior` é atualizada na linha 12. Por fim, o programa imprime o maior número lido após a execução do laço `for` na linha 14.

Terceiro modo

Algumas pessoas mais exigentes podem ainda estar insatisfeitas com o Código 4.13, pois embora seu laço de repetição incorpore a leitura de todos os termos, o mesmo se tornou mais complexo em comparação ao Código 4.12. Isso se dá porque, ao contrário do Código 4.12, o Código 4.13 não inicializa a variável `maior` antes de seu laço. Para construir um código mais elegante que os dois anteriores, seria preciso ler todas as variáveis dentro do laço de repetição, ao mesmo tempo em que se declara a variável `maior` antes do mesmo. Uma boa ideia é inicializar a variável `maior` com algum valor que fosse garantidamente menor ou igual a qualquer número que o usuário possa vir a fornecer. Essa é uma boa situação para se utilizar o número $-\infty$ ("menos infinito"), pois este número seria garantidamente menor ou igual a qualquer outro. Uma forma de fazer uma variável receber os valores $-\infty$ e $+\infty$ é:

```
u = float("-Infinity")
v = float("Infinity")
```

Note que os valores $-\infty$ e $+\infty$ pertencem a classe `float` e podem ser utilizados como qualquer outro número `float`. Todavia, operações envolvendo esses números podem acabar resultando em infinito ou em um resultado indeterminado, representado como o valor `float NaN` (*Not a Number* ou "Não é Número"):

```
>>> a = float("Infinity")
>>> a
inf
>>> a + 10      #infinito somado a um número finito resultará em infinito
inf
>>> 2*a         #infinito vezes um número positivo resultará em infinito
inf
>>> -3*a        #infinito vezes um número positivo resultará em -infinito
-inf
>>> a/a         #infinito dividido por infinito tem valor indeterminado (nan)
nan
```

Uma forma de fazer uma variável receber o valor `NaN` é:

```
w = float("NaN")
```

Voltando ao nosso exemplo, onde usamos o valor $-\infty$:

```
#programa que termos e informa o maior termo lido:

numTermos = int( input("Entre com o numero de termos: ") )

maior = float("-Infinity")

for i in range(1, numTermos+1):
    termo = float( input("Entre com o termo " + str(i) + ": ") )

    if termo > maior:
        maior = termo

print("Maior termo: ", maior)
```

Código 4.14: maior termo lido do usuário.

No Código 4.14, a variável `maior` é inicializada com o valor $-\infty$ na linha 5. Isso fará com que qualquer termo lido do teclado que não seja $-\infty$ nem `NaN` seja considerado maior que o valor de `maior`. Assim, ao ler o primeiro termo dentro do laço `for` na linha 8, ele passará no teste da linha 9 e atualizará o valor de `maior` na linha 10 (desde que não seja $-\infty$ nem `NaN`, é claro). A partir da segunda iteração do laço `for`, o programa passará a funcionar como o Código 4.12.

4.4.3 Variável sinalizadora: O exemplo de detecção de número primo

Um exemplo clássico no ensino de programação é um programa para responder se um determinado número é primo. Um número primo é um número natural que só é divisível de forma exata por 1 e por ele mesmo. Para determinar se um número natural `n` é primo, nossa primeira ideia é contar seus divisores inteiros no intervalo [2 `n-1`]. Para saber se `n` é divisível pro algum número `k`, usaremos a operação `n % k`, que calcula o resto da divisão de `n` por `k`. Se este resto for 0, temos que `k` é divisor de `n`. Esta ideia está implementada no Código 4.15.

```
1  #programa para determinar se um número é primo
2
3  n = int( input("Entre com o numero: ") )
4
5  numDivs = 0
6
7  for k in range(2, n):
8      if n % k == 0:
9          numDivs += 1
10
11 if numDivs == 0:
12     print("Este numero e primo!")
13 else:
14     print("Este numero nao e primo")
```

Código 4.15: determina se um numero é primo.

O Código 4.15 inicia lendo o número `n` que deve ser verificado quando a "primalidade" e convertendo-o para `int` (linha 3). A seguir, inicializamos na linha 5 a variável `numDivs`, cuja a finalidade é contar o número de divisores de `n` no intervalo [2 `n-1`]. O próximo passo é a declaração do laço `for` na linha 7 que fará a variável `k` iterar de 2 até `n-1`. Para cada um desses valores de `k`, testamos se o mesmo divide `n` (linha 8) avaliando se o resto da divisão de `n` por `k` é igual a 0. Em caso positivo, incrementamos o contador de divisores `numDivs` na linha 9. Após o laço `for`, testamos se o contador de divisores `numDivs` é iguala a zero (linha 11). Em caso positivo, temos que não encontramos qualquer divisor no intervalo [2 `n-1`], e, portanto, afirmamos que o número é primo (linha 12). Caso contrário, `numDivs` será maior que 0, significando que o número tem divisores no referido intervalo e, por consequência, dizemos que o mesmo não é primo na linha 14.

Note que a variável `numDivs` tem a função de sinalizar, ao final da execução do laço `for`, se o número `n` é primo ou não. Assim como nesse exemplo, em muitos contextos variáveis são utilizadas para sinalizar algum tipo de estado.

Exemplos de execução do Código 4.15:

```
Entre com um numero: 23
Este numero e primo!
```

```
Entre com um numero: 15
Este numero nao e primo
```

É válido mencionar que o Código 4.15 pode ter sua eficiência melhorada por meio de diversas estratégias. Uma delas é a incorporação uma cláusula `break` dentro do `if` na linha 10, pois, a partir do momento em que encontramos o primeiro divisor para `n`, já sabemos que o mesmo não é primo e, assim, podemos interromper o laço `for`.

4.5 Exercícios

1. Escreva um programa que receba um número do teclado e informe sua raiz quadrada real. Note que seu programa não deve aceitar números negativos como entrada, de modo que, se o usuário fornecer algum número menor que zero, seu programa deve solicitar o número novamente até o usuário fornecer uma entrada não-negativa.

   Exemplos:

   ```
   Entre com um numero: 25
   Raiz quadrada: 5.0
   ```

   ```
   Entre com um numero: -9
   Entrada invalida!
   Entre com um numero: -7
   Entrada invalida!
   Entre com um numero: 4
   Raiz quadrada: 2.0
   ```

2. Escreva um programa que leia um número positivo do teclado e informe se ele é par ou ímpar (assuma que o usuário sempre entrará com números inteiros). Seu programa deve tratar o caso em que o número lido é não positivo, informando uma mensagem de erro e solicitando o número novamente até que ele seja válido. Ao final da execução, seu programa deve perguntar ao usuário se ele deseja executar o programa novamente. Se o usuário entrar com o número zero, ele estará dizendo que não, e se entrar com qualquer outro número, estará dizendo que sim. Neste último caso, seu programa deve solicitar novamente a entrada e executar até o usuário não querer mais. Veja o exemplo:

```
Entre com um número: 10

10 e numero par.

Deseja executar o programa novamente? (0 - Nao) (1 - Sim): 1

Entre com um número: -8
Entrada inválida!
Entre com um número: -5
Entrada inválida!
Entre com um numero: 23

23 e numero impar

Deseja executar o programa novamente? (0 - Nao) (1 - Sim): 0

Tenha um bom dia!
```

3. Escreva um programa que leia um conjunto de números (termos) do teclado e imprima o produto de todos esses números. Antes de começar a ler os números, o programa deve solicitar o total de termos que o usuário pretende entrar. Não se esqueça de que um produtório de 0 termos deve resultar em **zero**.

   Exemplos:

```
Entre com a quantidade de termos do produtorio: 2
Entre com o termo 1: 3
Entre com o termo 2: -6

Produto dos termos: -18
```

```
Entre com a quantidade de termos do produtorio: 3
Entre com o termo 1: 7
Entre com o termo 2: 0.2
Entre com o termo 3: 10

Produto dos termos: 14
```

4. Faça um programa que calcule o fatorial de um número inteiro lido do teclado.

   Exemplos:

```
Entre com o numero: 5
Fatorial de 5: 120
```

```
Entre com o numero: 0
Fatorial de 0: 1
```

5. Faça um programa que leia um conjunto de números positivos do teclado e informe se algum numero do conjunto é múltiplo de 10. Assuma que o usuário não sabe com quantos números deseja entrar, de modo que seu programa deve ler números indefinidamente até o usuário entrar com o

primeiro número negativo, marcando assim o final da entrada. Note que esse último número negativo não faz parte do conjunto de entrada, e só tem a finalidade de indicar quando os dados acabam. Obs: você deve ler **todos** os números que o usuário digitar até o mesmo entrar com o primeiro valor negativo!

Exemplos:

```
Entre com o numero 1 do conjunto: 25
Entre com o numero 2 do conjunto: 10
Entre com o numero 3 do conjunto: -1

Existe multiplo de 10 neste conjunto
```

```
Entre com o numero 1 do conjunto: 56
Entre com o numero 2 do conjunto: 14
Entre com o numero 3 do conjunto: 191
Entre com o numero 4 do conjunto: -7

Nao existe multiplo de 10 neste conjunto
```

6. Escreva um programa que leia um vetor de n coordenadas e informe se o vetor se encontra no primeiro ortante (ortante positivo). Nota: um vetor se encontra no ortante positivo se **todas** as suas coordenadas são números positivos.

   Exemplos:

```
Entre com o numero de coordenadas: 4

Entre com a coordenada 1: 6
Entre com a coordenada 2: 5.3
Entre com a coordenada 3: -7
Entre com a coordenada 4: 12.34

Este vetor nao se encontra no primeiro ortante.
```

```
Entre com o numero de coordenadas: 5

Entre com a coordenada 1: 8
Entre com a coordenada 2: 99.99
Entre com a coordenada 3: 2.008
Entre com a coordenada 4: 1
Entre com a coordenada 5: 37.8

Este vetor se encontra no primeiro ortante.
```

7. Em uma conceituada universidade, o sistema de ingresso prevê provas de X disciplinas escolhidas de acordo com a carreira desejada. Para conseguir passar por uma entrevista e pleitear uma das vagas da universidade, cada candidato precisa obter grau igual ou superior que 5.0 em todas as provas e apresentar média final maior ou igual que 7.0 considerando todas as notas.

Escreva um programa em Python que leia as X notas de um candidato e informe sua média e se ele está apto a prosseguir na disputa pelas vagas. **Atenção:** Todas as X notas do candidato devem sempre ser lidas. Assuma que todas as entradas sempre serão válidas.

Exemplo:

```
Entre com o numero de provas: 4
Entre com a nota da prova 1: 8
Entre com a nota da prova 2: 4.5
Entre com a nota da prova 3: 9
Entre com a nota da prova 3: 7.2

Media das notas:  7.175
Este candidato nao esta apto a prosseguir
```

```
Entre com o numero de provas: 2
Entre com a nota da prova 1: 7.0
Entre com a nota da prova 2: 8.5

Media das notas:  7.75
Este candidato esta apto a prosseguir
```

```
Entre com o numero de provas: 3
Entre com a nota da prova 1: 5.0
Entre com a nota da prova 2: 6.5
Entre com a nota da prova 3: 7.5

Media das notas:  6.333333333333333
Este candidato nao esta apto a prosseguir
```

Capítulo 5

Strings

Strings são objetos que se destinam a representação e manipulação de textos em geral. Dessa forma, strings são sequências de caracteres sob uma ordem específica. Em Python, strings são objetos imutáveis da classe denominada `str`, e podem ser declaradas por meio de aspas:

```
1 >>> nome = "jessica"    #variável nome recebe um objeto string (
      str) representando o texto ``jessica''
2 >>> nome
3 "jessica"
```

Note a diferença entre as operações

```
1 nome = "jessica"
```

Código 5.1: atribuição de objeto string.

e

```
1 nome = jessica
```

Código 5.2: atribuição de objeto em outra variável.

Na primeira operação (Código 5.1), o uso das aspas faz com que a variável `nome` receba um objeto string com o texto "jessica". Note que a ausência das aspas na segunda operação (Código 5.2) faz com que a variável `nome` receba o mesmo conteúdo de uma outra variável denominada `jessica`. Assim, o Código 5.2 resultará em erro se não houver variável de nome `jessica` em seu contexto de execução.

Strings podem ser declaradas de modo equivalente usando aspas simples ou duplas. Assim, `"jessica"` equivale a `'jessica'`. Há ainda as strings de documentação (*docstrings*) que são strings declaradas usando aspas triplas. Este último tipo de string tem a finalidade de representar documentação sobre o código, por exemplo explicando sua finalidade e funcionamento. Por exemplo, podemos reescrever o Código 2.1 usando docstrings para fornecer informações sobre o mesmo:

```
1 """primeiro programa em Python
2 Este programa tem a finalidade de ilustrar a leitura de dados do
3 usuário e a impressão de informações na tela usando as
4 funções print e input.
5 autor: Wendel Melo """
6
7 nome = input("Digite o seu nome: ")
8 print("Ola ", nome, "!")
```

Código 5.3: uso de strings de documentação (docstrings).

Note que, no Código 5.3, usamos doscstrings para descrever sua finalidade nas linhas 1-5. Essas docstrings são usadas por ferramentas especiais para gerar documentação sobre códigos-fonte, o que é especialmente útil no desenvolvimento de módulos Python, conforme veremos no Capítulo 8. Em muitos casos, acabam sendo utilizadas como comentários de múltiplas linhas em Python.

Em Python, não há definição do caractere de fim de string *\0* conforme ocorre na linguagem C, assim como também não há um tipo para representação de um caracter isolado. No entanto, é sempre possível trabalhar com strings de um único caracter. Símbolos como *!*, -, *%*, *(*, *)*, *#* e até mesmo o espaço em branco ( ) são considerados caracteres, embora não sejam alfabéticos. Desse modo, a string `"oi, Ana!"` possui 8 caracteres, pois os sinais de pontuação e espaço também contam como caracteres. Nesse caso específico, temos 8 caracters distintos, pois as letras maiúsculas são consideradas como caracteres diferentes das minúsculas, conforme a seguir:

```
>>> "A" == "a"    #resulta em False, pois letras maiúsculas são
    diferenciadas das minúsculas
False
>>> "A" == "A"
True
```

Pode-se definir strings com caracteres numéricos:

```
>>> tex = "7"
>>> g = "120"
```

No entanto, pelo fato de serem strings, **estes objetos são tratados como texto e não como números**. Assim, não é possível realizar operações aritméticas diretamente com o conteúdo das variáveis `tex` e `g`:

```
>>> g + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Do mesmo modo, a comparação de igualdade entre uma string e um número resultará em `False`, pois os mesmos são considerados objetos representando coisas distintas.

```
>>> tex == 7
False
>>> tex == "7"
True
```

Todavia, se a string representar um número válido, é possível realizar a conversão para tipos numéricos. Destas maneira, operações aritméticas podem ser realizadas:

```
>>> int(tex) == 7
True
>>> float(g) + 5
125.0
```

5.1 Sequências de escape (constantes de barra invertida) e literais de string

As sequências de escape, também denominadas como constantes de barra invertida , tem o propósito de definir caracteres (bytes) especiais dentro de uma string. A mais comum é `\n` que é utilizada para encerrar a linha atual e ir para a próxima linha (pula linha). Sempre que necessário definir uma string que deva conter os caracteres aspas simples ou duplas, pode-se utilizar `\'` e `\"`, já que estes caracteres são utilizados para definir fim de strings. A Tabela 5.1 lista as possíveis sequências de escape e seus respectivos significados.

sequência	significado
`\a`	som de *bip* no auto falante
`\b`	*backspace*
`\f`	formatação em cascata
`\n`	pula linha (ENTER)
`\r`	*carriage return*
`\t`	tabulação horizontal
`\v`	tabulação vertical
`\ooo`	caracter com valor octal ooo
`\xhh`	caracter com valor hexadecimal hh
`\'`	aspas simples
`\"`	aspas duplas
`\\`	barra invertida

Tabela 5.1: sequências de escape para strings.

Exemplo:

```
>>> v = "Meu texto \n\n\t pulei duas linhas e tabulei!"
>>> print(v)
Meu texto

        pulei duas linhas e tabulei!
```

A letra `r` (maiúscula ou minúscula) antes de uma string indica que a mesma é uma string bruta (*raw string*), o que significa que possíveis sequências de escape presentes na mesma não serão interpretadas, conforme o Código 5.4:

```
>>> texto2 = r"\n nao pulou linha"
>>> print(texto2)
\n nao pulou linha
```

Código 5.4: exemplo de string bruta (*raw string*).

Por padrão, internamente Python representa strings na codificação ASCII, que estabelece um código único de 0 até 255 para a representação de cada ca-

racter. Em outras palavras, a codificação ASCII só conseguiria representar um conjunto de até 256 caracteres diferentes, o que ´e mais do que suficiente para representar as 26 letras do nosso alfabeto, maiúsculas e minúsculas, caracteres acentuados e numéricos, sinais de pontuação e demais símbolos de nossa escrita corrente[1]. Todavia, alguns idiomas, como russo, árabe e japonês, utilizam um conjunto diverso de caracteres, os quais não podem ser representados pela codificação ASCII. Para lidar com esses conjuntos diversos de símbolos foi criada a codificação Unicode, de modo a permitir a representação de textos de qualquer sistema de escrita usado atualmente.

Python permite a declaração de strings na codificação Unicode através da introdução da letra `u` (maiúscula ou minúscula) antes da mesma. As strings na codificação Unicode são tratadas de modo transparente pelo interpretador Python:

```
>>> nome = u"Karen Luise"
```

A codificação Unicode introduz novas sequências de escape em uma string de modo a permitir a representação dos diversos símbolos suportados.

5.2 Operações com strings

Os seguintes operadores podem ser utilizados sobre strings. É oportuno mencionar que pelo fato das strings serem objetos imutáveis, nenhuma operação pode modificar uma string existente, apenas gerar uma nova string com o resultado apropriado:

- **Comparação de ordem:** <, <=, >, >=:

  além da habitual comparação de igualdade e desigualdade através dos operadores == e !=, é possível comparar duas strings com respeito a ordenação relativa entre as mesma por meio dos operadores <, <=, >, >=. Para determinar a ordenação relativa entre as strings, o interpretador Python usará a ordem alfabética. Na realidade, ele se baseia na ordem dos caracteres na codificação ASCII, que coloca os caracteres em ordem alfabética com todo o conjunto de caracteres maiúsculos vindo antes de todo o conjunto de caracteres minúsculos. Assim, dadas duas strings alfabéticas `string1` e `string2`, a comparação `string1 < string2` retornará `True` se, ao adotar a ordenação alfabética (ASCII), `string1` vier antes de `string2`.

  Observe o exemplo a seguir. A comparação `"carol" < "mayara"` retorna `True` porque pela ordem alfabética, o texto `"carol"` viria antes de `"mayara"`.

  ```
  >>> "carol" < "mayara"
  True
  ```

  É preciso lembrar no entanto, que qualquer caractere maiúsculo vem antes de qualquer caractere minúsculo. assim, a comparação `"carol" < "Mayara"` retornará `False`, pois, pela ordenação ASCII, o `M` maiúsculo de `"Mayara"`

[1] É possível consultar todos os caracteres representados na codificação ASCII com seus respectivos códigos no endereço `https://www.asciitable.com/`

faria com que esse texto viesse antes de qualquer texto iniciado por caractere minúsculo.

```
>>> "carol" < "Mayara"
False
```

Quando as strings sendo comparadas possuírem caracteres não alfabéticos, será utilizada a ordem dos caracteres na codificação ASCII para determinar qual das mesmas viria primeiro em uma ordenação.

- **Comprimento: `len`**

  `len` retorna o comprimento (*lenght*) de objetos sequenciais em geral. No caso de uma string, o comprimento é dado pelo número de caracteres que a compõem.

```
>>> len("Diana")
5
```

```
>>> g = "mariana"
>>> len(g)
7
```

- **Concatenção: `+`**

  `+` concatena duas strings, isto é, gera uma nova string a partir da junção de duas strings:

```
>>> t = "abra" + "cadabra"
>>> t
"abracadabra"
```

- **Construção - Conversão para string: `str`**

  `str` funciona como construtor da classe, isto é, é capaz de gerar objetos string a partir de outros objetos.

```
>>> str(777)
"777"
```

```
>>> h = 49
>>> k = str(h)
>>> k
"49"
```

- **Formatação: `%`**:

  `%` permite a composição de uma string a partir da introdução de valores oriundos de objetos externos (formatação de string). O código `%d`, por exemplo permite a introdução de números inteiros em uma string:

```
>>> t = "hoje é dia %d do mes de agosto"%(10)
>>> t
"hoje é dia 10 do mes de agosto"
```

No Código 5.2, o código `%d` indica que um valor externo a string entrará na exata posição em que o mesmo aparece. Esse objeto externo, que no caso é o número 10, é indicado após o fechamento da string por meio do

operador `%`. Observe, a seguir, que é possível a introdução de mais de um valor externo a string:

```
>>> tex = "hoje é dia %s do mes %d do ano %d"%(10, 8, 2025)
>>> tex
"hoje é dia 10 do mes 8 do ano 2025"
```

O código `%f`, por sua vez, permite a introdução de números `float` na string. O programador C mais atento notará a similaridade com o padrão de uso da função `printf` desta linguagem:

```
>>> p = "%f metros"%(2.45)
>>> p
"2.450000 metros"
```

Pode-se especificar o tamanho do campo de inserção e o número de casas decimais utilizadas através de um número decimal no código. Por exemplo, o código `%0.3f` especifica que desejamos inserir um número float com 3 casas decimais:

```
>>> p = "%0.3f metros"%(2.45)
>>> p
"2.450 metros"
```

Já o código `%12.3f` indica que desejamos um campo com tamanho de 12 caracteres e 3 casas decimais:

```
>>> p = "%12.3f metros"%(2.45)
>>> p
"       2.450 metros"
```

Note, no exemplo anterior, que 7 espaços foram adicionados antes do número de modo a preencher todo o campo de 12 caracteres, já que o número em questão foi representado com apenas 5 caracteres.

Por fim, pode-se introduzir qualquer objeto em uma string através do código genérico `%s`:

```
>>> frase = "%s tem %s anos"%("Laura", 25)
>>> frase
'Laura tem 25 anos'
```

- **Participação como membro:** `in` e `not in`

  `in` retorna `True` se um objeto aparece em uma sequência e `False` caso contrário. No caso de strings, a operação Assim `string1 in string2` retornará `True` se `string1` aparecer em `string2`:

```
>>> "jes" in "jessica"
True
```

```
>>> "lim" in "Camila"
False
```

```
>>> "z" in "lueli"
False
```

```
>>> "ANA" in "mariana"
False
```

  Neste último exemplo, o operador `in` retorna `False` porque caracteres maiúsculos são diferenciados de minúsculos

Por sua vez, o operador `not in` fornece o resultado oposto ao do operador `in`, isto é, retorna `True` se a string à esquerda não aparecer dentro da string à direita:

```
>>> "ANA" not in "mariana"
True
```

- **Repetição: `*`**

  `*` gera uma nova string a partir da repetição de outra string:

```
>>> exp = "ai"*3
>>> exp
"aiaiai"
```

- **String vazia: `""`**

```
>>> tcc = ""
>>> tcc
""
```

5.3 Indexação e Fracionamento

Uma vez que as strings são sequências ordenadas, podemos acessar seus elementos pelo seu índice (posição). Começa-se a numerar os índices a partir do zero.

```
>>> aux = "universo"
```

Para a string `"universo"` definida acima, numera-se cada um de seus caracteres começando pelo zero, conforme a seguir:

0	1	2	3	4	5	6	7
u	**n**	**i**	**v**	**e**	**r**	**s**	**o**

Assim, utilizamos colchetes para indicar que nos referimos a um determinado índice da string. Por exemplo, `aux[0]` se remete ao elemento na posição 0 da string apontada pela variável `aux`:

```
>>> aux[0]
"u"
```

```
>>> c = aux[4]
>>> c
"e"
```

```
>>> aux[3]
"v"
```

```
>>> k = 5
>>> aux[k]
"r"
```

No nosso exemplo, pelo fato da string `"universo"` conter 8 caracteres, o maior índice que pode ser acessado é o 7. Assim, obtemos um erro se tentarmos acessar índices maiores do que esse valor:

```
>>> aux[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Uma forma genérica de acessar o último caractere de uma string não vazia, independentemente de seu tamanho, é por meio do uso do operador `len`:

```
>>> aux[len(aux) - 1]
"o"
```

Note que foi preciso subtrair o comprimento da string de uma unidade para acessar o último caracter, pois a contagem dos índices se inicia a partir do zero.

Python também define índices negativos para sequências ordenadas em geral. Nesse caso, a contagem é feita de forma reversa. Para o nosso exemplo, temos:

0	1	2	3	4	5	6	7
u	**n**	**i**	**v**	**e**	**r**	**s**	**o**
-8	-7	-6	-5	-4	-3	-2	-1

Desse modo, um elemento qualquer de uma sequência ordenada em Python podem ser acessados por meio do seu índice positivo, ou equivalentemente, por meio do seu índice negativo:

```
>>> aux[-7]
"n"
```

```
>>> aux[-4]
"e"
```

```
>>> aux[-1]
"o"
```

Portanto, conforme o exemplo anterior, um jeito mais simples de acessar o último elemento da sequência é através do índice -1.

Pelo fato das strings serem objetos imutáveis, não podemos trocar um determinado elemento por outro. Por exemplo, se tentarmos trocar o caracter "s"no índice por "b", obteremos um erro:

```
>>> aux[6] = "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

A operação acima faria total sentido se o objeto apontado por `aux` fosse alguma sequência ordenada mutável, como, por exemplo uma lista. Assim, em muitos casos, para realizar processamento de texto, é comum converter a string para lista, realizar as alterações desejadas e converter a lista alterada para string novamente, como no Código 5.5 a seguir:

```
1 >>> texto = list(aux)
2 >>> texto[6] = "b"
3 >>> aux2 = "".join(texto)
4 >>> aux2
5 "univerbo"
```

Código 5.5: realizando alteração em um texto por meio de conversão para lista.

Na linha 1 do Código 5.5 geramos um objeto lista (`list`) a partir da string em `aux`. Esse objeto lista será armazenado na variável `texto`. Em seguida, na linha 2 realizamos a alteração do elemento no índice 6 da lista em `texto`, sobrescrevendo-o com o caracter `"b"`. Na linha 3 geramos uma string a partir dos

elementos de `texto` através de um método denominado `join` e a armazenamos na variável `aux2` (discutiremos métodos de string na seção 5.7). Por fim, na linha 4 ecoamos a string obtida de modo a confirmar que a alteração desejada foi de fato realizada.

Através dos índices, também é possível acessar uma fração de uma determinada sequência ordenada. Por exemplo, a operação `aux[i:j]` se remete a fração da sequência apontada por `aux` que vai do índice `i` até o índice imediatamente anterior a `j`, isto é, vai de `i` até `j` sem incluir `j`, conforme os exemplos a seguir:

```
1 >>> aux[0:4]    #toma a substring do índice 0 até o índice 3
2 "univ"
```

A operação de fracionamento gera um novo objeto na memória. Podemos inclusive atribuir esse novo objeto a uma variável:

```
1 >>> p = aux[2:6]
2 >>> p
3 "iver"
```

É possível especificar um passo (intervalo) para o fracionamento. Por exemplo, se desejarmos obter a substring que vai do caracter 0 até o caracter 7, mas saltando de dois em dois, podemos fazer:

```
1 >>> aux[0:7:2]
2 "uies"
```

Note, no exemplo anterior, o uso de mais um `:` para especificar o passo do fracionamento. Se omitirmos o segundo índice, o interpretador assume que a substring deve ir até o final da string:

```
1 >>> aux[3: :1]
2 "verso"
```

Se o primeiro índice for omitido, o interpretador assume que a substring deve se iniciar desde o começo da string:

```
1 >>> aux[ :3]
2 "uni"
```

Podemos usar um passo negativo para percorrer a string ao reverso:

```
1 >>> aux[7:3:-1]
2 "osre"
```

```
1 >>> aux[7:3:-1]
2 "osre"
```

```
1 >>> aux[-1:-5:-1]
2 "osre"
```

Assim, um jeito fácil de obter uma sequência ordenada de trás para frente é fazendo:

```
1 >>> aux[ : :-1]
2 "osrevinu"
```

5.4 Percorrendo uma string

Podemos percorrer os elementos de uma string um a um por meio de um laço de repetição. O Código 5.6 percorre uma string através de seus índices.

```
1 dado = "Rachel"
2 for k in range(0, len(dado)):
3     print( dado[k] )
4 print("Tenha um bom dia!")
```

Código 5.6: percorrendo uma string por meio de seus índices.

Na linha 1 do Código 5.6, define a string que será percorrida atribuindo-a à variável `dado`. Na linha 2, declaramos um laço `for` onde a variável `k` percorrerá os índices da string. Note que `k` itera no intervalo de 0 até o comprimento da string em `dado` (sem incluir `dado`). A linha 3, que compõe o bloco de repetição, imprime o caracter na posição `k`. Após a execução do laço, a linha 4 imprime uma saudação de saudação e o programa é encerrado. Desse modo, o Código 5.6 imprimirá na tela:

```
R
a
c
h
e
l
Tenha um bom dia!
```

O Código 5.7 é equivalente ao Código 5.6. No entanto, no Código 5.7, o laço `for` na linha 2 faz com que a variável `c` percorra diretamente os elementos da string armazenada em `dado`. Em outras palavaras, `c` assumirá o valor de cada um dos caracteres de `dado`, um por vez, em sua respectiva ordem. Observe a diferença em comparação ao laço na linha 2 do Código 5.6, onde se percorrem os índices da string. O conteúdo impresso na tela pelo Código 5.7 é exatamente o mesmo daquele impresso pelo Código 5.6.

```
1 dado = "Rachel"
2 for c in dado:
3     print( c )
4 print("Tenha um bom dia!")
```

Código 5.7: percorrendo uma string iterando diretamente sobre seus elementos.

5.5 As funções ord e chr

As funções `ord` e `chr` se destinam a relacionar caracteres e sua respectiva codificação ASCII. Seja `x` uma string com um único caractere. `ord(x)` retornará o código ASCII do caractere em `x`. Por exemplo, para obter o código ASCII do `"A"`, podemos fazer:

```
1 >>> ord("A")
2 65
```

Note que os caracteres alfabéticos maiúsculos vêm em sequência na codificação ASCII a partir do código 65:

```
>>> ord("B")
66
```

```
>>> ord("C")
67
```

```
>>> ord("Z")
90
```

Por sua vez, os caracteres minúsculos se iniciam a partir do 97:

```
>>> ord("a")
97
```

```
>>> ord("b")
98
```

```
>>> ord("z")
122
```

Caracteres não alfabéticos também possuem código ASCII:

```
>>> ord("3")
51
```

```
>>> ord(" ")
32
```

```
>>> ord(";")
59
```

A função `chr`, por sua vez, realiza a operação reversa à de `ord`. Enquanto `ord` recebe um caracter e retorna seu respectivo código ASCII, `chr` recebe um código ASCII e retorna seu respectivo caracter associado:

```
>>> chr(65)
"A"
```

```
>>> chr(98)
"b"
```

```
>>> chr(43)
"+"
```

As funções `ord` e `chr` podem ser muito úteis em casos onde seja necessário realizar algum tipo de mapeamento entre caracteres que obedeça a alguma função matemática. Por exemplo, na Seção 5.6.4, é introduzido um programa onde estas funções são utilizadas para a "conversão" de caracteres minúsculos em maiúsculos.

5.6 Alguns exemplos

Nesta seção, faremos alguns exemplos básicos com strings. Os exemplos desta seção se tornariam mais simples com o uso dos métodos de strings, os quais veremos na Seção 5.7. O objetivo aqui é apenas adquirir prática e ilustrar o uso das operações de string até então vistas.

5.6.1 Removendo espaços

```
1  """Programa que lê uma string do teclado e a imprime
2  sem os caracteres espaço em branco"""
3
4  texto = input("Entre com o texto: ")
5
6  novotexto = ""
7  for c in texto:
8      if c != " ":
9          novotexto = novotexto + c
10
11 print("Texto sem espaços: ", novotexto)
```

Código 5.8: programa que remove espaços de uma string.

O Código 5.8 lê uma string (texto) do usuário e imprime o texto lido sem os caracteres espaço em branco. A linha 4 lê o texto do usuário e o armazena

na variável `texto`. Observe que aqui, não realizamos nenhuma conversão no dado lido do usuário, pois a função `input` já retorna uma string com o conteúdo digitado. A seguir, na linha 6, definimos a variável `novotexto` com uma string vazia. Essa variável tem a função de compor uma nova string com o conteúdo digitado pelo usuário sem os espaços em branco. Para tal, percorremos todos os caracteres da string em `texto`, e acrescentaremos em `novotexto`, através de concatenação de strings, apenas os caracteres diferentes de espaço em branco. Assim, o laço `for` declarado na linha 7 faz com que as variável `c` percorra cada caracter de `texto`. Já no bloco de repetição, a linha 8 testa se o caracter em `c` é diferente do caracter espaço em branco (`" "`). Caso seja, o caracter em `c` é concatenado com o conteúdo corrente da variável `novotexto` e armazenado na própria variável `novotexto` (linha 9). Como `c` assumirá como valor cada um dos caracter da string em `texto` na sua respectiva ordem, ao final da execução do laço `for`, `novotexto` conterá todos os caracteres de `texto`, com exceção dos espaços em branco, na sua respectiva ordem. Por fim, a linha 11 imprime a nova string composta. Note que a variável `novotexto` funciona como um acumulador que recebe uma espécie de somatório de caracteres nesse programa.

Um exemplo de execução desse programa seria:

```
Entre com o texto: um ano
Texto sem espaços:  umano
```

Acompanhando a evolução das variáveis

Como de praxe, caso não tenha compreendido totalmente o funcionamento do Código 5.8 aconselhamos a execução do mesmo com a impressão dos valores intermediários assumidos pela variável `novotexto`. Para tal seria suficiente colocar um `<tab>` a frente da linha 11. A seguir, faremos esse acompanhamento das mudanças nos valores das variáveis para o nosso exemplo de execução, onde o usuário entra com a string `"um ano"`:

0. **antes da execução do laço *for* na linha 7**: temos o seguinte quadro de variáveis:

Variáveis	Valores
`novotexto`	`""`
`texto`	`"um ano"`

1. **ao final da iteração 1 do for:** a variável `c` está com o valor do primeiro caracter de `texto` (`"u"`), e como o mesmo é diferente de espaço em branco, a linha 9 será executada e a variável `novotexto` passará a possuir o valor corrente de `novotexto` (string vazia) concatenado com o valor da variável `c`:

Variáveis	Valores
`c`	`"u"`
`novotexto`	`"u"`
`texto`	`"um ano"`

2. **ao final da iteração 2 do for:** agora a variável `c` assume o valor do segundo caracter de `texto` (`"m"`). Como este valor ainda difere do espaço

em branco, a linha 9 será novamente executada, e a variável `novotexto` assumirá o valor de `novotexto` (que agora é `"u"`) concatenado com o valor da variável `c`:

Variáveis	Valores
c	"m"
novotexto	"um"
texto	"um ano"

3. **ao final da iteração 3 do for:** nesta iteração, a variável `c` o terceiro caracter de `texto`, que é o espaço em branco. Nesse caso a linha 9 não será executada, o que fará com que a variável `novotexto` permaneça com o mesmo valor da iteração anterior:

Variáveis	Valores
c	" "
novotexto	"um"
texto	"um ano"

estendendo o raciocínio das iterações anteriores para as próximas iterações, temos:

4. **ao final da iteração 4 do for:**

Variáveis	Valores
c	"a"
novotexto	"uma"
texto	"um ano"

5. **ao final da iteração 5 do for:**

Variáveis	Valores
c	"n"
novotexto	"uman"
texto	"um ano"

6. **ao final da iteração 6 do for:**

Variáveis	Valores
c	"o"
novotexto	"umano"
texto	"um ano"

5.6.2 Contando o número de caracteres maiúsculos

```
1  """Programa que lê uma string do teclado e conta o
2  número de caracteres maiúsculos"""
3
4  texto = input("Entre com o texto: ")
5
6  nmaiusculos = 0
7  for c in texto:
8      if "A" <= c <= "Z":
9          nmaiusculos += 1
10
11 print("Numero de caracteres maiusculos: ", nmaiusculos)
```

Código 5.9: programa que conta o número de caracteres maiúsculos de uma string.

O Código 5.9 lê uma string (texto) do usuário e imprime o número de caracteres alfabéticos maiúsculos presentes na mesma. A linha 4 lê o texto do usuário (já como string) e o armazena na variável `texto`. Na linha 6, inicializamos a variável `nmaiusculos` com o valor 0. O objetivo desta variável é armazenar a contagem do número de caracteres maiúsculos da string em `texto`. Para isso, usamos a variável `c` para percorrer cada caracter dessa string com o laço `for` da linha 7. Assim, o bloco de repetição nas linhas 8-9 será executado uma vez para cada caracter da string `texto` (representado na variável `c`). Na linha 8, o programa testa se o caracter armazenado na variável `c` está entre o `"A"` (maiúsculo) e o `"Z"` (maiúsculo), isto é, se é um caractere alfabético maiúsculo. Em caso positivo, o valor da variável `nmaiusculos` é incrementado em uma unidade na linha 9. Assim, após a execução de todas as iterações do laço `for`, esta variável conterá o valor total de caracteres alfabéticos maiúsculos. Encerrando o programa, a linha 11 imprime a resposta esperada.

Um exemplo de execução desse programa seria:

```
Entre com o texto: MinGaU MataDor
Numero de caracteres maiusculos:  5
```

5.6.3 Contando o número de vogais

```
1  """Programa que lê uma string do teclado e conta o
2  número de vogais"""
3
4  texto = input("Entre com o texto: ")
5
6  nvogais = 0
7  for k in range(0, len(texto)):
8      if texto[k] in ("a","e","i","o","u","A","E","I","O","U"):
9          nvogais += 1
10
11 print("Numero de vogais: ", nvogais)
```

Código 5.10: programa que conta o número de vogais de uma string.

O Código 5.9 lê uma string (texto) do usuário e imprime o número de vogais não acentuadas presentes na mesma. Note que este código é similar ao Código 5.9. Um diferença está no modo como percorremos a string em `texto`. Observe que no Código 5.9, a variável `c` percorre os caracteres da string em `texto` diretamente. Já no Código 5.10, o `for` da linha 7 faz a variável `k` iterar sobre

os **índices** de `texto`. Assim, para percorrer os caracteres de `texto`, foi preciso usar a forma `texto[k]` na linha 8, uma vez que `k` varia sobre seus índices. Note ainda que o teste da linha 8 usa o operador `in` em conjunto com uma tupla que enumera as vogais maiúsculas e minúsculas. A seguir, um exemplo de execução desse código:

```
Entre com o texto: princesa JESSICA
Numero de vogais:  6
```

É válido apontar que poderíamos ter construído o laço `for` na linha 7 do Código 5.10 de modo a fazer uma variável `k` percorrer diretamente os caractere de `texto`, no lugar de percorrer seus índices. Optamos por essa forma nesse exemplo para demonstrar esse modo alternativo de percorrer uma string, visto que, em alguns casos, essa forma mais genérica é mais apropriada para a resolução de certos tipos de problemas, especialmente em casos onde é preciso percorrer diversas strings simultaneamente.

5.6.4 "Convertendo" caracteres minúsculos em maiúsculos com *ord* e *chr*

As funções `ord` e `chr` nos permitem manipular os códigos ASCII de caracteres. Estas funções podem ser realizadas para realizar mapeamentos entre caracteres que sejam definidos por funções matemáticas. Por exemplo, na Seção 5.5, vimos que o código ASCII dos caracteres alfabéticos minúsculos se iniciam a partir do número 97 seguindo a ordem alfabética. Assim, sabemos que o `"a"` possui o código 97, ao passo que `"b"` possui o código 98, o `"c"` possui o código 99, e assim, sucessivamente. Do mesmo modo, sabemos que os caracteres alfabéticos maiúsculos são introduzidos a partir do 65, com `"A"` tendo o código 65, `"B"` tendo o código 66, e assim, por diante. Desta forma, dado um determinado caracter alfabético minúsculo, sabemos que seu corresponde maiúsculo tem código ASCII 32 posições abaixo, o que significa que o código do caracter maiúsculo pode ser obtido ao se subtrair 32 do código do respectivo caracter minúsculo. O Código 5.11 utiliza esta curiosa propriedade para, a partir de uma string lida do usuário, construir uma nova string onde todos os caracteres minúsculos (não acentuados) são convertidos para maiúsculos:

```
1  """Programa que lê uma string do teclado e a imprime
2  com os caracteres minúsculos transformados em maiúsculos"""
3
4  texto = input("Entre com o texto: ")
5
6  novotexto = ""
7  for c in texto:
8      if "a" <= c <= "z":
9          carac = chr( ord(c) - 32 )
10     else:
11         carac = c
12     novotexto += carac
13
14 print("Novo texto: ", novotexto)
```

Código 5.11: programa que converte caracteres minúsculos para maiúsculos.

A linha 4 lê uma string do usuário. Em seguida, a linha 6 inicializa a variável `novotexto` com uma string vazia. O objetivo desta variável é armazenar a

string modificada, na qual os caracteres minúsculos da string em `texto` serão convertidos para minúsculos. Na linha 7, declaramos um laço `for` que fará a variável `c` percorrer cada caracter da string em `texto`. Já no laço de repetição, o `if` na linha 8 testa se o caracter correntemente armazenado na variável `c` está entre `"a"` (minúsculo) e `"z"` (minúsculo), isto é, teste se este caracter é alfabético minúsculo. Em caso positivo, é realizada a conversão na linha 9 de minúsculo para maiúsculo. Observe que o código ASCII do caracter em `c` é obtido através da função `ord`, subtraído de 32, convertido novamente para caracter com a função `chr` e armazenado na variável `carac`. Se o caracter em `c` não for alfabético minúsculo, não é necessária a realização de nenhuma conversão, e o próprio caracter em `c` é armazenado em `carac` na linha 11 dentro do bloco subordinado ao `else` (linha 10). O passo seguinte é a acumulação do caracter em `carac` na string armazenada em `novotexto` (linha 12). Note que, por comporem o bloco de repetição, as linhas 8-12 serão executadas uma vez para cada valor de `c`, sendo que, esta última variável assume cada caracter da string em `novotexto`, um por vez, em sua respectiva ordem. Finalizando o programa, a linha 14 imprime a string com os resultado esperado.

```
Entre com o texto: FE Garay!
Novo texto:  FE GARAY!
```

Note que, para fazer o Código 5.11, foi preciso saber que a diferença entre o código de um caracter maiúsculo e o de seu respectivo minúsculo é 32. Se não soubéssemos o valor exato dessa diferença, ainda assim poderíamos ter desenvolvido este programa substituindo o conteúdo da linha 9 pela expressão equivalente `carac = chr( ord(c) - ord("a") + ord("A") )`. Alguns podem até argumentar que, embora mais confusa a primeira vista, esta última forma deixa o código mais elegante, pois constantes soltas no código como o número 32 na linha 9 podem prejudicar o entendimento e a manutenibilidade de programas.

Como de costume, o leitor que ainda se encontrar confuso quanto ao funcionamento do Código 5.11 é encorajado a executar esse código com uso de `print`'s adicionais para acompanhar a evolução das variáveis no laço de execução.

5.7 Métodos de String

Nesta seção, apresentamos alguns dos métodos para string. No contexto de programação orientada a objetos , métodos são procedimentos (funções) específicos para uma determinada classe de objetos. Em geral, tipos de objetos (classes) podem definir uma série de métodos, e cada um desses métodos podem ser chamados a partir de qualquer objeto pertencente ao tipo (classe). Por exemplo, o tipo `str` define métodos que podem ser chamados por qualquer objeto string. Um desses métodos, denominado `upper` gera, a partir de uma string, uma nova string convertendo caracteres minúsculos para maiúsculos, conforme o exemplo:

```
1 >>> nome = "walewska"
2 >>> nome.upper()
3 "WALEWSKA"
```

Assim, através do uso do método `upper`, podemos reescrever o Código 5.11 de um modo muito mais simplificado:

```
1 """Programa que lê uma string do teclado e a imprime
2 com os caracteres minúsculos transformados em maiúsculos"""
3
4 texto = input("Entre com o texto: ")
5 novotexto = texto.upper()
6 print("Novo texto: ", novotexto)
```

Código 5.12: programa que converte caracteres minúsculos para maiúsculos com o método *upper*.

O Código 5.12 possui ainda a vantagem de funcionar também com caracteres alfabéticos acentuados. Observe que a maneira mais usual de utilizar um método de classe é escrevendo: `<objeto>.<nome do metodo>( <argumentos> )` , conforme a linha 5. Ao executar o método `upper` nessa linha, dizemos que o mesmo foi *chamado* ou *invocado*.

Em geral, métodos podem operar a partir do objeto pelo qual os mesmos foram chamados, receber argumentos de entrada, retornar valores ou até mesmo modificar, em alguns casos, o objeto a partir do qual foram chamados, caso este seja mutável. Para o caso específico das strings, que são objetos imutáveis, métodos não podem alterar o seu conteúdo. Assim, métodos da classe `str` como `upper` podem apenas retornar uma nova string com o conteúdo desejado, ou algum outro valor qualquer, dependendo de seu objetivo. A lista completa dos métodos de uma classe, com uma breve descrição, pode ser conferida através do uso da função `help` no prompt:

```
1 >>> help( str )
```

Pode-se também conferir o texto de ajuda específico de um determinado método:

```
1 >>> help( str.upper )
```

Alguns dos métodos de uso mais comum de `str` são:

- `count( substring )`: retorna a quantidade de vezes em que `substring` aparece na string sem sobreposição.

  ```
  1 >>> s = "abracadabra"
  2 >>> s.count("abra")
  3 2
  ```

  é possível passar argumentos adicionais especificando índices de início e de fim da contagem. Por exemplo, para contar a partir do terceiro caracter até o nono, lembrando sempre que a contagem dos índices se inicia no zero, basta fazer:

  ```
  1 >>> s = "abracadabra"
  2 >>> s.count("cada", 2, 8)
  3 1
  ```

- `find( substring )`: retorna o menor índice positivo onde `substring` ocorre na string, ou -1, caso não ocorra:

```
>>> n = "mariana"
>>> n.find("ria")
2
>>> n.find("chore")
-1
```

- `join( iteravel )`: retorna uma string concatenando strings presentes em um objeto iterável como uma lista ou uma tupla. A string utilizada para chamar o método é usada como separador entre as strings presentes no objeto iterável:

```
>>> palavras = ("vovô", "viu", "a", "uva")
>>> "".join( palavras )
"vovôviuauva"
>>> " ".join( palavras )
"vovô viu a uva"
>>> "XXX".join( palavras )
"vovôXXXviuXXXaXXXuva"
```

  No primeiro exemplo usando o método `join`, usamos como separador uma string vazia, ou seja, realizamos a concatenação sem separador. No segundo exemplo, usamos uma string com o caracter espaço (`" "`) e, deste modo, a string retornada conteve este caracter como separador. Por fim, no terceiro exemplo, usamos como separador a string `"XXX"`.

- `isalpha()`: retorna `True` se a string é composta **inteiramente** por caracteres alfabéticos, e `False` caso contrário:

```
>>> t = "jessica"
>>> t.isalpha()
True
```

```
>>> t = "barco da paz"
>>> t.isalpha()
False
```

  Note que o caracter espaço (`" "`) não é alfabético!

- `isdigit()`: retorna `True` se **todos** os caracteres da `string` são dígitos, e `False` caso contrário:

```
>>> n = "5209"
>>> n.isdigit()
True
```

```
>>> "ziriguidum".isdigit()
False
```

- `islower()`: retorna `True` se a string **não** possui qualquer caractere alfabético maiúsculo, e `False` caso contrário:

```
>>> b = "gabriela, canela!"
>>> b.islower()
True
```

```
>>> b = "Samantha Barbara"
>>> b.islower()
False
```

- `isupper()`: retorna `True` se a string **não** possui qualquer caractere alfabético minúsculo, e `False` caso contrário:

```
>>> c = "UFRJ"
>>> c.isupper()
True
```

```
>>> d = "Laura Fernanda "
>>> d.isupper()
False
```

- `lower()`: retorna uma nova string onde caracteres alfabéticos maiúsculos são convertidos para minúsculos:

```
>>> tex = "o galo canta COCORICÓ"
>>> tex.lower()
"o galo canta cocoricó"
```

- `replace( origem, destino )`: retorna uma nova string onde cada ocorrência de `origem` é subtituída por destino:

```
>>> sent = "Quem casa quer casa!"
>>> sent.replace( "casa", "fala" )
"Quem fala quer fala!"
```

  É possível passar um argumento opcional `maximo` especificando a quantidade máxima de substituições. Nesse caso, apenas as primeiras `maximo` aparições de `origem` serão substituídas:

```
>>> q = "ai ai ai ai"
>>> q.replace("ai", "hey", 3)
"hey hey hey ai"
```

  Para remover `origem` da string gerada, basta usar a string vazia como `destino`:

```
>>> r = "Eu não gosto de não estar presente!"
>>> r.replace(" não", "")
"Eu gosto de estar presente!"
```

- `split( separador )`: retorna uma lista de substrings. Essas substrings são separadas usando `separador` como delimitador:

```
>>> red = "Plantei bola pé de bola flor deu bola capim"
>>> red.split("bola")
["Plantei ", " pé de ", " flor deu ", " capim"]
```

  Note que `separador` não é considerado no resultado. Se `separador` for omitido, qualquer caracter que possa ser interpretado com espaço em branco (espaço, enter, tabulação, etc) será usado como delimitador:

```
>>> red.split()
["Plantei", "bola", "pé", "de", "bola", "flor", "deu", "
    bola", "capim"]
```

- `upper()`: retorna uma nova string onde caracteres alfabéticos minúsculos são convertidos para maiúsculos:

```
>>> frase = "As Rosas Não Falam"
>>> frase.upper()
"AS ROSAS NÃO FALAM"
```

5.7.1 Contando consoantes de uma string

```
1  """Programa que lê uma string do teclado e conta o
2  número de vogais"""
3
4  texto = input("Entre com o texto: ")
5  textoMin = texto.lower()
6
7  consoantes = 0
8
9  for c in textoMin:
10     if c.isalpha() and c not in ("a", "e", "i", "o", "u"):
11         consoantes += 1
12
13 print("Numero de consoantes: ", consoantes)
```

Código 5.13: programa que converte caracteres minúsculos para maiúsculos.

O Código 5.13 lê uma string do usuário e informa o número de consoantes presentes, incluindo as acentuadas como `"Ç"`. Para essa contagem de consoantes, é preciso estar atento ao fato de que o texto digitado pelo usuário pode conter consoantes maiúsculas e minúsculas simultaneamente. Por essa razão, com o objetivo de facilitar a contagem, após a leitura da string do usuário na linha 4 e sua atribuição à variável `texto`, é gerada na linha 7 uma nova versão dessa string com os caracteres maiúsculos convertidos para minúsculos através do método `lower`. Essa nova string é então atribuída à variável `textoMin`. Observe que, ao realizar a contagem de consoantes a partir de `textoMin` no lugar de `texto`, só é necessária a preocupação com consoantes minúsculas. Dessa forma, após inicializar o contador de consoantes com zero na linha 7 (variável `consoantes`), o laço `for` na linha 9 faz a variável `c` iterar sobre `textoMin`, o que significa que `c` assumirá o valor de cada caracter de `textoMin`, um por vez, em sua respectiva ordem. Já dentro do bloco de repetição, o `if` da linha 10 usa o método `isalpha`, para testar se o valor corrente em `c` é alfabético, em conjunto com os operadores `and` e `not in` para testar se o mesmo também não é vogal. Se ambos os testes resultarem em verdadeiro, a linha 11 incrementa o contador de consoantes. Por fim, a linha 13 exibe ao usuário o valor da contagem realizada.

5.8 Exercícios

1. Faça um programa que leia uma string do teclado e informe a quantidade de caracteres alfabéticos maiúsculos na string.

   Exemplo:

   ```
   Entre com uma string: MuiTA atençÃO!
   Numero de caracteres alfabéticos maiúsculos: 5
   ```

2. Faça um programa que leia uma string do teclado e imprima essa mesma string com os caracteres alfabéticos com caixa invertida, isto é, os caracteres maiúsculos devem ser impressos minúsculos e os maiúsculos devem ser impressos minúsculos.

   Exemplo:

```
Entre com o texto: O Amor eh FoGO quE arDE SeM se VER
Texto invertido: o aMOR EH fOgO QUe ARde sEm SE ver
```

3. Um palíndromo é uma palavra ou frase que tenha a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita com igual significado.

   Em um palíndromo, normalmente são desconsiderados os sinais ortográficos (diacrítico ou de pontuação), assim como o espaços entre palavras. As seguintes frases são exemplos de palíndromo:

   - Ande Edna
   - Ame o poema
   - Após a sopa
   - Socorram-me, subi no ônibus em Marrocos

   Escreva um programa que leia uma string do teclado e informe se a mesma é um palíndromo. Assuma que a string lida nunca conterá caracteres alfabéticos acentuados.

   Exemplos

```
Entre com o texto: sarros
Este texto nao e um palindromo
```

```
Entre om o texto: ame o poema
Este texto e um palindromo
```

```
Entre om o texto: ande, edna
Este texto e um palindromo
```

```
Entre com o texto: amora
Este texto nao e um palindromo
```

4. Escreva um programa que leia duas strings do teclado e informe se todos os caracteres da primeira string também aparecem na segunda.

   Exemplos:

```
Entre com a primeira string: amor bandido
Entre com a segunda string: andei mascarado na boate

Todos os caracteres da primeira string aparecem na segunda
```

```
Entre com a primeira string: viva
Entre com a segunda string: Eternamente estou

Nem todos os caracteres da primeira string aparecem na
segunda
```

5. A organização secreta "Guardiões da luz da juventude"utiliza o seguinte esquema para a codificação de suas mensagens ultra-secretas: 1) Sinais de

pontuação, números e espaços devem ser mantidos como estão na mensagem. 2) Cada letra deve ser substituída pela letra imediatamente subsequente no alfabeto, a exceção da letra "Z", que deve ser substituída pela letra "A". Escreva um programa que leia uma mensagem em sua forma original do teclado e faça a codificação da mensagem segundo o esquema da organização.

Exemplo:

```
Entre com a mensagem: O horario da partida ZETA e 12:45

Mensagem codificada: P ipsbsjp eb qbsujeb AFUB f 12:45
```

6. Tia Liliane, professora do Jardim Escola Pentelho Feliz, aplica testes de múltipla escolha aos seus alunos, onde cada opção de resposta à uma determinada questão é associada a uma letra de "A"até "E". As respostas de um determinado teste são armazenadas em uma string. Por exemplo, a string de respostas do último teste de Joãozinho foi:

   "CABED"

   indicando que Joãozinho respondeu "C"na primeira questão, "A"na segunda, "B"na terceira, "E"na quarta e "D"na quinta. Assim, a string de respostas sempre tem como comprimento, o número de questões da prova (quando algum aluno deixa uma questão em branco, o caracter espaço é utilizado). De forma análoga, o gabarito de cada teste também é representado como uma string com as opções corretas para cada questão. Em alguns casos, questões específicas do teste podem vir a ser anuladas. Nesse caso, o caracter "!"é utilizado para indicar a anulação da questão. Por exemplo, a string do gabarito do último teste foi :

   "C!ABD"

   indicando que a resposta correta da primeira questão é "C", a segunda questão foi anulada, ao passo que as respostas corretas das questões 3, 4 e 5 são A, B e D, respectivamente. Assumindo que cada questão respondida corretamente vale 10 pontos, e que quando uma questão é anulada, os alunos ganham a pontuação correspondente à questão independentemente do que tenham respondido, faça um programa que leia a string do gabarito, leia a string de respostas do aluno e informe a sua pontuação. Note que, cada teste pode ter um número arbitrário de questões.

   Exemplos:

```
Entre com o gabarito: BACD!EBC
Entre com as respostas do aluno: CACEBBDC

Pontuação do aluno: 40
```

7. Nos jogos de um determinado campeonato de futebol, cada time pode vencer, perder ou empatar. Vitórias contabilizam 3 pontos para o time vencedor e 0 pontos para o time perdedor, enquanto empates contabilizam

um ponto para cada time. Os resultados de cada time são armazenados em uma string onde 'V' representa vitória, 'D' representa derrota e 'E' representa o empate. Por exemplo, a string de vitórias do Tabajara Futebol clube no ano de 1958 foi:

"DDDVVE"

Indicando que o time perdeu as três primeiras partidas, venceu as duas seguintes e empatou a última. A sua tarefa é escrever um programa que leia as strings de resultado de cada time do campeonato e informe qual foi o time campeão, isto é, qual time acumulou a maior quantidade de pontos. Assuma que só existe um campeão para cada campeonato e que as strings de resultados só conterão caracteres maiúsculos válidos. Note que você NÃO deve perguntar a quantidade de jogos dos times:

Exemplo:

```
Entre com o numero de times: 4

Entre com a string de resultados do time 1: VVE
Entre com a string de resultados do time 2: DEV
Entre com a string de resultados do time 3: DEE
Entre com a string de resultados do time 4: VDD

Maior pontuacao: 7
Campeao: time 1
```

8. (**Desafio**) Faça um programa que leia duas strings do teclado e informe se a segunda string aparece, exatamente como foi lida, dentro da primeira. Para fazer esse programa, não é permitido utilizar os operadores `in` e `:` além de qualquer método da classe `str`.

   Exemplos:

```
Entre com a primeira string: Meu cacareco azul
Entre com a segunda string: careco

A segunda string aperece dentro da primeira
```

```
Entre com a primeira string: Vestido CurTo
Entre com a segunda string: curto

A segunda string nao aperece dentro da primeira
```

```
Entre com a primeira string: aaab
Entre com a segunda string: aab

A segunda string aperece dentro da primeira
```

```
Entre com a primeira string: O nome dela e Jessica
Entre com a segunda string: nome dela

A segunda string aperece dentro da primeira
```

Capítulo 6

Funções

No contexto de programação, podemos definir uma função como sendo uma porção de código que pode receber argumentos de entrada, realizar procedimentos específicos e retornar (ou não) um valor. A porção de código que define uma função possui um certo isolamento em relação ao restante do código, e pode ser convocada (chamada) a executar um número arbitrário de vezes em um programa.

Até aqui, já utilizamos algumas funções pré-definidas pela própria linguagem Python. Por exemplo, usamos a função `input` sempre que precisamos obter algum dado digitado pelo usuário. Usamos a função `print` quando precisamos exibir informação na tela e `range` para formar objetos sequenciais a partir de progressões aritméticas.

Podemos pensar em uma função como sendo uma máquina fechada que executa um determinado procedimento. De um lado, a função recebe insumos, que seriam os *argumentos de entrada*. Do outro lado, com base nos argumentos de entrada recebidos, a função pode devolver algum produto, que é denominado retorno ou *argumento de saída*.

Por exemplo, a função `round` serve para calcular o arredondamento de um número real `float` até o valor inteiro mais próximo como no exemplo a seguir:

```
1 >>> round(6.8)
2 7
```

Código 6.1: exemplo de uso da função *round*.

No Código 6.1, temos, na linha 1, uma chamada à função `round`. Aproveitamos para apontar desde já, que o uso dos parênteses após o nome da função é o que efetivamente faz a função ser convocada (chamada) a executar seu procedimento. Dentro do par de parênteses, passamos os argumentos de entrada para a função. Esta função recebe um único argumento de entrada, que é o número real a partir do qual o arredondamento será calculado. Desse modo, no contexto do Código 6.1, o valor `6.8` é o argumento de entrada passado à `round`. A partir desse valor, a função realiza seu procedimento (cálculos) e retorna (devolve) o valor `7`, que é o arredondamento de 6.8 para o valor inteiro mais próximo. Assim, o valor `7` é denominado como retorno ou argumento de saída da função no âmbito do Código 6.1.

A partir do Código 6.1, pode-se desde discutir aspectos relacionados ao uso de funções. Primeiramente, ao utilizar uma função como `round`, precisamos

saber o que a função faz, de um modo geral, o que implica também em saber como a mesma recebe os argumentos de entrada e qual é a saída esperada a partir destes. Todavia, observe que não é preciso conhecer os detalhes internos de como a função realiza as suas operações. Assim, do ponto de vista do utilizador de `rand`, é suficiente saber que a função realiza arredondamento de números reais para o número inteiro mais próximo, mas não é preciso ter ciência do código exato que compõe a função `rand`. Ao utilizar a função já pronta `rand`, é como se, de certo modo, terceirizássemos a tarefa de fazer um trecho de código que calcule esse arredondamento, uma vez que esta função foi construída por outra pessoa. Assim, por meio do uso de funções, é possível construir programas a partir de porções de código escritas por terceiros, o que aumenta a produtividade e permite que seja possível desenvolver programas cada vez mais complexos a partir de códigos pré-existentes.

Podemos ainda apontar como benefícios trazidos pelo uso das funções:

- *Permitem a reutilização de código*, o que pode aumentar a produtividade, diminuir a quantidade de erros no código e tornar programas menores e mais fáceis de dar manutenção;

- *Possibilitam a decomposição de procedimentos*: decompor uma tarefa complexa em uma série de subtarefas de menor complexidade pode ser uma boa estratégia para a resolução de um problema. Essa decomposição também facilita a esquematização quando existem subtarefas que são executadas diversas vezes ao longo do processo como um todo;

- *Podem tornar um código mais legível*: no lugar de usar a função `rand` no Código 6.1, poderíamos nós mesmos ter feito código que fizesse o arredondamento por meio de `if`, `else` e algumas operações. Todavia, é muito mais simples entender um código de uma linha que traga o nome `rand`, do que um bloco de diversas linhas de código que faça a operação equivalente. Isso, é claro, quando o nome da função é bem escolhido de modo a dar uma boa noção do que ela faz. Desse modo, fica desde já o conselho para a escolha de bons nomes para suas variáveis e funções.

- *Facilitam a manutenção e a correção de erros*: imagine que um programa necessite fazer arredondamento diversas vezes, mas seu programador não fez uso de função para tal operação. Se, posteriormente, este programador descobrir que havia um erro na sua lógica de cálculo de arredondamento, ele precisará varrer todo o programa consertando todos os trechos onde essa lógica foi utilizada. No entanto, se este mesmo programador tivesse utilizado uma função apropriada para fazer este arredondamento, ainda que se descobrisse um erro na lógica dessa função, só seria preciso consertar a porção de código que define essa função uma única vez. A partir daí, todo o restante do código que fizesse chamada a essa função estaria automaticamente consertado. Além disso, o próprio fato de possibilitar reutilização de código, permitir decomposição e aumentar a legibilidade também faz com que o bom uso das funções simplifique a manutenção do código fonte.

6.1 Definindo suas próprias funções

No Código 6.1, utilizamos a função `round` para calcular o arredondamento de um número real. Por ser uma função pré-definida da linguagem Python, não foi preciso que definíssemos a porção de código que especifica suas operações, pois a mesma já foi definida pelos próprios desenvolvedores da linguagem em algum momento. Nem sempre há funções pré-definidas que implementem os procedimentos de que precisamos em um contexto. Por essa razão, em muitos casos, é necessário definir nossas próprias funções no desenvolvimento de um programa.

Podemos declarar funções através da cláusula `def`. A forma geral é exibida pelo Código 6.2:

```
1 def <nome da função> (argumento1, argumento2, ..., argumenton):
2 <tab> <instrução 1>
3 <tab> <instrução 2>
4 ⋮
5 <tab> <instrução n>
6 <primeira instrução pós-função>
```

Código 6.2: forma geral da cláusula *def* para a defnição de funções.

Na linha 1 do Código 6.2, temos a chamada linha de declaração de uma função. Nessa linha é preciso especificar um nome para a função logo após a cláusula `def`. Em geral, as mesmas regras que se aplicam a nomeação de variáveis também se aplicam a nomeação de funções[1], isto é, é permitido o uso de caracteres alfanuméricos e `_` (*underline*), sendo que o primeiro caracter não pode ser numérico. Lembre-se de que espaços não são permitidos em nomes de funções ou variáveis! Após a definição do nome, são listados, em um par de parênteses nomes para os argumentos de entrada que a função recebe, separados por vírgula. Observe que cada função pode receber um número arbitrário de argumentos de entrada, e deve ser dado um nome diferente para cada um deles dentro desse par de parênteses na linha 1.

Cada função é possui um bloco de instruções que especifica o que deve ser feito a cada vez que a função for chamada a ser executada. Esse bloco de instruções é representado nas linhas 2-5. Seguindo o padrão da linguagem, cada linha desse bloco deve ser precedida por um caractere de tabulação (`<tab>`) ou número específico de espaços em branco para que seja possível determinar quais são as instruções que estão subordinadas à função. Assim, o interpretador Python saberá que o bloco de instruções vinculado a uma função se encerra imediatamente antes da linha que for prefixada por essa tabulação, que no Código 6.2 é representado pela linha 6. Podemos entender que o bloco de código que define uma função está de certo modo isolado do restante do código, o que significa que variáveis definidas dentro desse bloco não podem ser exergadas de fora dele.

A cláusula `def` apenas define uma função, especificando quais operações devem ser executadas a cada vez que a função for invocada. A cada vez que a função for invocada a executar suas operações, dizemos que houve um chamamento à função, ou que a função foi chamada, no sentido de que a função foi

[1] na realidade, Python trata funções como se fossem objetos. Nesse caso, o nome da função pode ser visto como uma variável que aponta para seu respectivo objeto função

chamada a executar seu procedimento. Uma função pode ainda retornar um valor para quem a chamou. Esse retorno deve ser especificado através de uma cláusula especial denominada `return`, ao qual ilustraremos nos exemplos a seguir. Além de retornar um valor, a cláusula `return` também encerra a execução de uma função.

Para ilustrar o uso de funções, vamos inicialmente definir uma função que calcula uma potencia, isto é, dado um número *base* e um outro número *expoente*, nossa função calcula o valor $base^{expoente}$ e o retorna. Chamaremos nossa função de (adivinhe só?!) `potencia`.

```
1 def potencia(base, expoente):
2   resultado = base ** expoente
3   return resultado
```

Código 6.3: função que calcula potenciação a partir de uma base e um expoente.

Ao analisar o Código 6.3, pode-se perceber que, na linha 1, declaramos uma função chamada *potencia*. Note que esta linha define ainda que esta função deve receber dois argumentos de entrada. O primeiro deles foi nomeado como *base*, ao passo que o segundo foi nomeado como *expoente*. Podemos entender `base` e `expoente` como sendo variáveis que representam os argumentos recebidos pela função. Podemos então fazer qualquer operação comum às variáveis usando `base` e `expoente`. Na linha 2, calculamos a potenciação esperada. Por fim, na linha 3, este resultado é retornado, isto é, é devolvido para quem realizar uma chamada à função.

A seguir, temos um exemplo de uso da função potencia. Após rodar o Código 6.3 através da IDLE, poderíamos chamar a função pelo prompt fazendo:

```
1 >>> valor = potencia(3, 2)
2 >>> valor
3 9
```

Note que, no exemplo anterior, definimos uma variável denominada `valor` que receberá o resultado retornado por `potencia(3, 2)`. A expressão `potencia(3, 2)` provocará uma chamada à função potência, o que fará com que o interpretador Python procure pelo trecho de código que a define para que então, este trecho seja executado. Assim, a execução de `potencia(3, 2)` acarretará na execução do Código 6.3 fazendo `base = 3` e `expoente = 2`. Observe que a posição declarada dos argumentos de entrada é utilizada para determinar qual argumento receberá qual valor. Como `base` foi declarada como sendo o primeiro argumento de `potencia`, esta variável é que receberá o primeiro valor passado a função dentro dos parênteses na chamada `potencia(3, 2)`, isto é, o valor 3. Por sua vez, como expoente foi declarado como sendo o segundo argumento de `potencia`, esta variável receberá o segundo valor passado dentro dos parênteses, isto é, o número 2. Assim, todo o Código 6.3 será executado com `base = 3` e `expoente = 2`. Desse modo, a variável `resultado` apontará para o valor `9` (3^2) na linha 2, e este valor será retornado para quem chamou a função na linha 3. Assim, a variável `valor` receberá o número `9`. Note que, inicialmente, o número de valores passados à função deve casar com o número de argumentos que a recebe em sua definição. Se um número superior ou inferior de argumentos for passado á função, resultará em erro.

6.2 Escopo de função

É importante compreender que, no Código 6.3, as variáveis `base`, `expoente` e `resultado` existem apenas no escopo (contexto) da função `potencia`. Isto significa que estas variáveis existem apenas enquanto a função `potencia` estiver sendo executada, e que essas variáveis não podem ser acessadas por uma linha de código que esteja fora da função `potencia`. Uma vez que o código que define uma função está, de certo modo, isolado do restante, e como se essas três variáveis fossem exclusivas da função `potencia` não podendo, de modo algum, serem acessadas de fora desta função. Assim, tentar imprimir o valor de `resultado` a partir do prompt resultará em erro, pois olhando do ponto de vista do prompt, é como se não existisse essa variável (só pode ser enxergada dentro da função potencia), conforme o exemplo a seguir:

```
>>> valor = potencia(3, 2)
>>> valor
9
>>> print(resultado)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(resultado)
NameError: name 'resultado' is not defined
```

É oportuno frisar aqui que, como as variáveis definidas dentro de uma função são apenas enxergadas dentro da mesma, em princípio, o único modo de devolver um valor calculado dentro da função é através da cláusula `return`. É um erro comum, por parte de principiantes, o esquecimento dessa cláusula. Sem o uso de `return`, todo o valor calculado dentro do bloco da função poderá ficar inacessível. Para exemplificar, vamos criar uma nova função para potenciação onde propositalmente omitiremos a cláusula `return`, denominada `potencia2`:

```
1 def potencia2(base, expoente):
2   resultado = base ** expoente
```

Código 6.4: função que calcula potenciação sem a cláusula return (incompleta).

Ao chamarmos a função `potencia2`, o resultado de `base ** expoente` será calculado segundo a linha 2 do Código 6.4, no entanto, este valor será perdido ao final da execução da função, uma vez que o mesmo não é retornado. Por definição, quando uma função termina sua execução sem retornar qualquer valor, automaticamente Python a faz retornar o valor `None`. Assim, se tentarmos atribuir o resultado dessa função à uma variável, veremos que essa variável assumirá o valor `None`, conforme o seguinte exemplo:

```
>>> v = potencia2(3, 2)
>>> v
>>> v == None
True
```

É válido mencionar ainda que, cada chamada a uma função faz com que a mesma seja executada, de certo modo, isoladamente. Isso significa que cada execução de uma função é realizada em um escopo particular, que definirá suas próprias variáveis e valores calculados. Por exemplos, ao executarmos

```
>>> potencia(4, 0)
1
```

Será criado um contexto (escopo) de execução temporário para a função `potencia`, onde existirá uma variável `base` com o valor `4`, uma variável `expoente` com o valor `0` e uma variável `resultado` com o valor `1`. Essas variáveis existirão apenas enquanto a função `potencia` estiver sendo executada para essa chamada específica. Após a execução das função nessa chamada, o escopo criado para a execução será destruído, e assim, as variáveis do chamado escopo local de `potencia` (`base`, `expoente`, `resultado`) serão destruídas também, sobrevivendo apenas o objeto retornado pela função, que nesse caso, será o número `1` (observe que é o objeto retornado sobrevive, e não a variável dentro da função que aponta para ele). Se, por acaso, uma nova chamada à função for feita, por exemplo:

```
1 >>> potencia(5, 3)
2 125
```

Será criado um novo escopo local para execução de uma nova chamada á função `potencia`. Nesse novo escopo local, existirá uma outra variável chamada `base`, agora com o valor `5`, uma outra variável chamada `expoente`, agora com o valor `3` e uma outra variável com o valor `resultado`, agora com o valor `125`. Essas novas três variáveis serão destruídas quando a função `potencia` terminar sua execução para esta chamada, pois seu escopo de execução será destruído, sobrevivendo apenas o objeto retornado pela função (nesse caso, o valor `125`).

Podemos então imaginar que, a cada chamada a função `potencia`, será criado um novo escopo de execução, que conterá uma nova variável `base`, uma nova variável `expoente` e uma nova variável `resultado`. As variáveis criadas em um escopo particular não tem qualquer relação com as criadas em outro escopo de execução para outra chamada. Fazendo uma abstração um tanto bizarra, podemos pensar que, a cada vez que `potencia` for chamada, se abrirá um "universo paralelo" para sua execução. Esse universo paralelo conterá suas próprias variáveis `base`, `expoente` e `potencia` e será destruído quando a função terminar sua execução. Ao se chamar a função `potencia` novamente, um outro universo paralelo, totalmente diferente dos anteriores, será criado para a sua execução (e posteriormente destruído), e assim sucessivamente. A esse "universo paralelo" onde a função é executada, damos o nome de *escopo local*.

Para ilustrar o uso de uma função em um programa, o Código 6.5 fornece um programa onde usamos a função `potencia` para calcular potenciação de valores lidos do usuário.

```
1  def potencia(base, expoente):
2    resultado = base ** expoente
3    return resultado
4
5
6  b = float( input("Entre com uma base: ") )
7  exp = float( input("Entre com um expoente: ") )
8
9  pot = potencia(b, exp)
10 print("Resultado de ", b, "elevado a ", exp, ": ", pot)
```

Código 6.5: potenciação com base e expoente lidos do teclado

Nas linhas 1-3 do Código 6.5, temos a definição da função `potencia`. Ressaltamos, mais uma vez, que a clausula `def` apenas se destina a definição de uma função, sem efetivamente executar o bloco de código subordinado á mesma. Nas linhas 6 e 7 lemos do usuário valores para base e expoente, respectivamente, nas variáveis `b` e `exp`, para então passá-los à chamada à função `potencia` na linha

9. Por fim, o resultado é impresso na linha 10. Salientamos que poderíamos ter chamado a variável `b` de `base` e `exp` de `expoente` sem causar qualquer conflito com as variáveis `base` e `exp` definidas no escopo da função `potencia`. Pelo fato das funções executarem em um universo (escopo) diferente, o interpretador Python entenderia que a variável `base` definida dentro da função é diferente daquela definida fora desta, isto é, seriam tratadas como variáveis diferentes apesar de possuírem o mesmo nome.

Um exemplo de execução do Código 6.5 é fornecido a seguir:

```
Entre com uma base: 4
Entre com um expoente: 3
Resultado de  4.0 elevado a  3.0:  64.0
```

Um leitor mais crítico já deve estar se perguntando há algum tempo qual seria a utilidade de definir uma função para o cálculo de uma potenciação, no lugar de usar diretamente o operador `**`. A resposta é que de fato não qualquer utilidade prática na definição dessa nossa função `potencia`. Apenas o fizemos para tentar prover um exemplo didático de construção e uso de funções. Nas Seções seguintes, construiremos funções mais úteis.

6.3 Comentários adicionais sobre funções

É válido apontar que, no Código 6.5, fazemos leitura de dois valores para então passá-los a função `potencia`, que faz o cálculo desejado e retorna o resultado. Este resultado é então impresso pelo programa. Aproveitamos esse exemplo para ressaltar que, na grande maioria das vezes, funções devem se comunicar com o mundo exterior através do recebimento dos argumentos de entrada, e do retorno dos valores de saída. Em geral, funções não costumam ler dados com a função `input` ou imprimir com a função `print`, embora exceções a essa "regra" possam ocorrer. Ao deixar a função `potencia` receber os dados por argumento de entrada em vez da leitura direta do teclado, deixamos a função genérica o bastante para ser usada sempre que for preciso calcular a potenciação entre dois números, independentemente da forma como esses números foram obtidos, seja pelo teclado, arquivo, rede, ou ainda como resultados de outras contas. O mesmo vale quanto ao retorno do resultado. Muitos principiantes acreditam, por exemplo, que uma função deve imprimir o seu resultado final, mas está é uma escolha ruim. Idealmente, a função deve apenas retornar o seu resultado, deixando a decisão sobre o que fazer com ele para quem chamou a função. Em muitos casos, quando usamos uma função, não queremos que nada seja exibido na tela, e quando de fato quisermos a impressão, é de nossa responsabilidade faz?-la por conta própria com o resultado obtido, em vez de esperar que a função o faça, exatamente como foi feito no Código 6.5, onde usamos `input` e `print` fora da função `potencia` e fazemos a função se comunicar com o restante do programa através dos seus argumentos de entrada e retorno. É válido ressaltar todavia, que exceções podem ocorrer, e que algumas funções podem ser especificamente criadas visando entrada ou saída de dados, mas este não costuma ser o caso geral.

Pode-se aproveitar a definição da função `potencia` para fazer observações interessantes sobre as mesmas. Primeiramente, é possível chamar funções em Python passando os argumentos fora da ordem especificada em sua definição.

Para tal, é necessário especificar os nomes dos argumentos sendo passados, por exemplo:

```
>>> potencia(expoente = 5, base = 2)
32
```

Note que, no exemplo anterior, especificamos o expoente antes da base. Para isso, foi necessário saber o nome dado aos argumentos da função, pois o contrário, o interpretador entenderia que deveria calcular 5^2 no lugar de 2^5.

Pode-se definir também um valor padrão (*default*) para os argumentos de uma função. Por exemplo, podemos redefinir nossa função `potencia` como no Código 6.6:

```
1 def potencia(base, expoente = 2):
2   resultado = base ** expoente
3   return resultado
```

Código 6.6: função que calcula potenciação a partir de uma base e um expoente.

Observe que, na linha 1 do Código 6.6, há a atribuição `expoente = 2`. O que esta expressão faz é definir um valor padrão para o argumento `expoente` caso este não venha a ser passado em alguma chamada a função. Assim, se chamarmos a função `potencia` passando apenas um único argumento de entrada, o interpretador assumirá que esse argumento é a base e o que expoente é `2`. Se o expoente for passado á função, então o valor passado será utilizado normalmente:

```
>>> potencia(9)
81
```

```
>>> potencia(6)
36
```

```
>>> potencia(7, 4)
2401
```

Para os acostumados com outras linguagens de programação, é válido ressaltar que as funções em Python também não exigem a declaração de tipo dos argumentos de entrada nem do valor retornado. Essa característica é uma demonstração do que chamamos de *polimorfismo*, que se remete a capacidade de executar o mesmo código com diferentes tipos de dados. Observe que nossa função `potencia` funcionará com quaisquer argumentos de entrada que implementem o operador `**`. Assim, a mesma função funcionará com argumentos de entrada `int`, `float` `complex` sem a necessidade de redefinir a função para cada um desses tipos. Perceba ainda que o tipo do objeto retornado dependerá dos tipos dos argumentos de entrada, e que, caso algum dia surja uma nova classe de objetos que implemente o operador `**`, nossa função ainda funcionará com objetos dessa nova classe sem qualquer alteração em nosso código.

6.4 Mais exemplos

6.4.1 Exemplo: função para o cálculo do módulo de um número

O bloco de código que compõe o corpo de uma função pode conter qualquer instrução válida em Python. É possível também que esse bloco de instruções contenha um número arbitrário de cláusulas `return`, conforme ilustrado pelo Código 6.7, que define uma função para calcular o módulo de um número.

```
1 def modulo(numero):
2   if numero >= 0:
3     return numero
4   else:
5     return -numero
```

Código 6.7: função que calcula o módulo de um número.

Na linha 1 do Código 6.7, temos a declaração de uma função chamada `modulo` que recebe um argumento de entrada chamado `numero`. Na linha 2, realizamos um teste para avaliar se o número recebido é positivo ou zero. Caso afirmativo, retornamos o próprio valor da variável `numero` na linha 2. Caso contrário, retornamos o oposto do valor em `número` na linha 5. A função `modulo` tem a mesma finalidade da função `abs`, já nativa da linguagem, e apenas a escrevemos aqui para ilustrar conceitos didáticos.

É válido destacar que bloco de código que compõe o corpo de uma função pode possuir um número arbitrário de cláusulas `return`. Todavia, sempre que uma cláusula `return` for executada, ela provoca o encerramento imediato da execução da função, com o consequente retorno do objeto indicado. Desse modo, não faria muito sentido escrever uma função como a do Código 6.8:

```
1 def gera7( ):
2   return 7
3   print("Senta que la vem a historia!")
4   return 5
```

Código 6.8: função que retorna 7.

A função `gera7` do Código 6.8 possui duas peculiaridades. A primeira delas, é que ela não recebe argumentos de entrada. Todavia ainda é preciso usar um par de parênteses em sua declaração na linha 1. Observe que a linha 2 traz uma cláusula `return`, o que significa que, quando esta função for chamada, sua execução se encerrará logo após a execução da linha 2. Observe ainda que a função possui ainda outras duas linhas de código 3 e 4, mas essas duas linhas jamais serão alcançadas devido ao `return` na linha 2, que por si só já provoca o encerramento da execução da função.

6.4.2 Exemplo: função para o cálculo de arranjo

Vamos agora construir um exemplo mais funcional. Vamos construir uma função que que receba dois números n e p e calcule A_n^p, isto é, o arranjo simples de n elementos tomados p a p. O cálculo de A_n^p pode ser realizado segundo a equação 6.1:

$$A_n^p = \frac{n!}{(n-p)!} \tag{6.1}$$

Note que, para calcular A_n^p segundo a equação 6.1, será necessário calcular o fatorial de dois números. Por essa razão, no Código 6.9, além da função para o cálculo do arranjo, escreveremos uma função auxiliar para o calculo do fatorial.

```
1 def fatorial(num):
2   fat = 1
3   for k in range(num, 0, -1):
4     fat = fat * k
5   return fat
6
7 def arranjo(n, p):
8   a = fatorial(n)//fatorial(n-p)
9   return a
```

Código 6.9: funções para o cálculo de fatorial e de arranjo.

No Código 6.9, as linhas 1-5 definem a função `fatorial` para o cálculo do fatorial de um número. As linhas 7-9 definem a função `arranjo`, que recebe n e p e retorna o valor de A_n^p. Observe que, na linha 8, são feitas duas chamadas à função `fatorial` para os cálculos de $n!$ e $(n-p)!$. Apesar da função `arranjo` fazer uso da função `fatorial`, a linguagem Python não exige que `fatorial` seja declarada antes de `arranjo`. Graças a sua dinâmica de tipagem e execução, Python apenas exige que a função `fatorial` já esteja declarada somente no momento em que `arranjo` for efetivamente executada pela primeira vez, o que significa que estas duas funções poderiam estar declaradas em qualquer ordem.

Uma peculiaridade da linguagem Python é que funções são tratadas como objetos. Ao declarar a função `arranjo`, por exemplo, é como se tivéssemos criado uma variável chamada `arranjo` que aponta para um objeto função. Poderíamos, por exemplo, fazer essa variável `arranjo` passar a apontar para um outro objeto qualquer, inclusive declarar outra função com esse nome para "sobrescrever" a primeira. Esta particularidade também permite que uma função possa ser declarada dentro de outra. Por exemplo, poderíamos declarar a função fatorial dentro da função arranjo, conforme o Código 6.10.

```
1 def arranjo(n, p):
2   def fatorial(num):
3     fat = 1
4     for k in range(num, 0, -1):
5       fat = fat * k
6     return fat
7
8   a = fatorial(n)//fatorial(n-p)
9   return a
```

Código 6.10: função encapsulada para o cálculo de fatorial e de arranjo.

Note que ao declarar a função `fatorial` dentro da função `arranjo`, devido as regras de escopo local, a função `fatorial` só poderá ser chamada de dentro da função `arranjo`. Essa técnica é denominada *encapsulamento* e visa "esconder" procedimentos auxiliares (no caso em questão, a função `fatorial`) ao só tornar disponível o chamamento do procedimento principal (no nosso exemplo, a função `arranjo`). Note que, desse modo, a cada execução de `arranjo`, uma nova função `fatorial` será criada em seu escopo local, o que acaba sendo menos eficiente do que a versão não encapsulada no Código 6.9.

6.4.3 Exemplos: funções sobre objetos sequenciais

Para ilustrar que não há mistério na elaboração com funções, o Código 6.11 traz a definição da função `contaMinusculos` que recebe uma string e retorna a quantidade de caracteres alfabéticos minúsculos presentes na mesma:

```
1 def contaMinusculos(texto):
2     contador = 0
3     for c in texto:
4         if c.islower() == True:
5             contador += 1
6
7     return contador
```

Código 6.11: função que conta a quantidade de caracteres minúsculos presentes em uma string.

Após rodar o Código 6.11 na IDLE, poderíamos chamar a função passando-lhe uma string qualquer, por exemplo:

```
1 >>> contaMinusculos("Lua De cRIStAL")
2 5
```

```
1 >>> t = "festa DO EstICA e PUXa"
2 >>> contaMinusculos(t)
3 9
```

6.5 Variáveis locais e globais

Na Seção 6.2, vimos que variáveis criadas dentro do bloco de código subordinado à uma função não podem ser acessadas de fora do bloco. A cada chamada à uma função, um novo escopo local de execução é gerado, e é nesse escopo local que as variáveis utilizadas pela função são mantidas até o final da execução do código da função, quando o respectivo escopo local é destruído. Embora variáveis criadas dentro do contexto de uma função não possam ser acessadas de fora desta, a recíproca não é verdadeira. Isso significa que, de dentro de uma função, é possível acessar uma variável criada de fora desta, conforme ilustrado pelo Código 6.12.

```
1 w = 5
2 def fun1():
3     return w
4
5 v = fun1()
6 print( v )
```

Código 6.12: função que acessa variável criada no módulo envolvente.

Observe que, no Código 6.12, temos a definição da função `fun1`, a qual utiliza, na linha 3 a variável `w`, que foi definida fora da função, na linha 1. Este programa é executado sem qualquer erro pelo interpretador Python. Durante a execução da linha 3, ao não encontrar variável `w` definida no escopo de `fun1`, o interpretador procurará por variável que tenha esse nome fora do escopo de `fun1`, encontrando assim a variável definida na linha 1 e corretamente imprimindo o valor `5` na linha 6. Pelo fato de `w` ter sido definido fora da função, dizemos que `w` é uma *variável de escopo global*, ou simplesmente *variável global*, dentro de `fun1`.

É interessante reforçar, entretanto, que, as variáveis do escopo local têm preferência na busca por variável na execução de uma função. Veja, por exemplo, o Código 6.13:

```
1 w = 5
2 def fun2():
3     w = 7       #define uma npva variável w em escopo local
4     return w
5
6 v = fun2()
7 print("fun2 retornou: ", v)
8 print("w vale: ", w)
```

Código 6.13: função que acessa variável criada no escopo local.

Observe que, na linha 1 do Código 6.13, é definida uma variável de nome `w`. Dentro da função `fun2`, temos, na linha 3, a atribuição `w = 7`. No entanto, pelo fato dessa atribuição ser feita dentro do bloco subordinado à função `fun2`, o interpretador Python criará uma nova variável `w` dentro do escopo local de `fun2`. Em outras palavras, é como se existisse, simultaneamente, duas variáveis distintas de nome `w` no Código 6.13. A primeira delas, criada na linha 1, em escopo global, e a segunda, criada na linha 3, no escopo local de `fun2`. Desse modo, a execução do Código 6.13 terá como saída:

```
fun2 retornou:  7
w vale:  5
```

Note que, na execução da linha 4, pelo fato de estar vinculada à `fun2`, será buscada `w` no escopo local de `fun2`. Na execução da linha 8, que está fora de qualquer função, será buscada `w` no escopo global. Por essa razão, `fun2` retornará `7`, ao passo que a variável `w` em escopo global continuará valendo `5`.

Podemos então enunciar que a seguinte ordem utilizada pelo interpretador Python na busca por algum nome:

1. Busca-se primeiro no escopo local;
2. Se o nome não for encontrado, busca-se nos escopos locais das funções envolventes;
3. Se o nome não for encontrado, busca-se no escopo do módulo (arquivo) envolvente (escopo global);
4. Se o nome ainda não for encontrado, busca-se no módulo de nomes internos do Python, denominado `__builtin__`. Neste módulo, nomes pré-reservados como `str`, `for` e `if` são pre-definidos.

Pelo fato do escopo local ter prioridade sobre o global, podemos forçar uma função a trabalhar com nome no escopo global através da cláusula `global`, conforme exemplificado no Código 6.14:

```
1 w = 5
2 def fun3():
3     global w    #obriga a função a trabalhar com w apenas em
          escopo global.
4     w = 9
5     return w
6
7 v = fun3()
8 print("fun3 retornou: ", v)
9 print("w vale: ", w)
```

Código 6.14: função que força acesso a variável em escopo global.

Note que, na linha 3 do Código 6.14, a expressão `global w` obriga que a função `fun3` trabalhe com a variável `w` apenas em escopo global. Desse modo, a atribuição `w = 9`, realizada na linha 4, aletrará a variável `w` criada em escopo global na linha 1. Por essa razão, a execução deste programa gerará a seguinte saída:

```
fun3 retornou:  9
w vale:  9
```

É de extrema importância destacar que variáveis globais não são bem vistas no mundo da programação. As boas práticas dizem que funções devem trabalhar o máximo possível apenas em escopo local, de modo a interferir o mínimo possível no escopo global. Se for preciso que uma função utiliza algum valor definido em escopo global, o ideal é que este seja recebido como argumento de entrada dessa função. Essa regra simples simplifica o entendimento, a manutenção e a correção de erros nos programas. Por essa razão, devemos evitar ao máximo o uso de variáveis globais e procurar sempre declarar os valores necessários ao trabalho de uma função como argumento de entrada. Como sempre, no mundo real, podem haver exceções a regra, mas lembre-se de que são apenas **exceções**!

6.6 Recursão

Dizemos que uma entidade é *recursiva* quando é possível defini-la a partir de si própria. Um exemplo clássico de recursão é a definição do fatorial de um número natural n, denotado por $n!$. Além da definição clássica:

$$n! = n(n-1)(n-2)\dots 1$$

, podemos também definir $n!$ de forma recursiva:

$$n! = \begin{cases} 1, & \text{se } n = 0, \\ n(n-1)!, & \text{se } n > 0. \end{cases} \tag{6.2}$$

Observe que a Equação (6.2) define o fatorial de um número natural de forma completa. Note ainda que $n!$ é definido em função de $(n-1)!$, isto é, o cálculo do fatorial de um número depende do cálculo do fatorial de outro. Pelo fato de definir o fatorial usando o próprio fatorial, dizemos que a definição na Equação (6.2) é recursiva. A expressão $n! = n(n-1)!$ é denominada como *relação de recorrência*. Note que, para que uma definição recursiva seja completa, é necessária a definição de um ou mais *casos base*, que são casos definidos sem o uso da relação de recorrência. No nosso exemplo, o caso base foi definido para o fatorial de zero. Em algumas situações, pode ser necessária a definição de mais de um caso base para que recursão fique bem definida. O número de casos base necessários se remete ao quanto uma relação de recorrência "observa" termos anteriores de modo recursivo. Por exemplo, na Equação (6.2), a relação de recorrência observa apenas o termo imediatamente anterior. Por essa razão apenas um caso base é necessário. No entanto, poderíamos ter definido a relação de recorrência como:

$$n! = n(n-1)(n-2)!$$

Nesse caso, para o cálculo de $n!$, o termo mais "anterior" observado envolvendo fatorial é $(n-2)!$, o significa que são necessários dois casos base para essa relação de recorrência.

No contexto computacional, a recursividade se remete basicamente a algum tipo de função ou procedimento que possui a habilidade de chamar a sí próprio em seu algoritmo. Por exemplo, poderíamos definir uma função em Python para o cálculo do fatorial usando a definição recursiva (6.2)

```
1 def fatorial(n):
2   if n == 0:
3     return 1
4   elif n > 0:
5     r = n*fatorial(n-1)
6     return r
```

Código 6.15: função recursiva para cálculo do fatorial de um número.

A função definida no Código 6.15 é capaz de calcular corretamente o fatorial de um número, conforme os exemplos a seguir:

```
1 >>> fatorial(0)
2 1
```

```
1 >>> fatorial(4)
2 24
```

A figura 6.1 ilustra o cálculo de 4! por meio de nossa função `fatorial`. Observe que, para a obtenção de 4!, nossa função `fatorial` precisa calcular 3!, o que, por sua, vez, demanda o cálculo de 2!, o que também exige o cálculo de 1!, que, por fim, necessitou do cálculo de 0!. Até o cálculo da base da recursão em 0!, uma pilha de chamadas a função `fatorial` foi deixada em aberto para os cálculos de 4!, 3!, 2! e 1!. Após o cálculo de 0!, foi então possível resolver o problema de calcular 1!, o que, por sua vez, possibilitou o cálculo de 2!, que permitiu então obter 3!. Por fim, o valor de 4! acabou sendo calculado e retornado, encerrando assim a pilha de chamadas a função `fatorial`.

$$
\begin{aligned}
4! = 4 \times \underbrace{3!}& \\
3 \times \underbrace{2!}& \\
2 \times \underbrace{1!}& \\
1 \times \underbrace{0!}& \\
& 1
\end{aligned}
$$

Figura 6.1: cálculo recursivo de 4!.

Um outro exemplo clássico de recursão se remete aos famosos *números de Fibonacci*, os quais formam seguinte sequência:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$$

O i-ésimo número da sequência de Fibonacci, denotado por f_i, pode ser definido como:

$$
f_i = \begin{cases} 0, & \text{se } i = 1, \\ 1, & \text{se } i = 2, \\ f_{i-1} + f_{i-2}, & \text{se } i > 2. \end{cases} \tag{6.3}
$$

Pela Equação (6.3), podemos perceber que o primeiro número f_1 da sequência de Fibonacci é 0 e o segundo número f_2 é 1. A partir daí, a relação de

recorrência diz que cada termo f_i é dado pela soma dos dois termos imediatamente anteriores, isto é $f_{i-1} + f_{i-2}$. Observe que temos aqui uma recursão de ordem 2, pois o termo mais "anterior" observado pela relação de recorrência está duas unidades a frente. O Código 6.16 traz uma função Python que retorna o i-ésimo termo da sequência de Fibonacci. Note que nesta código, colocamos os resultados imediatamente após a cláusula `return` sem o uso de variável intermediária.

```
1 def fibonacci(i):
2   if i == 1:
3     return 0
4   elif i == 2:
5     return 1
6   elif i > 2:
7     return (fibonacci(i-1) + fibonacci(i-2))
```

Código 6.16: função recursiva para cálculo do fatorial de um número.

Exemplos de uso da função `fibonacci` são exibidos a seguir:

```
1 >>> fibonacci(1)
2 0
```

```
1 >>> fibonacci(2)
2 1
```

```
1 >>> fibonacci(4)
2 2
```

```
1 >>> fibonacci(5)
2 3
```

Em geral, procedimentos recursivos têm a vantagem de serem considerados mais simples que os seus equivalentes não recursivos. Muitas pessoas, por exemplo, considerariam a função `fatorial` recursiva (Código 6.15) mais simples de entender e implementar do que a versão não recursiva (Código 6.9). O mesmo vale para a função `fibonacci` (leitores estão desafiados a construir uma versão não recursiva da função `fibonacci`). Uma desvantagem da recursão se remete a eficiência. Em geral, os procedimentos não recursivos são mais eficientes que seus equivalentes recursivos. Isso se dá, primeiramente, porque a execução de funções recursivas envolvem uma série de chamadas a própria função. Para cada uma dessas chamadas, é necessária a realização de alguns procedimentos, como, por exemplo, a criação de um novo escopo local, o que demanda algumas operações extras. Ademais, a perda de eficiência se torna mais crítica quando a relação de recorrência possui ordem maior que 1. Observe que, para calcular f_i, a função `fibonacci` precisa calcular f_{i-1} e f_{i-2}. Por precisar de outros dois termos de Fibonacci para cada termo diferente dos casos base, pode haver uma grande repetição de trabalho. Por exemplo, para o cálculo de f_6, vamos analisar o comportamento quanto aos termos calculados segundo uma árvore, exibida na Figura 6.2:

Note que a obtenção de f_6 exigirá o cálculo de f_5 e f_4. No entanto, ao computar f_5, será necessário calcular novamente f_4. Desse modo, o termo f_4 será calculado duas vezes! Estendendo o raciocínio, podemos perceber que o termo f_3 será calculado 3 vezes, ao passo que f_2 será calculado 5 vezes e f_1 será calculado 3 vezes. Toda essa repetição de cálculo de termos gera um considerável desperdício de esforço computacional em comparação com o cálculo não recursivo dos termos por meio de um laço de repetição, por exemplo. Essa repetição desnecessária se torna mais crítica quanto maior for o índice do termo calculado. Todavia, é

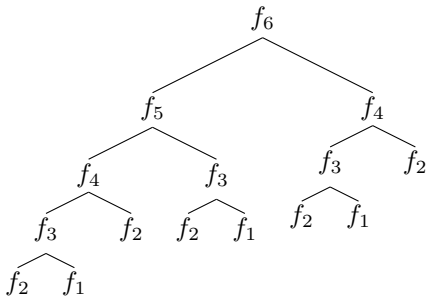

Figura 6.2: árvore de cálculo recursivo do sexto termo da sequência de Fibonacci (f_6).

possível utilizar técnicas para a redução ou até eliminação desse *overhead*, com a contrapartida de aumentar a complexidade da lógica do procedimento.

6.7 Funções com número arbitrário de argumentos de entrada

6.7.1 Por Tupla

É possível construir funções que possam recebam um número arbitrário de argumentos de entrada. Por exemplo, vamos supor que desejamos construir uma função `soma` que receba uma quantidade qualquer de número e retorne o somatório dos mesmos, conforme os seguintes exemplos:

```
>>> soma(7)
7
```

```
>>> soma(2, 8)
10
```

```
>>> soma(1, 3, 2)
6
```

```
>>> soma(5,4,1,9)
19
```

Para construir nossa função `soma`, teremos que declarar um argumento de entrada com prefixado pelo operador `*`, conforme o Código 6.17:

```
def soma( *parcelas ):
  s = 0
  for p in parcelas:
    s = s + p
  return s
```

Código 6.17: função que calcula soma de um numero arbitrário de argumentos.

Note que, na declaração da função `soma`, especificamos um único argumento de entrada, que está precedido por `*`. Este operador faz com que a função possa receber um número qualquer de argumentos de entrada (inclusive nenhum). Todos os argumentos passados em uma determinada chamada à função serão colocados em uma tupla, a qual será atribuída à variável `parcela`. Assim, podemos encarar `parcela` como sendo uma tupla que contém todos os argumentos de entrada passados à função.

Pode-se também declarar argumentos ordinários juntamente com o argumento que receberá o *, desde que este último venha após todos os ordinários, como pode ser verificado no Código 6.18:

```
1 def funcao(arg1, arg2, *tupla):
2   print("arg1: ", arg1)
3   print("arg2: ", arg2)
4   print("tupla: ", tupla)
```

Código 6.18: função que recebe argumentos ordinários e um argumento tupla.

No código 6.18, a função `funcao` foi declarada com dois argumentos ordinários `arg1` e `arg2`, seguido do argumento `tupla`, que pode encaixotar um número arbitrário de argumentos em uma tupla. Isso significa que esta função deve receber ao menos dois argumentos, de modo a obrigatoriamente preencher `arg1` e `arg2`. Todos os argumentos passados a partir da terceira posição serão acessados por meio da tupla apontada pela variável `tupla`, conforme os exemplos:

```
1 >>> funcao(True, 7)
2 arg1:  True
3 arg2:  7
4 tupla:  ()
```

```
1 >>> funcao("Mara", None, 0)
2 arg1:  Mara
3 arg2:  None
4 tupla:  (0,)
```

```
1 >>> funcao(False, 2, 8,9)
2 arg1:  False
3 arg2:  2
4 tupla:  (8, 9)
```

```
1 >>> funcao("ana",3,1,"oi",2)
2 arg1:  ana
3 arg2:  3
4 tupla:  (1, 'oi', 2)
```

Ok, o leitor mais atento já deve estar questionando sobre o fato de que havíamos dito que, em geral, funções não devem imprimir valores. No entanto, fizemos essas impressões em `funcao` apenas para ilustrar o funcionamento da passagem de argumentos de entrada para nossa função, com a devida "licensa poética". Abordaremos ainda o funcionamento de tuplas no Capítulo 7.

6.7.2 Por Dicionario

De modo similar ao recebimento de um número arbitrário de argumentos de entrada por tupla, também é possível utilizar dicionários para cumprir tal tarefa. Nessa situação, quem fizer a chamada à função deverá nomear cada valor passado como argumento de entrada.

O Código 6.19 traz a definição da função `funcaoDic`, que recebe um número arbitrário de argumentos nomeados por dicionário. Note o uso do operador ** antes do argumento de entrada.

```
1 def funcaoDic(**args):
2   print('Dicionario de argumentos: ', args)
```

Código 6.19: função que recebe número variável de argumentos nomeados em um dicionário.

A seguir, um exemplo de uso da função `funcaoDic`. Reiteramos que, uma vez que o argumento `args` dessa função foi prefixado por **, é mandatório a atribuição de um nome diferente para cada valor passado à função. Após rodar o Código 6.19 na IDLE, podemos escrever, no *prompt*:

```
>>> funcaoDic( nome="jessica", idade=27, mulher=True )
```

A seguir, fornecemos o resultado da chamada anterior:

```
>>> funcaoDic( nome="jessica", idade=27, mulher=True )
Dicionario de argumentos: {'mulher': True, 'nome': 'jessica', '
    idade': 27}
```

Observe, por exemplo, que o valor `27` através do nome `idade`. Este nome atribuído ao valor será convertido em string para ser armazenado como chave no dicionário apontado por `args`. Algo similar ocorre com os demais valores passados à função. Se você não está entendendo o dicionário de argumentos impresso no exemplo, não se desespere! Discutiremos o funcionamento dos dicionários no Capítulo **??**.

Também é possível declarar uma função que receba argumentos ordinários juntamente com um argumento prefixado com `*` e um prefixado por `**`, desde que todos os argumentos ordinários venham antes do argumento com `*`, e o argumento com `*` venha antes do argumento com `**`, conforme o exemplo a seguir:

```
def funcaoTotal(arg1, arg2, arg3, *arg4, **arg5):
  return None
```

Código 6.20: função que recebe argumentos ordinários (*arg1*, *arg2* e *arg3*), um número variável de argumentos em tupla (*arg4*) e um número variável de argumentos nomeados em um dicionário (*arg5*).

6.8 Exercícios

1. Escreva um programa que leia um ângulo em graus e informe o respectivo valor do ângulo em radianos. Seu programa deve implementar uma função para realizar a essa conversão.

   Exemplos:

   ```
   Entre com o valor do angulo em graus: 45
   Ángulo 45.0 em radianos: 1.570796
   ```

   ```
   Entre com o valor do angulo em graus: 60
   Ángulo 60.0 em radianos: 2.094395
   ```

   ```
   Entre com o valor do angulo em graus: 90
   Ángulo 90.0 em radianos: 3.141593
   ```

2. Escreva um programa que leia um número n e imprima o valor do somatório dos primeiros n números positivos. Apenas para exercitar, faça em seu programa uma função que calcule esse somatório de modo **recursivo** e **retorne** o resultado.

   Exemplos:

   ```
   Entre com o valor de n: 3
   Soma dos 3 primeiros numeros inteiros positivos: 6
   ```

```
Entre com o valor de n: 7
Soma dos 7 primeiros numeros inteiros positivos: 28
```

3. Faça um programa que leia um número i e imprima o i-ésimo número da sequência de Fibonacci. Seu programa deve contemplar uma função específica **sem o uso de recursão** que receba o número i e **retorne** o valor esperado.

   Exemplos:

```
Entre com o valor de i: 6
6-esimo termo na sequencia de fibonacci: 5
```

```
Entre com o valor de i: 10
10-esimo termo na sequencia de fibonacci: 34
```

4. Faça uma função que leia um número positivo do teclado, de modo similar à função `input`. A diferença é que, caso o usuário não forneça um número positivo, sua função deve repetir a leitura até que um número positivo seja lido. Note que, o código da função poderá usar a função `input` e deverá **retornar** o número lido. Junto com a função, faça um programa que chame a função 3 vezes para a leitura de 3 números distintos. Aproveite para fazer seu programa imprimir os números em ordem decrescente.

   Exemplos:

```
Entre com um numero inteiro positivo: 0
Numero invalido!
Entre com um numero inteiro positivo: -4
Numero invalido!
Entre com um numero inteiro positivo: 9
Entre com um numero inteiro positivo: 14
Entre com um numero inteiro positivo: 5
Numeros em ordem decrescente: 14 9 5
```

```
Entre com um numero inteiro positivo: 55
Entre com um numero inteiro positivo: 20
Entre com um numero inteiro positivo: 38
Numeros em ordem decrescente: 55 38 20
```

5. Escreva um programa que calcule o seno de um ângulo em radianos pela série de Taylor com aproximação de ordem p em torno de $\hat{x} = 0$ segundo a fórmula:

$$\sin x = \sum_{k=0}^{p} (-1)^k \frac{x^{2k+1}}{(2k+1)!} \tag{6.4}$$

   Seu programa deve possuir uma função chamada `fatorial` para o cálculo do fatorial. Para o cálculo do seno, seu programa deverá contemplar uma função chamada `seno` que recebe x e p e retorna a aproximação do seno segundo a Equação 6.4.

   Exemplos:

```
Entre com o angulo em radianos: 3.1415
Entre com o valor de p: 10
Seno de 3.1415: 0.000093
```

```
Entre com o angulo em radianos: 0.95
Entre com o valor de p: 8
Seno de 0.95: 0.813416
```

6. O famoso Triângulo de Pascal pode ser definido recursivamente da seguinte forma

   (a) Cada linha i tem uma quantidade i de termos numéricos (colunas)

   (b) O primeiro e o último termo em cada linha são 1's

   (c) Os demais termos são definidos como a soma do termo imediatamente acima e do antecessor do termo de cima.

$$
\begin{array}{ccccccc}
1 & & & & & & \\
1 & 1 & & & & & \\
1 & 2 & 1 & & & & \\
1 & 3 & 3 & 1 & & & \\
1 & 4 & 6 & 4 & 1 & & \\
1 & 5 & 10 & 10 & 5 & 1 & \\
1 & 6 & 15 & 20 & 15 & 6 & 1
\end{array}
$$

Figura 6.3: exemplo de Triângulo de Pascal

Faça uma função **recursiva** que calcule um determinado termo do triângulo de Pascal. Sua função deve receber a linha e a coluna onde o termo se encontra e retornar o respectivo termo. Faça também um programa que leia a linha e a coluna do teclado e chame a função.

Exemplos:

```
Entre com a linha do termo: 2
Entre com a coluna do termo: 1
Termo na linha 2, coluna 1: 1
```

```
Entre com a linha do termo: 3
Entre com a coluna do termo: 3
Termo na linha 3, coluna 3: 1
```

```
Entre com a linha do termo: 6
Entre com a coluna do termo: 4
Termo na linha 6, coluna 4: 10
```

```
Entre com a linha do termo: 20
Entre com a coluna do termo: 19
Termo na linha 20, coluna 19: 19
```

Capítulo 7

Listas e Tuplas

Em Python, Listas são sequências de objetos arbitrários sob uma ordem determinada. Podemos encará-las como um vetor ou *array* de objetos conforme linguagens como C ou Fortran. Todavia, constataremos que as listas em Python possuem muito mais maleabilidade do que os *arrays* das linguagens clássicas. Primeiramente, as listas em Python podem englobar, simultaneamente, objetos de diferentes tipos, conforme o exemplo:

```
1 >>> lista = [28, 3, "leidiana", 2.718, True]
2 >>> lista
3 [28, 3, 'leidiana', 2.718, True]
```

Observe que listas são declaradas usando colchetes, com seus elementos separados por vírgula. Uma lista pode conter qualquer tipo de objeto em seu interior, inclusive tuplas ou outras listas:

```
1 >>> L = [ 9,  ["lenir", "vanda"], (2,3,5,7) ]
2 >>> L
3 [9, ['lenir', 'vanda'], (2, 3, 5, 7)]
```

Em Python, é comum usar o termo *contêiner* para designar objetos que podem "conter" outros objetos em seu interior. Desse modo, listas são consideradas como objetos *contêiners*. Pelo fato de serem contêiners que adotam uma ordem para seus elementos, podemos realizar operações de indexação e fracionamento sobre listas, conforme apresentado na Seção 5.3:

```
1 >>> lista[2]
2 'leidiana'
3 >>> lista[1:4]
4 [3, 'leidiana', 2.718]
5 >>> L[1][0]
6 'lenir'
```

Código 7.1: indexação e fracionamento com listas.

Note que, no exemplo 7.1, na linha 5 temos a dupla indexação `L[1][0]`. Essa operação é possível porque, no índice 1 da lista apontada por `L`, temos a lista `["lenir", "vanda"]`. Pelo fato de haver outra lista no índice 1, é possível acessar também os índices dessa última. Assim, a operação `L[1][0]` resgatará o elemento que está no índice 0 da lista que está presente no índice 1 de L, que, no caso, é a string `"lenir"`. Poderíamos ainda usar mais uma operação de indexação para acessar algum índice dessa string, por exemplo fazendo:

```
>>> L[1][0][3]
'i'
```

Listas são objetos mutáveis, isto é, podem ser alterados na memória. Para compreender melhor esse processo, considere a seguinte atribuição

```
>>> lista1 = [11, 13, 17]
```

A partir da atribuição anterior, teremos o seguinte esquema na memória representado pela Figura 7.1. Note que, no espaço de objetos, será gerado um objeto lista que pode compreender três objetos. De modo a facilitar a compreensão, podemos considerar que, em cada índice da lista temos uma "variável" (apontador) que pode apontar para qualquer objeto. Representamos essas "pseudo-variáveis" ou apontadores como `@ap0` (apontador do índice 0), `@ap1` (apontador do índice 1) e `@ap2` (apontador do índice 2). Cada um desses apontará para o respectivo objetivo colocado em seu índice na lista.

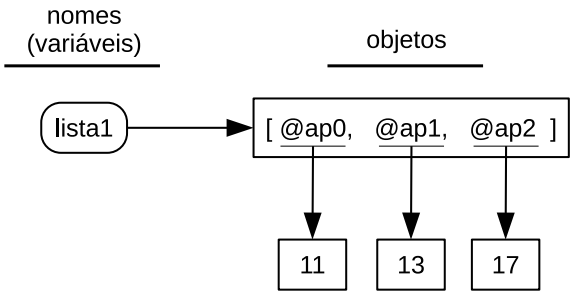

Figura 7.1: estado corrente da memória após a atribuição *lista1 = [11, 13, 17]*.

Podemos então alterar a lista gerada, por exemplo fazendo a atribuição:

```
>>> lista1[2] = -1
```

Interpretamos o comando `lista1[2] = -1` como sendo atribua o valor -1 ao índice `2` de `lista1`. Desse modo, ao ecoarmos o valor apontado por `lista1`, teremos:

```
>>> lista1
[11, 13, -1]
```

A Figura 7.2 exibe o esquema de variáveis e objetos na memória após a última atribuição. Note que o apontador do índice 2 de `lista1` agora aponta para o objeto `int` com o valor `-1`. Por sua vez, o objeto `int` com valor `17` é removido da memória, uma vez que este não mais é apontado.

É possível também realizar atribuições sobre fatias de uma lista:

```
1 >>> lista2 = [2, 4, 6]
2 >>> lista2[1: ] = [3, 5, 7, 11, 13]
3 >>> lista2
4 [2, 3, 5, 7, 11, 13]
```

Código 7.2: atribuição sobre fracionamento.

Observe que, na linha 1 do Código 7.2 é definida uma lista com três elementos. Na linha 2, realizamos uma atribuição sobre a fatia da lista que vai

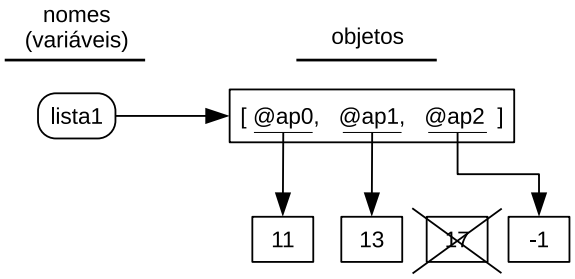

Figura 7.2: estado corrente da memória após a atribuição *lista1[2] = -1*.

do índice `1` até o seu final. Podemos pensar nessa operação como sendo uma espécie de substituição de uma sublista por outra lista. Repare que a sublista sendo substituída possui apenas `2` elementos, ao passo que a nova lista sendo atribuída possui `5`. Todavia, isso não é problema para uma linguagem de altíssimo nível como Python que aumentará o número de elementos da lista após a operação na linha 1, conforme pode ser verificado na linha 2. É possível ainda realizar atribuição sobre fatias não contíguas de uma lista, como no exemplo a seguir:

```
1 >>> lista3 = [1, 3, 5, 7, 9, 11]
2 >>> lista3[0::2] = [500, 600, 800]
3 >>> lista3
4 [500, 3, 600, 7, 800, 11]
```

Para o caso de fatias não contíguas, como no exemplo anterior, a lista sendo atribuída deve possuir o mesmo tamanho da sublista sendo substituída. Se você ficou confuso sobre esse último código, pode ser esclarecedor relembrar como funciona a operação de fatiamento consultando a Seção 5.3.

7.1 Operações com listas

Os seguintes operadores podem ser utilizados no contexto de listas em Python:

- **Comprimento: `len`**

  O operador `len` retorna o comprimento de objetos sequenciais em geral. No caso de listas, o comprimento é dado pelo número de objetos em seu interior.

  ```
  1 >>> dados = [True, 2, None, 1+6j]
  2 >>> len(dados)
  3 4
  ```

- **Concatenação: `+`**

  `+` concatena objetos sequencias em geral . Assim, este operador é capaz de gerar uma nova lista a partir de outras duas:

  ```
  1 >>> v = [1, 3, 5] + ["oi", "tchau"]
  2 >>> v
  3 [1, 3, 5, 'oi', 'tchau']
  ```

- **Construção - conversão para lista:** `list`

  `list` funciona como construtor da classe, isto é, é capaz de gerar listas a partir de outros objetos sequenciais.

```
>>> idades = (3, 7, 10, 17)
>>> list(idades)
[3, 7, 10, 17]
```

- **Lista varia:** `[]`

  Com um par de colchetes, é possível declarar uma lista vazia, isto é, uma lista sem nenhum elemento

```
>>> bens = []
```

- **Participação como membro:** `in` e `not in`

  `in` retorna `True` se um objeto aparece em uma sequência e `False` caso contrário. Assim, podemos utilizar este operador para saber se um objeto aparece em uma lista:

```
>>> garotas = ['larissa', 'isabela', 'tatiana']
>>> 'bruna' in garotas
False
>>> 'isabela' in garotas
True
```

  Por sua vez, o operador `not in` fornece o resultado oposto ao de `in`, isto é, retorna `True` se o objeto à esquerda não aparecer dentro da sequência à direita e `False` caso contrário.

- **Remoção de elementos:** `del`

  O operador `del` pode ser utilizado para a remoção de elementos de objetos sequenciais mutáveis, como listas:

```
>>> nums = [ 10, 20, 40, 80 ]
>>> del nums[2]
>>> nums
[10, 20, 80]
>>> del nums[1:]
>>> nums
[10]
```

  Note que é preciso especificar índices ou fatias do(s) elemento(s) sendo removido(s) da lista.

- **Repetição:** `*`

  O operador `*` pode ser utilizado para a repetição de sequências. Assim, o mesmo pode gerar uma nova lista a partir da repetição dos elementos de outra:

```
>>> vals = [3, "agosto"]
>>> vals * 4
[3, 'agosto', 3, 'agosto', 3, 'agosto', 3, 'agosto']
```

  Observe que, com esse operador, pode-se rapidamente gerar uma lista com 100 elementos `0` fazendo `[0]*100`.

- **Desempacotador de elementos: `*`**

  A partir da versão 3.5, o operador `*` também pode ser utilizado para "desempacotar" elementos de um objeto *contêiner* em Python:

```
1 >>> primos = [1, 3, 5]
2 >>> numeros = [0, *primos, 10]
3 >>> numeros
4 [0, 1, 3, 5, 10]
```

Código 7.3: desempacotamento de elementos de uma lista.

No Código 7.3, a linha 1 faz a variável `primos` apontar para uma lista de 3 elementos. A seguir, a linha 2 utiliza os elementos dessa lista para compor uma nova lista, atribuída à variável `numeros`, através do desempacotamento de elementos com o operador `*`. É válido que ressaltar que após a operação de desempacotamento da linha 2, a lista apontada por `primos` permanece inalterada. Também é útil observar, todavia, que sem o operador `*`, a operação resulta em uma lista diferente, onde o segundo elemento é outra lista, conforme pode ser verificado a partir do Código 7.4:

```
1 >>> primos = [1, 3, 5]
2 >>> numeros = [0, primos, 10]
3 >>> numeros
4 [0, [1, 3, 5], 10]
```

Código 7.4: inclusão de uma lista como elemento de outra lista.

A operação de desempacotamento também pode ser utilizada para passar elementos de uma lista como argumentos à uma função:

```
1 >>> primos = [1, 3, 5]
2 >>> def soma(a, b, c):
3         return a + b + c
4 >>> soma( *primos )
5 9
```

Código 7.5: desempacotamento de elementos de uma lista para chamada de função.

No Código 7.5, na linha 4, a função `soma` é chamada recebendo como argumentos de entrada os elementos da lista apontada por `primos`. Assim, será aberto um escopo local para a execução da função `soma` fazendo `a = primos[0]`, `b = primos[1]` e `c = primos[2]`, isto é, `a = 1`, `b = 3` e `c = 5`. Desse modo, a função retorna o resultado `9`. Note que, no desempacotamento de uma lista para chamada de uma função, é necessário que o número de elementos na lista seja compatível com o número de argumentos de entrada que a função sendo chamada deve receber.

7.2 Métodos de listas

Na Seção 5.7, vimos que métodos são funções definidas dentro de uma classe (tipo). A classe das listas, denominada `list`, também possui sua relação de métodos, aos quais podem ser executados a partir de qualquer objeto lista. Uma

vez que listas são objetos mutáveis, seus métodos estão habilitados a realizarem alterações no próprio objeto a partir dos quais são chamados. Alguns dos métodos de uso mais comum de `list` são:

- `append( objeto )`: Anexa `objeto` ao final da lista:

```
1 >>> primos = [2, 3]
2 >>> primos.append(5)
3 >>> primos
4 [2, 3, 5]
```

Código 7.6: uso do método append.

Observe que a linha 2 do Código 7.6 utiliza o método `append` para inserir o objeto `5` na lista apontada por `primos`. Poderíamos ter conseguido efeito semelhante com a operação `primos = primos + [5]`. No entanto, é preferível o uso do método `append`, uma vez que ele altera a própria lista para que comporte um novo objeto, no lugar de criar uma nova lista totalmente nova com o operador `+`. Assim, a operação com `append` é mais eficiente do que com `+`.

É de suma importância ressaltar que este método altera a própria lista a partir da qual está sendo chamado. Note que esse comportamento é diferente dos métodos de string, que não possuem este poder de alteração devido a imutabilidade dos objetos da classe. Observe portanto que este método não retorna qualquer valor, o que faz com que o `None` seja automaticamente retornado pelo interpretador Python. Assim, seria equivocado atribuir o retorno do método à variável, conforme a seguir:

```
1 >>> primos = primos.append(7)  # EQUIVOCADO! Método append
      retorna None!
2 >>> primos
3 None
```

Observe que, no exemplo anterior, a atribuição do resultado do método `append` à variável `primos` fez com que essa última passasse a apontar para o objeto `None`. Assim, a lista original apontada por `primos` pode acabar sendo perdida, caso não haja outra variável apontando para a mesma.

- `count( objeto )`: **retorna** o número de vezes que `objeto` aparece na lista:

```
1 >>> sorteio = ["cara", "cara", "coroa", "cara"]
2 >>> sorteio.count("cara")
3 3
4 >>> sorteio.count("coroa")
5 1
```

Observe que esse método apenas retorna um valor numérico, não provocando qualquer alteração na lista.

- `extend( sequencia )`: acrescenta os elementos de `sequencia` ao final da lista.

```
>>> cidades = ["rio", "ann arbor"]
>>> cidades.extend( ["bh", "uberlandia"] )
>>> cidades
['rio', 'ann arbor', 'bh', 'uberlandia']
```

- `index( objeto )`: **retorna** o índice onde `objeto` aparece pela primeira vez. Caso `objeto` não apareça na lista, um erro (exceção) é lançado.

```
>>> n = [3, 7, 9, 7, 2]
>>> n.index(7)
1
```

Adicionalmente, o método pode receber argumentos opcionais que especificam índices de início e de fim para a busca de `objeto`.

```
>>> n.index(7, 2)     #busca por 7 a partir do indice 2
3
>>> n.index(7, 2, 4)   #busca por 7 entre os indices 2 e 4
3
```

- `insert( indice, objeto )`: insere `objeto` na lista no índice especificado.

```
>>> paises = ["brasil", "portugal", "espanha"]
>>> paises.insert(1, "eua")
>>> paises
['brasil', 'eua', 'portugal', 'espanha']
```

- `remove( objeto )`: remove a primeira ocorrência de `objeto` na lista. Se `objeto` não aparecer na lista, um erro (exceção) é lançado.

```
>>> sorteio = ["cara", "cara", "coroa", "cara"]
>>> sorteio.remove("cara")
>>> sorteio
['cara', 'coroa', 'cara']
```

- `reverse( )`: reverte a posição dos elementos da lista.

```
>>> paises = ["brasil", "portugal", "espanha"]
>>> paises.reverse()
>>> paises
['espanha', 'portugal', 'brasil']
```

- `sort( )`: ordena os elementos da lista.

```
>>> paises = ["brasil", "portugal", "espanha", "argentina"]
>>> paises.sort()
>>> paises
['argentina', 'brasil', 'espanha', 'portugal']
```

Note que por padrão, a ordenação ascendente (do menor para o maior). É possível passar um parâmetro booleano denominado `reverse` para especificar a ordenação descendente (do maior para o menor):

```
>>> notas = [23, 89, 100.0, 14, 56]
>>> notas.sort( reverse = True )
>>> notas
[100.0, 89, 56, 23, 14]
```

Até antes da versão 3 de Python, o método `sort` podia ordenar listas que misturavam tipos diferentes de objetos usando uma ordem de tipos pré-estabelecida pela linguagem. A partir da versão 3, essa ordem foi abolida, e o método `sort` apenas opera com listas onde todos os objetos sejam do mesmo tipo, ou haja alguma implementação de ordem entre os tipos diferentes presentes na lista (é possível usar `sort` para ordenar uma lista com objetos `int`, `float` e `complex`, por exemplo).

Para ver a relação completa de métodos da classe `list`, é possível utilizar a função `help` no prompt:

```
>>> help(list)
```

Pode-se exibir a ajuda específica de um determinado, por exemplo, o `sort`:

```
>>> help(list.sort)
```

7.3 Percorrendo os elementos de uma lista

Podemos percorrer os elementos de uma lista por meio de um laço de repetição. O Código 7.7 percorre uma lista através de seus índices.

```
personagens = ["doug", "patti", "skeeter", "judy"]

for i in range(0, len(personagens)):
    print( personagens[i] )
print("--Sou eu!")
```

Código 7.7: percorrendo uma lista através de seus índices.

A execução do Código 7.7 produzirá como saída:

```
doug
patti
skeeter
judy
--Sou eu!
```

Também é possível percorrer uma lista diretamente com um laço `for`, conforme exemplificado no Código 7.8, que produzirá a mesma saída do Código 7.7:

```
personagens = ["doug", "patti", "skeeter", "judy"]

for i in personagens:
    print( i )
print("--Sou eu!")
```

Código 7.8: percorrendo uma lista diretamente.

Note que, no Código 7.7, a linha 3 faz a variável `i` percorrer os índices da lista apontada por `personagens`. Por sua vez, no Código 7.8, a linha 3 faz `i` percorrer os elementos da lista apontada por `personagens` diretamente.

Percorrer diretamente os elementos de uma lista pode ser a estratégia mais elegante em diversas situações. Todavia, utilizar os índices da sequência pode ser mais apropriado quando for necessário usar a posição dos elementos (por exemplo, calcular o somatório apenas dos itens nos índices ímpares), ou quando for preciso percorrer mais de uma sequência simultaneamente. O Código 7.9 traz um exemplo onde percorremos, simultaneamente, uma lista com nomes e outra lista com sobrenomes usando índices.

```
1 nomes = ["jonas", "carlos", "rodrigo"]
2 sobrenomes = ["degrave", "sartin", "duarte"]
3
4 for i in range(0, 3, 1):
5   print( nomes[i], sobrenomes[i] )
```

Código 7.9: percorrendo duas listas através dos índices.

A execução do Código 7.9 é exibida a seguir:

```
jonas degrave
carlos sartin
rodrigo duarte
```

7.4 Alguns exemplos

7.4.1 Ordenação de valores lidos

O Código 7.10 traz um exemplo de programa que lê n números do teclado e os imprime de forma ordenada:

```
1  #programa que lê numeros e os imprime de modo ordenado:
2
3  n = int( input("Entre com a quantidade de numeros: ") )
4
5  numeros = [0]*n  #Cria uma lista com n elementos 0
6  for i in range(0, n):
7    numeros[i] = float( input("Entre com o numero %s: "%(i)) )
8
9  numeros.sort() #ordena os numeros
10
11 #imprimindo os numeros da lista ja ordenada:
12 print("Numeros ordenados:")
13 for num in numeros:
14   print(num)
```

Código 7.10: ordenação de numeros lidos.

Na linha 3 do Código 7.10, a quantidade de números a serem ordenados é lida e armazenada na variável `n`. Na linha 5, é criada uma lista de `n` elementos 0, para armazenar todos os `n` números que serão lidos através do laço `for` nas linhas 6 - 7. Essa lista é então armazenada na variável `numeros`. Observe que, na linha 7, cada número lido é armazenado em uma posição diferente da lista apontada por `numeros`. Assim, utilizamos a lista para não perder nenhum dos valores lidos. Observe que a variável `i` deve iterar no conjunto de índices de `numeros`, isto é, de `0` até `n-1`, para que o armazenamento dos valores ocorra corretamente.

Como usuários não estão acostumados a começar a contar a partir do zero, a linha 7 usa o valor `i+1` na string que é passada à função `input`. Seguindo a lógica do programa, a linha 9 faz a ordenação dos valores armazenados na lista apontada por `numeros`. Após esta ordenação, os números serão impressos em ordem graças ao `for` nas linhas 13 - 14. Um exemplo de execução do Código 7.10 é fornecido a seguir:

```
Entre com a quantidade de numeros: 4
Entre com o numero 1: 67
Entre com o numero 2: -23
Entre com o numero 3: 140
Entre com o numero 4: 12
Numeros ordenados:
-23.0
12.0
67.0
140.0
```

7.4.2 Média aritmética e média geométrica

Dado uma sequência de T termos não negativos, $t_1, t_2, \ldots, t_T$, definimos a média aritmética M_A como:

$$M_A = \frac{t_1 + t_2 + \cdots + t_T}{T} \tag{7.1}$$

Por sua vez, a média geométrica destes mesmos termos pode ser definida como:

$$M_G = \sqrt[T]{t_1 \times t_2 \times \cdots \times t_T} \tag{7.2}$$

Dada uma turma de T alunos, construiremos um programa que lê notas entre 0 e 100 e informe quantos alunos ficaram acima da médias aritmética, e quantos alunos ficaram acima da média geométrica. Observe que, devido ao fato de precisarmos comparar cada nota com as médias aritmética e geométrica, que são podem ser determinadas após a leitura da última nota, será necessário utilizar alguma estrutura para armazenar todas as notas lidas. Assim, seguiremos os seguintes passos para solucionar a situação proposta:

1. Ler o número de alunos da turma;
2. Ler cada nota da turma, armazenando os valores lidos em uma lista;
3. Calcular as médias aritmética e geométrica da turma;
4. Comparar cada nota armazenada na lista de notas com as médias, contando quantas notas estão acima da média aritmética, e quantas notas estão acima da média geométrica.

Em nosso programa, exibido no Código 7.11, temos as seguintes funções:

1. `mediaAritmetica`: função que recebe uma lista com valores e retorna sua média aritmética, definida na linha 4;

2. `mediaGeometrica`: função que recebe uma lista com valores e retorna sua média geométrica, definida na linha 10;

3. `contaAcimaPatamar`: função que recebe uma lista com valores, um patamar, e retorna quantos valores da lista estão acima do patamar, definida na linha 18.

```
1  #programa que lê notas de uma turma e informa quantos
2  #alunos ficaram acima da média geométrica
3
4  def mediaAritmetica( valores ):
5    soma = sum(valores)
6    nvalores = len(valores)
7    media = float(soma)/nvalores
8    return media
9
10 def mediaGeometrica( valores ):
11   nvalores = len(valores)
12   produto = 1
13   for t in valores:
14     produto = t * produto
15   media = produto**( 1.0/nvalores )
16   return media
17
18 def contaAcimaPatamar( valores, patamar ):
19   nacima = 0
20   for t in valores:
21     if t > patamar:
22       nacima += 1
23   return nacima
24
25
26 nalunos = int( input("Entre com o numero de alunos: ") )
27 notas = []
28
29 for i in range(1, nalunos+1, 1):
30   nota = float( input("Entre com a nota do aluno %s: "%(i) ) )
31   notas.append( nota)
32
33 #calculando a media aritmetica das notas
34 ma = mediaAritmetica( notas )
35 mg = mediaGeometrica( notas )
36
37 acima_ma = contaAcimaPatamar(notas, ma)
38 acima_mg = contaAcimaPatamar(notas, mg)
39
40 print("Media aritmetica: ", ma)
41 print("Alunos acima da media aritmetica: ", acima_ma)
42 print("Media geometrica: ", mg)
43 print("Alunos acima da media geometrica: ", acima_mg)
```

Código 7.11: contagem de valores acima das médias aritmética e geométrica.

Na linha 26, o Código 7.11 faz a leitura do número de alunos na turma. Na linha 27, é inicializada uma lista vazia para armazenar as notas dos alunos, que são lidas nas linhas 29 - 31. Note que cada nota lida é acrescentada na lista apontada por `notas` na linha 31. Essa estratégia de composição da lista de notas é diferente da utilizada no Código 7.10, que já inicializa uma lista de zeros com

o tamanho apropriado e sobrescreve cada zero com um valor lido. No Código 7.11, a lista começa vazia e vai sendo gradativamente ampliada com cada nova nota lida. A título de curiosidade, a mesma estratégia de armazenamento de valores em lista utilizada no Código 7.10 poderia ter sido também usada no Código 7.11.

Nas linhas 34 e 35 do Código 7.11, são chamadas as funções definidas para os cálculos das médias aritmética e geométrica, ao passo que as linhas 37 e 38 aproveitam a função `contaAcimaPatamar` para contar o número de alunos acima da média aritmética e geométrica. Por fim, as linhas 40 - 43 imprimem os resultados obtidos.

É válido apontar que, na função `mediaAritmetica`, é utilizada a função `sum`, já definida pela linguagem Python, para calcular o somatório dos elementos de uma sequência na linha 5. Ainda nessa mesma função, é realizada uma divisão garantindo que o numerador seja do tipo `float` na linha 7, apenas uma precaução para garantir que esta função funcione adequadamente no Python 2 calculando a divisão real no lugar da divisão inteira.

Um exemplo de execução do Código 7.11 é exibido a seguir:

```
Entre com o numero de alunos: 4
Entre com a nota do aluno 1: 90
Entre com a nota do aluno 2: 25
Entre com a nota do aluno 3: 74
Entre com a nota do aluno 4: 51
Media aritmetica:  60.0
Alunos acima da media aritmetica:  2
Media geometrica:  53.98164359142943
Alunos acima da media geometrica:  2
```

7.5 Usando listas para processamento de texto

Pelo fato das strings serem objetos imutáveis não é possível realizar alterações sobre as mesmas. Em algumas situações, isso, gera empecilhos para a realização de processamento de texto. Por exemplo, ao realizar a atribuição a seguir:

```
1 >>> nome = "gessyka"
```

Ao se tentar trocar o primeiro caracter da string apontada por `nome` de `"g"` para `"j"`, obtemos o seguinte erro, ocasionado pela imutabilidade das strings:

```
1 >>> nome[0] = "j"
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: 'str' object does not support item assignment
```

Uma forma de superar a limitação da imutabilidade de strings é realizar a conversão para listas, conforme o Código 7.12:

```
1  >>> nome = "gessyka"
2  >>> nomelista = list(nome)
3  >>> nomelista
4  ['g', 'e', 's', 's', 'y', 'k', 'a']
5  >>> nomelista[0] = "j"
6  >>> nomelista[4] = "i"
7  >>> nomelista[5] = "c"
8  >>> nomelista
9  ['j', 'e', 's', 's', 'i', 'c', 'a']
10 >>> nomenovo = "".join(nomelista)
11 >>> nomenovo
12 'jessica'
```

Código 7.12: processamento de texto através de lista.

Observe que, na linha 2, uma string é atribuída à variável `nome`. Na linha 2, é atribuída à variável `nomelista` uma lista gerada partir dos caracteres da string em `nome`, de modo a permitir as trocas diretas de caracteres realizadas nas linhas 5-7. O resultado final das alterações sobre a lista é exibido na linha 9. A partir da lista alterada, na linha 10 geramos uma string através do método `join` (apresentado na Seção 5.7). O resultado final da operação é ecoado na linha 11.

7.6 Usando listas para manusear matrizes

É possível usar listas aninhadas para manusear matrizes[1]. Por exemplo, considere as matrizes:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 \end{bmatrix} \tag{7.3}$$

podemos representá-las através do seguinte código:

```
1 >>> A = [ [1,2], [3,4], [5,6] ]
2 >>> B = [ [7,8,9,10], [11,12,13,14] ]
```

Código 7.13: definição das matrizes da equação (7.3) usando listas aninhadas.

Observe que as matrizes A e B foram representadas como listas de "linhas", onde cada linha é representada como uma lista com os respectivos coeficientes. Desse modo, pode-se acessar coeficientes matriciais através de seus índices de linha e coluna, com a peculiaridade de que os índices começam a serem contados a partir do zero:

```
1 >>> A[2][0] #acessa o elemento na linha 2 e coluna 0 de A
2 5
3 >>> B[1][3] #acessa o elemento na linha 1 e coluna 3 de B
4 14
```

Observe que o comprimento da lista apontada `A` dá o número de linhas da matriz. O número de colunas é dado pelo comprimento de uma das listas inter-

[1] Apesar do uso de listas aninhadas ser uma forma rudimentar de operar com matrizes em Python, é importante a apresentação dessa técnica com o objetivo de desenvolver a habilidade de programação de estruturas aninhadas. Formas mais práticas de lidar com matrizes envolvem o uso de pacotes especializados, como, por exemplo, NumPy.

nas de `A`. Pelo fato da matriz ser representada por listas aninhadas, precisamos de laços aninhados para percorrer todos os seus elementos.

7.6.1 Função para impressão de matriz

O Código 7.14 apresenta uma função que imprime os coeficientes de uma matriz com a representação aqui discutida:

```
1 def imprimeMatriz(matriz):
2   for linha in matriz:
3     for coef in linha:
4       print(coef, " ", end = "") #end="" para não pular linha
5     print() #só para pular linha
```

Código 7.14: função para impressão de matriz armazenada sob listas aninhadas.

Na linha 1 do Código 7.14, temos a declaração da função que receberá como argumento a matriz a ser impressa. Na linha 2, temos um laço `for` que fará a variável `linha` percorrer a lista apontada por `matriz`. Desse modo, `linha` assumirá cada uma das listas internas que representam a matriz. Na linha 3, temos um novo laço `for`, onde a variável `coef` percorrerá todos os coeficientes de lista apontada por `linha`. Note que este segundo `for` está dentro do laço de repetição do primeiro, o que significa que a linha 3 será executada para cada lista que represente uma linha da matriz. Na linha 4 cada coeficiente é impresso. Uma vez que desejamos imprimir todos os coeficientes lado a lado, passamos o argumento `end = ""` à função `print` para determinar que, ao final da impressão, não deve ser pulada uma linha na tela. Por fim, a linha 5 chama mais uma vez a função `print` com o único objetivo de pular uma linha na tela após a impressão de todos os coeficientes de cada linha da matriz. Observe que a função `imprimeMatriz` possui a peculiaridade de imprimir informação na tela e de não retornar qualquer valor.

A seguir, chamamos a função `imprimeMatriz` passando como argumento as matrizes definidas no Código 7.13.

```
1 >>> imprimeMatriz(A)
2 1  2
3 3  4
4 5  6
```

```
1 >>> imprimeMatriz(B)
2 7  8  9  10
3 11  12  13  14
```

7.6.2 Geração de uma matriz de zeros

O Código 7.15 define a função `geraMatrizZeros`, que gera uma matriz de zeros.

```
1 def geraMatrizZeros(nlinhas, ncolunas):
2   matriz = [0]*nlinhas
3   for i in range(0, nlinhas):
4     matriz[i] = [0]*ncolunas
5   return matriz
```

Código 7.15: função para geração de uma matriz de zeros.

A linha 1 do Código 7.15 declara a função `geraMatrizZeros` que recebe como argumentos o número de linhas (`nlinhas`) e o número de colunas (`ncolunas`) que a matriz gerada deve possuir. Na linha 2, criamos uma lista com `nlinhas` posições preenchidas com `0`. Posteriormente, cada um desses valores `0` será

sobrescrito por uma lista de `ncolunas` posições também preenchidas com `0`, graças ao laço `for` na linha 3 e a atribuição na linha 4. Por fim, a linha 5 retorna a matriz gerada.

A seguir, exemplos de chamada à função `geraMatrizZeros`:

```
1 >>> geraMatrizZeros(2, 4)
2 [[0, 0, 0, 0], [0, 0, 0, 0]]
```

```
1 >>> geraMatrizZeros(3, 2)
2 [[0, 0], [0, 0], [0, 0]]
```

Leitores mais aguçados poderiam sugerir que uma forma mais simples de gerar uma matriz de zeros seria através da operação `matriz = [ [0]*ncolunas ]*nlinhas]`. Apesar de aparentemente essa operação gerar o resultado esperado, ela está equivocada, pois na realidade, ela fará com que todas as linhas da matriz apontem para a mesma lista de `ncolunas` zeros. Assim, a estrutura de matriz não será satisfatoriamente gerada, conforme o Código 7.16:

```
1 >>> nlinhas = 3
2 >>> ncolunas = 2
3 >>> matriz= [[0]*ncolunas]*nlinhas #jeito ERRADO de gerar matriz
4 >>> matriz
5 [[0, 0], [0, 0], [0, 0]]
6 >>> matriz[0][0] = 7
7 >>> matriz
8 [[7, 0], [7, 0], [7, 0]]
```

Código 7.16: geração equivocada de uma matriz de zeros.

Repare que, após gerar a matriz equivocadamente na linha 3, a operação de trocar o elemento na linha 0 e coluna 0 por 7 (linha 6) fez com também fossem trocados os elementos na coluna 0 nas demais linhas (veja a linha 8), pois, na realidade, todos as linhas de `matriz` apontam para a mesma lista de coeficientes, conforme ilustrado na Figura 7.3. Isso ocorre porque o operador `*` apenas copia os apontadores de uma lista para gerar a repetição da mesma.

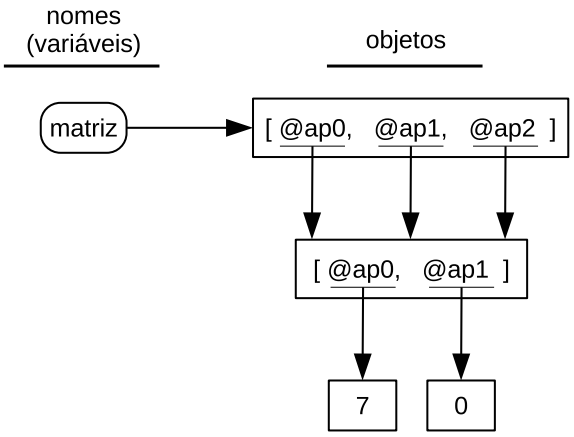

Figura 7.3: estado corrente da memória após geração de estrutura matricial equivocada (Código 7.16).

7.6.3 Exemplo: multiplicação de matriz por fator

O Código 7.17 introduz um programa que lê uma matriz do teclado, elemento a elemento, e imprime o resultado da multiplicação da matriz por um fator, também lido do teclado. As linhas 3 e 9 definem, respectivamente, as funções `geraMatrizZeros` e `imprimeMatriz`, já discutidas nas Seções 7.6.1 e 7.6.2.

Na linha 15 do Código 7.17, definimos a função `multiplicaMatrizFator` que recebe uma matriz M e um fator f, e retorna uma matriz $\bar{M}$, onde $\bar{M} = fM$. Após a geração da estrutura da nova matriz com a função `geraMatrizZeros` na linha 19, sobrescrevemos os coeficientes da nova matriz com os resultados das multiplicações dos coeficientes da matriz original pelo fator (linhas 21 - 23).

Após a leitura do teclado das dimensões da matriz (isto e quantidade de linhas e colunas) nas linhas 28 e 29, geramos uma estrutura para a matriz que será lida do teclado (linha 31). Os coeficientes zero desta matriz serão sobrescritos com valores lidos do teclado nas linhas 33 - 35. Em seguida, chamamos na linha 39 nossa função `multiplicaMatrizFator` para realizar a multiplicação da matriz lida com o fator lido na linha 37. Ao final, chamamos a função `imprimeMatriz` para imprimir o resultado na linha 45.

A seguir, um exemplo de execução do Código 7.17:

```
Entre com o numero de linhas: 2
Entre com o numero de colunas: 3
Entre com o coeficiente na linha 0, coluna 0: 1
Entre com o coeficiente na linha 0, coluna 1: 2
Entre com o coeficiente na linha 0, coluna 2: 3
Entre com o coeficiente na linha 1, coluna 0: 4
Entre com o coeficiente na linha 1, coluna 1: 5
Entre com o coeficiente na linha 1, coluna 2: 6
Entre com o fator de multiplicacao: 50
Matriz original:
1.0  2.0  3.0
4.0  5.0  6.0
Matriz multiplicada:
50.0  100.0  150.0
200.0  250.0  300.0
```

7.7 Cópia rasa e cópia profunda

Listas são objetos mutáveis, o que nos permite fazer alterações nas mesmas. Em algumas situações, pode ser desejável criar uma cópia de uma lista para que seja possível realizar alterações sobre a cópia ao mesmo tempo em que preserva a lista com os dados originais. Por exemplo, suponha a criação da lista a seguir:

```
1 >>> L1 = [1, 2, 3]
```

para a criação de uma cópia da lista `L1`, um programador descuidado poderia realizar a operação:

```
1 >>> L2 = L1     #não gera copia de L1
```

No entanto, a operação acima não copia a lista, apenas faz a variável `L2` apontar

```
#programa que le uma matriz, um fator, e multiplica
#a matriz pelo fator lido
def geraMatrizZeros(nlinhas, ncolunas):
  matriz = [0]*nlinhas
  for i in range(0, nlinhas):
    matriz[i] = [0]*ncolunas
  return matriz

def imprimeMatriz(matriz):
  for linha in matriz:
    for coef in linha:
      print(coef, " ", end = "")
    print()

def multiplicaMatrizFator( matriz, fator ):
  nlinhas = len(matriz)
  ncolunas = len(matriz[0])

  novamatriz = geraMatrizZeros(nlinhas, ncolunas)

  for i in range(0, nlinhas):
    for j in range(0, ncolunas):
      novamatriz[i][j] = fator * matriz[i][j]

  return novamatriz


numLinhas = int( input("Entre com o numero de linhas: ") )
numColunas = int( input("Entre com o numero de colunas: ") )

M = geraMatrizZeros(numLinhas, numColunas) #gerando uma matriz

for i in range(0, numLinhas):
  for j in range(0, numColunas):
    M[i][j] = float( input("Entre com o coeficiente na linha %s,
        coluna %s: "%(i,j)) )

fator = float( input("Entre com o fator de multiplicacao: ") )

Mfactor = multiplicaMatrizFator(M, fator)

print("Matriz original: ")
imprimeMatriz(M)

print("Matriz multiplicada: ")
imprimeMatriz(Mfactor)
```

Código 7.17: programa que le matriz e a multiplica por um fator.

para o mesmo objeto que L1 aponta. Assim, ao realizar qualquer alteração na lista, como, por exemplo:

```
>>> L2[0] = 9
```

A alteração será refletida tanto no objeto apontado por L1 quanto no apontado por `L2`, pois, na realidade, ambas as variáveis apontam para o mesmo objeto:

```
>>> L2
[9, 2, 3]
```

```
>>> L1
[9, 2, 3]
```

Uma forma de gerar uma cópia de uma lista é através da operação de fracionamento, como no exemplo a seguir:

```
>>> L1 = [1,2,3]
>>> L2 = L1[:]
```

Pelo fato da operação de fracionamento gerar uma nova sequência, a operação `L1[:]` efetivamente irá gerar uma cópia da lista apontada por `L1`. Esse tipo de cópia é denominado como cópia rasa porque possui a característica de copiar apenas os apontadores da lista original, conforme ilustrado na Figura 7.4.

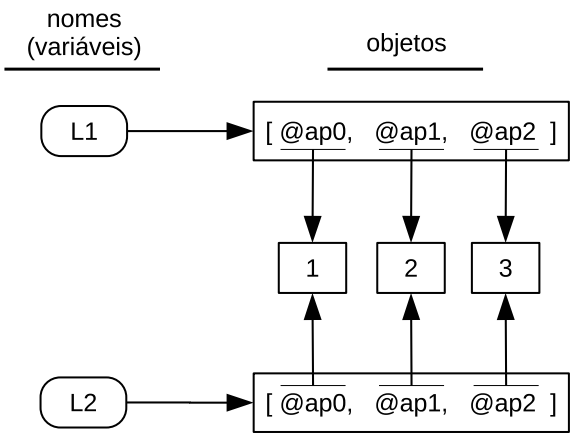

Figura 7.4: estado corrente da memória após operação de cópia rasa (*L1 = [1,2,3]; L2 = L1[:]*).

Através da Figura 7.4, podemos observar que a cópia rasa não copia os objetos sendo apontados pelos apontadores da lista, pois a cópia se limita apenas ao primeiro nível (apenas os apontadores). Assim, os apontadores de ambas as listas apontarão para os mesmos objetos. Como todos os objetos apontados são imutáveis, o fato de haverem diversos apontadores para o mesmo objeto não traz risco de alteração acidental. Podemos, por exemplo, alterar a lista L2 com a operação:

```
>>> L2[0] = 5
```

que, ainda assim, o objeto apontado por L1 se mantém inalterado, conforme observado na Figura 7.5.

A cópia rasa se mostrará eficaz sempre que se desejar copiar um objeto que só contenha objetos imutáveis, como no exemplo aqui discutido. Todavia, se o

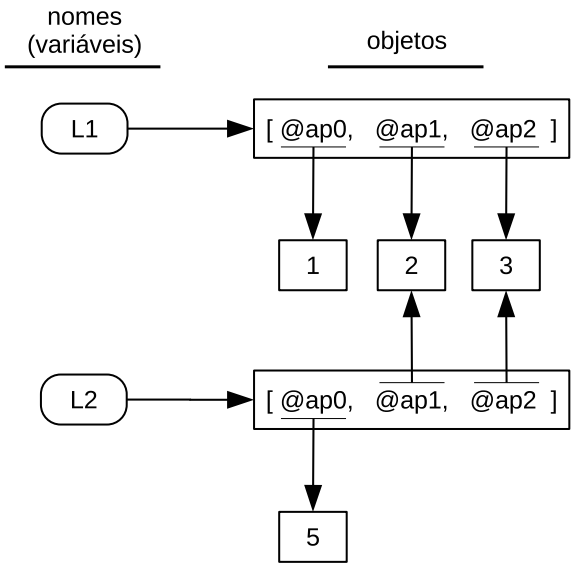

Figura 7.5: estado corrente da memória após alteração de cópia rasa (*L2[0] = 5*).

objeto sendo copiado possuir um objeto imutável em seu interior, a cópia rasa pode ser suficiente. Por exemplo, considere a seguinte atribuição e cópia rasa:

```
1 >>> L3 = [7, [4,5]]
2 >>> L4 = L3[:]
```

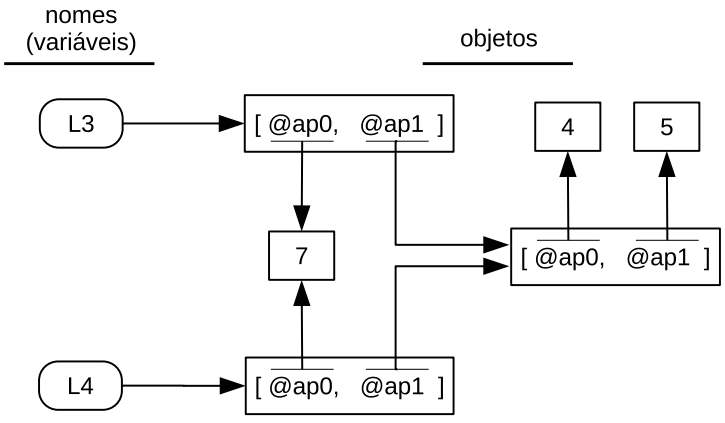

Figura 7.6: estado corrente da memória após operação de cópia rasa (*L3 = [7, [4,5]]; L4 = L3[:]*).

As instruções anteriores gerarão o estado na memória exibido na Figura 7.6. Observe que, ao alteramos a lista interna, a alteração será refletida em ambos os objetos apontados por `L3` e `L4`, pois seus respectivos apontadores de índice 1 (`@ap1`) apontam para essa mesma lista, conforme a seguir:

```
>>> L4[1][0] = 0
>>> L3
[7, [0, 5]]
>>> L4
[7, [0, 5]]
```

A Figura 7.7 ilustra o estado da memória após a alteração anterior.

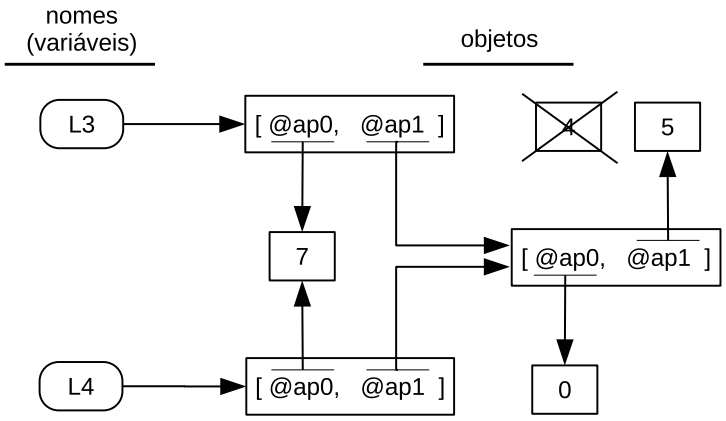

Figura 7.7: estado corrente da memória após alteração de cópia rasa (*L4[1][0] = 0*).

Para o caso do objeto sendo copiado conter objetos imutáveis, o mais indicado é a cópia profunda. Diferentemente da cópia rasa, a cópia profunda "mergulha" nos apontadores copiando todo objeto que for encontrado até o último nível. Para realizar uma cópia profunda, podemos usar a função `deepcopy` presente no módulo `copy` conforme exemplificado a seguir:

```
>>> import copy
>>> L3 = [7, [4,5]]
>>> L4 = copy.deepcopy(L3)
>>> L4[1][0] = 0
>>> L3
[7, [4, 5]]
>>> L4
[7, [0, 5]]
```

Código 7.18: cópia profunda de lista.

A Figura 7.8 ilustra o estado da memória após a operação `L4 = copy.deepcopy(L3)`. Note que a função `deepcopy` gera uma cópia de todos os objetos apontados a partir da lista apontada por `L3`. O módulo `copy` é nativo da linguagem Python. No Capítulo 8 discutiremos em detalhes o processo de importação de módulos.

7.8 Tuplas

Assim como as listas, tuplas também são sequências de objetos arbitrários sob uma ordem determinada, com a diferença de que tuplas são objetos imutáveis e podem ser declaradas utilizando parênteses no lugar de colchetes:

```
>>> tupla = ("sayuri", True, -14.86, [7,8])
```

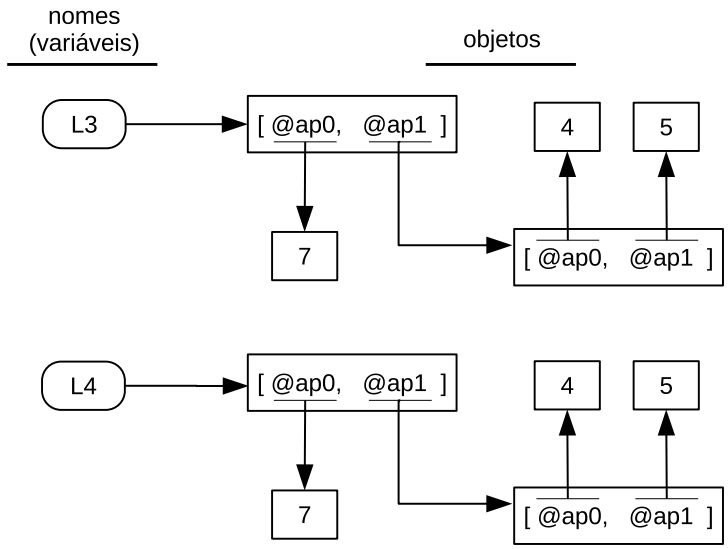

Figura 7.8: estado corrente da memória após cópia profunda (Código 7.18).

Observe que tuplas também podem outras tuplas ou listas. Quando não há ambiguidade, é possível declarar tuplas sem o uso de parênteses:

```
>>> figuras = "adrianne", "ariel", "beth"
>>> figuras
('adrianne', 'ariel', 'beth')
```

Operações definidas para listas que não tragam modificações também são definidas para tuplas. Assim operadores como `+`, `*`, `in`, `not in`, `len` assim como indexação e fracionamento também estão definidos para tuplas. O nome da classe (construtor) é `tuple`, que permite gerar tuplas vazias ou a partir de outros objetos sequenciais.

```
>>> tu1 = tuple()#tupla vazia
>>> tu1
()
```

```
>>> tu3 = tuple( [3,4,5] )
>>> tu3
(3, 4, 5)
```

```
>>> tu2 = tuple( "caue" )
>>> tu2
('c', 'a', 'u', 'e')
```

```
>>> tu4 = tuple({"joao", 10})
>>> tu4
(10, 'joao')
```

Pelo fato de serem objetos mutáveis, listas não podem ser utilizadas em alguns contextos restritos a objetos imutáveis como, por exemplo, a participação como membro de conjunto (`set`) ou uso como chave de dicionários (abordados no Capítulo **??**). Todavia, por serem objetos imutáveis, tuplas podem ser utilizadas nessas situações. Ademais, tuplas também permitem atribuições múltiplas, que são atribuições simultâneas a mais de uma variável:

```
>>> (a,b) = (3, 5)
>>> a
3
>>> b
5
```

```
>>> x, y, z = (7, 8, 19)
>>> x
7
>>> y
8
>>> z
19
```

Observe pelo exemplo anterior, que foi possível atribuir valores a diferentes variáveis simultaneamente utilizando tuplas. As tuplas de ambos os lado da atribuição devem possuir o mesmo tamanho, e a atribuição é feita segundo a ordem dos elementos. Desse modo, a primeira variável da tupla à esquerda receberá o primeiro valor da tupla à direita, e, assim, sucessivamente.

7.9 Exercícios

1. Faça um programa que leia dois vetores do teclado, com n coordenadas cada um, e informe se os vetores são múltiplos um do outro. Nota: um vetor v é dito ser múltiplo de um vetor u se existe um número real α tal que $v = \alpha u$. Por exemplo, o vetor $v = [8, 14, 12]$ é considerado múltiplo de $u = [4, 7, 6]$, pois as coordenadas de v podem ser obtidas multiplicando as coordenadas de u por 2. Para fazer este programa, defina uma função que receba como argumento de entrada duas listas representando, cada uma, um vetor, e, **retorne** `True` se os vetores são múltiplos entre si e `False` caso contrario.

   Exemplo:

   ```
   Entre com o valor de n: 3

   Entre com a coordenada 1 do vetor 1: 10
   Entre com a coordenada 2 do vetor 1: 7
   Entre com a coordenada 3 do vetor 1: -8

   Entre com a coordenada 1 do vetor 2: 0
   Entre com a coordenada 2 do vetor 2: 19
   Entre com a coordenada 3 do vetor 2: 8

   Estes vetores não são multiplos entre si
   ```

2. A mediana é uma medida estatística definida como o termo central de uma amostra após sua ordenação. Por exemplo, a amostra:

   $$\{2, 1, 2, 5, 6, 11, 9\}$$

   ao ser ordenada fica:

   $$\{1, 2, 2, 5, 6, 9, 11\}$$

   . Logo, sua mediana é 5, pois este é o elemento presente na quarta posição (posição central) da ordenação.

   Para uma amostra com número par de termos, a mediana é tomada como a média aritmética entre as duas posições centrais. Por exemplo, a mediana

da amostra ordenada:

$$\{-1, 6, 7, 18\}$$

é $\frac{6+7}{2} = 6,5$

Escreva um programa que leia uma amostra de números do teclado e informe a sua mediana. Os números devem ser lidos separadamente. A parte específica do cálculo da mediana deve ser feita em uma função separada apenas para este fim. Esta função deve receber como argumento de entrada uma lista com a amostra e então **retornar** o valor da mediana. **Obs:** a função não deve alterar a amostra original do usuário, portanto, para fazer a ordenação da amostra, a função deve ordenar uma cópia da lista recebida como argumento de entrada.

Exemplos:

```
Entre com o tamanho da amostra: 5
Entre com o elemento 1 da amostra: 9
Entre com o elemento 2 da amostra: 7
Entre com o elemento 3 da amostra: -3
Entre com o elemento 4 da amostra: 16
Entre com o elemento 5 da amostra: 4
Mediana da amostra:  7.0
```

```
Entre com o tamanho da amostra: 4
Entre com o elemento 1 da amostra: 7
Entre com o elemento 2 da amostra: 2
Entre com o elemento 3 da amostra: 11
Entre com o elemento 4 da amostra: 6
Mediana da amostra:  6.5
```

3. Escreva uma **função** que receba um número inteiro positivo m como argumento de entrada e **retorne** uma lista com todos os divisores inteiros de m.

   Exemplos:

```
>>> calculaDivisores(10)
[1, 2, 5, 10]
```

```
>>> calculaDivisores(12)
[1, 2, 3, 4, 6, 12]
```

4. Um número perfeito é um número natural para o qual a soma de todos os seus divisores positivos próprios (excluindo ele mesmo) é igual ao próprio número. Por exemplo, o número 6 é um número perfeito, pois:

   $$6 = 1 + 2 + 3$$

   O próximo número perfeito é o 28, pois:

   $$28 = 1 + 2 + 4 + 7 + 14.$$

   Os quatro primeiros números perfeitos são: 6, 28, 496 e 8128. Escreva uma função que recebe um número m como entrada e retorna **True** se o m for perfeito e **False** caso contrário. Para fazer essa função, use a função gerada no exercício anterior.

   Exemplos:

```
>>> ehNumeroPerfeito(6)
True
>>> ehNumeroPerfeito(28)
True
```

```
>>> ehNumeroPerfeito(10)
False
>>> ehNumeroPerfeito(25)
False
```

5. Faça um programa que leia dois números inteiros positivos e informe o Máximo Divisor Comum (MDC) entre os dois números lidos. Faça o cálculo do MDC em uma função separada.

   Exemplos:

```
Entre com o primeiro numero: 12
Entre com o segundo numero: 16
Maximo Divisor Comum entre 12 e 16: 4
```

```
Entre com o primeiro numero: 60
Entre com o segundo numero: 10
Maximo Divisor Comum entre 60 e 10: 10
```

```
Entre com o primeiro numero: 23
Entre com o segundo numero: 37
Maximo Divisor Comum entre 23 e 37: 1
```

6. Você deve fazer um programa que descubra os ganhadores da mega quarta, uma modalidade de jogo de azar da *Bachatóvia* similar à mega sena onde apenas 4 dezenas são sorteadas. Assuma que cada aposta sempre é constituída de 4 dezenas, de 00 até 40. Seu programa deve ler o número de jogadores n, as apostas de 4 dezenas de cada jogador e, por último, **ler as 4 dezenas sorteadas**. Após as leitura das dezenas sorteadas, seu programa deve informar quais são os jogadores vencedores, **se houver algum**.

   Exemplo:

```
Entre com o numero de jogadores: 2

Dezena 1 do jogador 1: 22
Dezena 2 do jogador 1: 13
Dezena 3 do jogador 1: 35
Dezena 4 do jogador 1: 8

Dezena 1 do jogador 2: 40
Dezena 2 do jogador 2: 4
Dezena 3 do jogador 2: 21
Dezena 4 do jogador 2: 32

Dezena 1 do sorteio: 8
Dezena 2 do sorteio: 22
Dezena 3 do sorteio: 13
Dezena 4 do sorteio: 35
```

```
Vencedores da mega quarta:
jogador 1
```

Note que as dezenas podem não ser informadas em ordem e que pode haver mais de um jogador vencedor ou até mesmo nenhum. Assuma que o usuário sempre fornecerá entradas válidas, inclusive para as dezenas.

7. A variância e o desvio padrão são uma medidas estatísticas calculadas sobre uma amostra de números. Primeiramente, seja x_1, x_2, ..., x_n uma amostra de n números. A média da amostra $\bar{x}$ é definida como:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Uma vez tendo calculado a média $\bar{x}$, podemos calcular a variância v a partir da expressão:

$$v = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} = \frac{(x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \cdots + (x_n - \bar{x})^2}{n-1}$$

O desvio padrão d_p, por sua vez, é definido como a raiz quadrada da variância, isto é:

$$d_p = \sqrt{v}$$

Faça um programa que leia uma amostra de números e informe a sua média, variância e desvio padrão. Nesse programa, você deve declarar as seguintes três funções:

(a) `calculaMedia(amostra)`: recebe uma lista representando uma amostra de números e retorna sua média $\bar{x}$;

(b) `calculaVariancia(amostra)`: recebe uma lista representando uma amostra de números e retorna sua variância v;

(c) `calculaDesvioPadrao(mostra)`: recebe uma lista representando uma amostra de números e retorna seu desvio padrão d_p.

Exemplo:

```
Entre com o numero de elementos da amostra: 4
Entre com o elemento 1 da amostra: 6
Entre com o elemento 2 da amostra: 8
Entre com o elemento 3 da amostra: 10
Entre com o elemento 4 da amostra: 5

Media da amostra: 7.25
Variancia da amostra: 4.916666666666667
Desvio padrão da amostra: 2.217355782608345
```

8. Faça um programa que leia uma matriz quadrada $n \times n$ elemento a elemento do teclado e informe o somatório de elementos da sua diagonal principal. Faça uma função separada para o cálculo do somatório da diagonal.

Exemplo:

```
Entre com o numero de linhas e colunas da matriz: 3
Entre com o elemento 1,1: 1
Entre com o elemento 1,2: 2
Entre com o elemento 1,3: 0
Entre com o elemento 2,1: 4
Entre com o elemento 2,2: 5
Entre com o elemento 2,3: 0
Entre com o elemento 3,1: 7
Entre com o elemento 3,2: 8
Entre com o elemento 3,3: 9
Matriz lida:
1.0  2.0  0.0
4.0  5.0  0.0
7.0  8.0  9.0
Somatorio da diagonal:  15.0
```

9. Faça uma função que receba uma matriz quadrada A e retorne o valor do somatório dos elementos da diagonal secundária de A.

   Exemplo:

   ```
   >>> A = [ [1,2,0], [4,5,0], [7,8,9] ]
   >>> somaDiagonalSecundaria(A)
   12.0
   ```

10. Faça uma função que receba uma matriz A e retorne A^t, onde A^t é a matriz transposta de A.

    Exemplo:

    ```
    >>> A = [ [1,2,3], [4,5,6] ]
    >>> matrizTransposta(A)
    [[1, 4], [2, 5], [3, 6]]
    ```

11. Faça uma função que receba uma matriz A e retorne `True` se A for uma matriz triangular superior, e `False` caso contrário. **Obs:** uma matriz é dita triangular superior se for quadrada e todos os elementos abaixo da diagonal principal são 0.

    Exemplos:

    ```
    >>> A = [ [1,2,3], [0,5,6], [0,0,9]]
    >>> ehTriangularSuperior(A)
    True
    ```

    ```
    >>> B = [ [1,2,3], [4,5,6], [7,0,9] ]
    >>> ehTriangularSuperior(B)
    False
    ```

    ```
    >>> C = [ [1,2], [3,4], [5,6] ]
    >>> ehTriangularSuperior(C)
    False
    ```

12. Faça uma função que receba duas matrizes A e B de mesma dimensão e retorne uma matriz C, tal que $C = A + B$.

    Exemplo:

```
>>> A = [ [2,4], [3,1] ]
>>> B = [ [1,2], [6,5] ]
>>> somaMatriz(A, B)
[[3, 6], [9, 6]]
```

13. Faça uma função que receba duas matrizes A e B de mesma dimensão e retorne uma matriz C, tal que $C = A * B$.

    Exemplo:

```
>>> A = [ [2,4,6], [5,5,3] ]
>>> B = [ [7,-1,0], [2,3,1], [0,5,0] ]
>>> multiplicaMatriz(A,B)
[[22, 40, 4], [45, 25, 5]]
```

Capítulo 8

Importação de Módulos

Em nosso contexto, *módulos* são arquivos contendo código Python. No desenvolvimento de aplicações computacionais, muitas vezes é conveniente dividir o código fonte em diversos arquivos (módulos) de modo a facilitar a manutenção e o reaproveitamento desse código. Ao se dividir o código fonte em diversos módulos, é necessário realizar a integração das diversas partes da aplicação, o que é facilitado através do processo de *importação de módulos*. Em geral, define-se um módulo principal (*main*) o qual poderá então importar alguns dos outros módulos pertencentes à aplicação, que por sua vez, poderão importar cada um, um outro subconjunto de módulos, e, assim, sucessivamente.

A linguagem Python já traz um conjunto bastante diversificado com centenas de módulos que formam a denominada *biblioteca padrão* da linguagem. Isso significa que qualquer instalação padrão da linguagem disponibilizará esses módulos para serem importados em qualquer aplicação. No geral, esses módulos trazem definições (variáveis, funções, classes, exceções, etc) que facilitam o tratamento de problemas corriqueiros no desenvolvimento de programas. Por exemplo, no Código 7.18 (Seção 7.7), utilizamos a função `deepcopy` para realizar uma operação de cópia profunda. Pelo fato da função `deepcopy` estar definida dentro do módulo `copy`, foi necessário importar esse último para ter acesso à essa função.

Conforme o Código 7.18 indica, `import` é a cláusula da linguagem que permite a um módulo importar outros módulos. Temos as seguintes variações de uso da cláusula import:

- `import <modulo>`: Realiza a importação do módulo denominado `<modulo>`. Todas as definições executadas em `<modulo>` estarão disponíveis em um espaço de nomes (*namespace*) também denominado `<modulo>`. Pelo fato de executar as definições de `<modulo>` em um espaço de nome próprio, não há risco dessas definições sobrescreverem as definições do módulo importador. O Código 8.1 ilustra esse tipo de uso de `import` no prompt de comando, onde o módulo `math`, que traz definições matemáticas e trigonométricas, é utilizado para calcular o logaritmo de 5 na base 10:

```
1 >>> pi = 3.1415
2 >>> import math
3 >>> v = math.log10(5)
4 >>> v
5 0.6989700043360189
6 >>> math.pi
7 3.141592653589793
8 >>> pi
9 3.1415
```

Código 8.1: importação do módulo *math*.

Na linha 1 do Código 8.1, definimos uma variável denominada `pi`. Na linha 2, realizamos a importação do módulo `math`, que será então executado e terá todas as suas definições disponíveis no espaço de nomes de mesmo nome do módulo, isto é, `math`. Na linha 3, usamos a função `log10` de `math`, cuja finalidade é calcular e retornar o logaritmo do número recebido como argumento de entrada na base 10. É curioso notar que, no módulo `math`, também há a definição de uma variável denominada `pi` com um número maior de casas decimais. Mas pelo fato dessa definição ser realizada dentro do espaço de nomes `math`, essa definição de `pi` não sobrescreve nossa definição de `pi` feita na linha 1, como pode ser constatado nas linhas 6-9. Assim, temos então duas variáveis distintas no contexto em questão: a primeira delas é a nossa variável `pi`, e a segunda, `math.pi`, definida no espaço de nomes de `math`. Observe que para acessar qualquer elemento do espaço de nomes, será necessário escrever `<espaço de nomes>.<elemento>`. Assim, um elemento definido no espaço de nomes `math` será acessado escrevendo-se `math.<elemento>`, conforme fizemos `math.log10` e `math.pi`. Em geral, pode-se consultar a listagem de elementos definidos em módulo passando-se seu espaço de nomes para a função `help` no prompt de comando:

```
1 >>> import math
2 >>> help(math)  #imprime ajuda do módulo math
```

- `import <modulo> as <espaço de nome>`: Realiza a importação de `<modulo>` renomeando o espaço de nomes para `<espaço de nome>`. Assim, os elementos do módulo importado poderão ser acessados escrevendo-se `<espaço de nome>.<elemento>`. É comum realizar essa forma de importação para abreviar o espaço de nomes gerado, facilitando assim a escrita do código. No exemplo a seguir, importamos o módulo `math` renomeando seu espaço de nomes para `ma`. Assim, acessamos a variável `pi` escrevendo `ma.pi`:

```
1 >>> import math as ma
2 >>> ma.pi
3 3.141592653589793
```

- `import <modulo1>, <modulo2>, ... , <modulon>`: Importa diversos módulos de uma só vez em uma única linha. Será criado um espaço de nome para cada módulo importado. Por exemplo, a linha seguinte importa os módulos `os`, `sys` e `pickle` de uma só vez:

```
1 import os, sys, pickle
```

O módulo `os` disponibiliza definições a nível de sistema operacional, enquanto `sys` traz definições sobre o interpretador Python em sí. Por fim, `pickle` é um módulo com definições para serialização e deserialização de objetos, que discutiremos em mais detalhes na Seção **??** (Capítulo **??**). Também é possível renomear os espaços de nomes gerados.

- `from <modulo> import <elemento>`: Importa apenas `<elemento>` de `<modulo>`. Demais definições de `<modulo>` não estarão disponíveis para uso. Essa forma de importação não gera um espaço de nomes em separado do módulo sendo importando, o que traz o risco das definições do módulo sendo importado sobrescreverem as do módulo importador.

```
1 >>> pi = 7
2 >>> from math import pi
3 >>> pi
4 3.141592653589793
5 >>> log10(5)
6 Traceback (most recent call last):
7 File "<stdin>", line 1, in <module>
8 NameError: name 'log10' is not defined
```

Código 8.2: importação de elemento do módulo *math*.

O exemplo 8.2 ilustra esse comportamento. Observe que há a definição de uma variável `pi` na linha 1. A importação na linha 2 sobrescreve o valor essa variável, como pode ser constatado nas linhas 3-4, pois a atribuição de `pi` feita em `math` afetará a nossa variável definida na linha 1 devido ao fato de não ser gerado um espaço de nomes em separado para o módulo sendo importado. Note ainda que obtemos um erro na execução da linha 5 ao tentarmos utilizar a função `log10`, pois, apesar de estar definida no módulo `math`, a mesma não foi disponibilizada devido a forma de importação utilizada na linha 2.

- `from <modulo> import <elemento1>, <elemento2>, ..., <elementon>`: Importa apenas `<elemento1>`, `<elemento2>`, ..., `<elementon>` de `modulo`. É como a forma anterior, porém trazendo diversos elementos de uma só vez do módulo sendo importado. Essa forma de importação também não gera um espaço de nomes em separado o módulo sendo importando, o que traz o risco das definições do módulo sendo importado sobrescreverem as do módulo importador. No exemplo a seguir, usamos essa forma para importar os elementos `pi`, `log10` e `exp` (função que calcula exponencial de um número) do módulo `math`:

```
1 >>> from math import pi, log10, exp
2 >>> pi
3 3.141592653589793
4 >>> log10(5)
5 0.6989700043360189
6 >>> exp(3)
7 20.085536923187668
```

- `from <modulo> import *`: Importa todos os elementos de `<modulo>`, mas sem geração de um espaço de nomes próprio para o módulo importado. Novamente, com essa forma de importação, há o risco de sobrescrita das

definições do módulo importador. No exemplo a seguir, usamos essa forma de importação para trazer todos os elementos do módulo `math`. Note que acessamos esses elementos diretamente por seu nome, uma vez que não foi gerado espaço de nomes para `math`:

```
>>> from math import *
>>> pi
3.141592653589793
>>> log10(5)
0.6989700043360189
```

Apesar de haverem variações quanto ao seu uso, a filosofia em geral é a mesma. Quando se importa um módulo, todo o seu conteúdo é executado do início ao fim, mesmo nos casos onde apenas um subconjunto de elementos do módulo será disponibilizado. De modo a potencializar a utilidade da importação de módulos, é muito comum que os módulos sendo importados apenas tragam definições como variáveis, funções, classes e exceções para serem disponibilizadas aos módulos importadores. No entanto, os módulos sendo importados podem trazer qualquer código válido na linguagem Python, e este código também será executado no momento da importação. É válido frisar que a cláusula `import` de Python é ligeiramente diferente da cláusula `include` presente nas linguagens C e C++. Enquanto `include` apenas faz inclusão textual de um arquivo C/C++ em outro, a cláusula `import` de Python provoca a importação em tempo de execução, o que significa que `import` faz um módulo ser executado do início ao fim no ponto em que for encontrada pelo interpretador. É possível, por exemplo, importar um módulo apenas se uma determinada condição for verdadeira, colocando a instrução de importação dentro de um bloco subordinado a uma cláusula `if`.

É válido frisar que qualquer módulo com código Python pode ser importado por outro módulo, o que inclui módulos criados pelo usuário e módulos de pacotes prontos que podem ser baixados na internet e instalados junto aos módulos da biblioteca padrão. Para consultar os módulos disponíveis em uma instalação Python, pode-se usar a função a função `help` passando-se uma string com o conteúdo `"modules"` no prompt:

```
>>> help("modules")
```

Para que a importação de um módulo ocorra com sucesso, é necessário que o interpretador Python possa encontrar o módulo sendo importado. Por padrão, o interpretador buscará um módulo de acordo com a seguinte ordem:

1. No diretório do módulo importador;

2. Nos diretórios definidos na variável de ambiente de sistema `PYTHONPATH` (se estiver configurada);

3. Nos diretórios onde as bibliotecas padrão e pacotes adicionais estão instalados;

4. Nos diretórios definidos nos arquivos texto *.pth*, que definem um diretório de busca por linha. O interpretador buscará nesses diretórios por módulos sendo importados

Todos esses diretórios são incluídos na variável `path` disponível no módulo `sys`. Alterar essa variável também é um modo de alterar os diretórios onde o interpretador buscará pelos módulos sendo importados.

8.1 Funções matemáticas com o módulo *math*

`math` é um módulo da biblioteca padrão do Python que traz a definição de funções matemáticas e trigonométricas recorrentes. O Código 8.3 traz um exemplo de programa que recebe o valor de ângulo em graus, o converte para radianos e calcula seus respectivos seno, cosseno e tangente. Para a conversão do ângulo de graus para radianos e os cálculos trigonométricos, foram utilizadas as funções `radians` (conversão de graus para radianos), `sin` (cálculo de seno), `cos` (cálculo de cosseno) e `tan` (cálculo de tangente), todas definidas dentro do módulo `math`.

```
1  #programa que lê um ângulo em graus, converte para radianos
2  #e calcula seno, cosseno e tangente
3  import math
4
5  anguloGraus = float( input("Entre com o angulo em graus: ") )
6  anguloRadianos = math.radians(anguloGraus)
7
8  seno = math.sin(anguloRadianos)
9  cosseno = math.cos(anguloRadianos)
10 tangente = math.tan(anguloRadianos)
11
12 print("Angulo em radianos:", anguloRadianos)
13 print("Seno:", seno)
14 print("Cosseno:", cosseno)
15 print("Tangente:", tangente)
```

Código 8.3: uso das funções *radians*, *sin*, *cos* e *tan* do módulo *math*.

A seguir, um exemplo de execução do Código 8.3:

```
Entre com o angulo em graus: 45
Angulo em radianos: 0.7853981633974483
Seno: 0.7071067811865475
Cosseno: 0.7071067811865476
Tangente: 0.9999999999999999
```

8.2 Geração de números aleatórios com o módulo *random*

Em algumas situações, é necessário construir programas que precisam obter números de forma aleatória. Na realidade, a rigor, teríamos que dizer que os números são obtidos de modo pseudo-aleatório, pois, pelo fato do computador ser uma máquina determinística, não é possível fazer um artefato de *software* que gere números verdadeiramente de forma aleatória, pois uma mesma entrada à um algoritmo deve sempre produzir uma mesma saída, e não uma saída aleatória. Esse entrave é contornado através do uso de algoritmos especializados que, a partir de uma determinada entrada, denominada como *semente de geração*,

geram uma sequência de valores, simulando assim a geração de número aleatórios. Pelo fato da mesma semente fornecida a um determinado algoritmo de geração produzir sempre a mesma sequência de valores, dizemos que a geração é pseudo-aleatória.

O módulo `random` traz definições úteis para a geração de números pseudo-aleatórios em Python. Nesse módulo, é definido um novo tipo de dado (classe), denominado `Random` que nos permite utilizar algoritmos sofisticados para geração de números pseudo-aleatórios segundo diversas distribuições estatísticas. Os principais métodos da classe `Random` são:

- `seed(valor)`: utiliza `valor` como semente de geração, isto é, como o parâmetro que determinará a sequência de números gerada;
- `random()`: gera um número real (`float`) pseudo-aleatório no intervalo [0 1) e o retorna. Note que o intervalo é fechado em 0 e aberto em 1, o que significa que 0 pode ser gerado, mas 1 não. A grosso modo, pode-se dizer que o método "sorteia" um número real no intervalo descrito.
- `randint(a, b)`: gera um número inteiro (`int`) pseudo-aleatório no intervalo `[a b]` e o retorna. Note que tanto `a` quanto `b` podem ser gerados. A grosso modo, pode-se dizer que o método "sorteia" um número inteiro no intervalo descrito.
- `choice(sequencia)`: escolhe pseudo-aleatoriamente um elemento de `sequencia` e o retorna. `sequencia` pode ser um objeto sequencial qualquer, isto é, um objeto que carregue consigo a ideia de que abriga outros objetos dentro de si, como strings, listas, tuplas, dicionários, conjuntos e etc. Após a execução do método, `sequencia` permanece inalterada.

Existem diversos outros métodos presentes na classe `Random`. Para consultar a listagem completa, pode-se usar a função `help`, no prompt de comando:

```
>>> import random
>>> help(random)
>>> help(random.Random)
```

```
#código que gera dois números aleatórios no intervalo [0 100)
import random
gerador = random.Random()
gerador.seed(1997)
num1 = 100 * gerador.random()
num2 = 100 * gerador.random()

print("Primeiro aleatório:", num1)
print("Segundo aleatório:", num2)
```

Código 8.4: geração dois números pseudo-aleatórios no intervalo [0 e 100).

O Código 8.4 traz um exemplo onde geramos dois números reais pseudo-aleatórios no intervalo [0 100). Na linha 2, importamos o módulo `random`. Na linha 3, geramos um objeto da classe `Random` e atribuímos à variável `gerador`. Na linha 4, fornecemos ao nosso objeto `Random` o valor `1997` como semente de geração. Os números pseudo-aleatórios são gerados nas linhas 5 e 6, e impressos nas linhas 8 e 9. A execução do Código 8.4 gerou a seguinte saída:

```
Primeiro aleatório: 76.67370131695601
Segundo aleatório: 23.545984228400663
```

O fato de fornecemos uma constante (1997) como semente de geração na linha 4 fará com que o Código 8.4 sempre gere os mesmos dois números aleatórios em todas as vezes em que for executado, pois a semente de geração é o que define a sequência de números obtida (faça o teste para verificar). A classe `Random` nos dá a opção de não fornecer uma semente de geração. Nesse caso, será gerada automaticamente uma semente para o objeto gerador, e a classe tentará sempre gerar uma semente diferente a cada criação de um objeto `Random`. Isso significa que, se retirarmos a linha 4, cada execução do Código 8.4 poderia fornecer, em tese, sempre dois número aleatórios diferentes (faça o teste para verificar). Idealmente, a semente de geração de números pseudo-aleatórios deve ser fornecida apenas uma única vez para o objeto gerador antes da geração do primeiro número pseudo-aleatório a cada execução (com raríssimas exceções). Um erro comum de principiantes é acreditar que deve fornecer uma semente de geração antes da geração de cada número pseudo-aleatório no lugar de fornecer a semente de geração apenas uma única vez.

Para gerar um número real pseudo-aleatório no intervalo [-20 60), podemos fazer:

```
1 #código que gera número aleatório no intervalo [-20 60)
2 import random
3 gerador = random.Random()
4 numero = -20 + (60 + 20)*gerador.random()
```

Código 8.5: geração de número pseudo-aleatório no intervalo [-20 60).

Note, que, para gerar um número pseudo-aleatório no intervalo `[a b)`, podemos fazer `a + (b-a)*gerador.random()`, onde `gerador` é um objeto do tipo `Random`. Note que o Código 8.5 não fornece semente de geração explicitamente, o que faz que com cada execução do mesmo possa produzir um número diferente. Uma possível saída para a execução desse código seria:

```
Número gerado: -9.832504004984708
```

```
1  #código que simula o lançamento de uma moeda viciada que, ao ser
2  #lançada, tem 30% de chance de dar cara e 70% de dar coroa
3  import random
4
5  numLancamentos = 10000
6  probabilidadeCara = 0.3
7
8  ncaras = 0
9  ncoroas = 0
10
11 gerador = random.Random()
12
13 for k in range(0, numLancamentos):
14     aleatorio = gerador.random()
15     if aleatorio < probabilidadeCara:
16         ncaras += 1
17     else:
18         ncoroas += 1
19
20 print("Numero de caras:", ncaras)
21 print("Numero de coroas:", ncoroas)
```

Código 8.6: simulação de lançamentos consecutivos de uma moeda viciada.

O Código 8.6 simula 10000 lançamentos de uma moeda viciada, que, ao ser lançada, possui 30% de probabilidade de dar cara e 70% de probabilidade de dar coroa. Nas linhas 5 e 6, definimos as variáveis que atuarão como parâmetros do programa, `numLancamentos` e `probabilidadeCara` , que são, respectivamente, o número de lançamentos da moeda e a probabilidade da moeda dar cara ao ser lançada. Note que, a rigor, não precisaríamos ter declarado essas variáveis, podendo então usar seus valores diretamente como constantes ao longo do programa. Todavia, apontamos que a forma adotada é mais elegante de se programar, pois torna o código mais legível (especialmente para aqueles não o desenvolveram). Além do mais, se torna mais fácil alterar algum dos parâmetros do programa. Por exemplo, para realizar uma nova simulação com outra probabilidade para a moeda dar cara, já se sabe que é suficiente alterar apenas a linha 6. Se não houvesse a definição na linha 6, teríamos que observar todo o código procurando pelos locais onde a probabilidade 0.3 foi utilizada, o que pode ser muito mais propenso a enganos.

As linhas 8 e 9 do Código 8.6 inicializam contadores para o número de vezes em que a moeda deu cara e o número de vezes em que a moeda deu coroa nas simulações de lançamento. Na linha 11, é instanciado um objeto `Random` para a geração de número pseudo-aleatórios. Para cada simulação de lançamento da moeda, sortearemos um número no intervalo [0 1). Se um número sorteado for menor que 0.3 (30%), declaramos que, em nossa simulação, a face sorteada foi cara. Caso contrário, consideramos que a face sorteada foi coroa. O laço nas linhas 13-18 é responsável por repetir a simulação dos lançamentos. Observe a geração do número pseudo-aleatório que definirá a face da moeda (linha 14) e o teste nas linhas 15-18 sobre o número gerado para a definição da face da moeda. Por fim, nas linhas 20 e 21 são impressos os números de lançamentos em que a moeda deu cara e coroa, respectivamente. Um exemplo de execução do Código 8.6 é mostrado a seguir. Pelo fato de não ajustarmos a semente de geração explicitamente, cada execução deste código pode produzir um resultado diferente.

```
Numero de caras: 2972
Numero de coroas: 7028
```

8.3 Funções de tempo com o módulo *time*

O módulo `time` traz definições relativas ao tempo. É útil, por exemplo, em aplicações onde seja necessário manipular dados temporais, medir tempo de execução de tarefas, paralisar a execução do programa por um determinado período de tempo, dentre outras operações. O módulo `time` define um tipo (classe) para armazenamento de data/hora denominada `struct_time` que separa os diversos campos de data/hora, tornando assim a leitura mais fácil para seres humanos. Dessa forma, um objeto da classe `struct_time` possui os seguintes atributos:

- `tm_year`: ano, por exemplo, 2001;
- `tm_mon`: mês do ano, no intervalo entre 1 e 12;
- `tm_mday`: dia do mês, no intervalo entre 1 e 31;
- `tm_hour`: horas, no intervalo entre 0 e 23;
- `tm_min`: minutos, no intervalo entre 0 e 59;
- `tm_sec`: segundos, no intervalo entre 0 e 61 (os valores 60 e 61 são segundos bissextos);
- `tm_gmtoff`: deslocamento em relação ao fuso horário de referência *Tempo Universal Coordenado* (UTC);
- `tm_wday`: dia da semana, no intervalo entre 0 e 6. A segunda-feira é considerada como o dia 0;
- `tm_yday`: dia do ano, no intervalo entre 1 e 366;
- `tm_isdst`: 1 se o horário de verão estiver em vigor, 0, se não estiver, e -1 se essa informação é desconhecida;
- `tm_zone`: abreviação do nome da zona de tempo;

Podemos construir um objeto `struct_time` passando os primeiros nove argumentos em ordem dentro de alguma sequência para o construtor. Assim, para representar a hora 17:30:45 do dia 3 de agosto de 1997, podemos fazer:

```
1 >>> import time
2 >>> horario = time.struct_time((1997, 8, 3, 17, 30, 45, 0,0,0))
3 >>> horario
4 time.struct_time(tm_year=1997, tm_mon=8, tm_mday=3, tm_hour=17,
      tm_min=30, tm_sec=45, tm_wday=0, tm_yday=0, tm_isdst=0)
```

É oportuno ressaltar que Python oferece o módulo `datetime` que define tipos de dados de manipulação mais simples para lidar com datas/horas. Por essa razão, muitos desenvolvedores tem preferido utilizar as classes definidas nesse último módulo para lidar com dados dessa natureza no lugar de utilizar

`struct_time` (consulte a documentação de `datetime` para mais detalhes). Todavia, o módulo `time` ainda possui funções com grande utilidade que valem a pena de serem conhecidas.

Algumas das funções mais comumente utilizadas no módulo `time` são:

- `clock()`: retorna um número relacionado ao tempo de processamento (CPU) do programa. A diferença entre dois resultados de `clock` nos dá uma medida do tempo de processamento (em segundos) que um conjunto de operações demandou para ser executado.

```
1  #programa que gera 1000000 números pseudo-aleatórios
2  #e mede o tempo de processamento
3  import random
4  import time
5
6  clockInicio = time.clock()  ###ponto de referencia inicial
7
8  nRepeticoes = 1000000
9  gerador = random.Random()
10 for k in range(0, nRepeticoes):
11     gerador.random()
12
13 clockFim = time.clock()      ###ponto de referência final
14
15 tempoExecucao = clockFim - clockInicio
16 print("Tempo de execução:", tempoExecucao)
```

Código 8.7: medição do tempo de processamento da geração de um milhão de números pseudo-aleatórios.

Por exemplo, o Código 8.7 mede o tempo de processamento para a geração de um milhão de números pseudo-aleatórios. Nas linhas 3 e 4 importamos os módulos `random` e `time`, respectivamente. Nas linhas 6 e 13 armazenamos o resultado de chamadas a função `clock`. Na linha 15, fazemos a diferença entre o último valor de `clock` e o primeiro. O valor dessa diferença será a medida (em segundos) do tempo de processamento das instruções executadas dentre as duas chamadas à função `clock`. Assim, a variável `tempoExecucao` terá uma medida do tempo em que os processadores da máquina gastaram executando as instruções entre as linhas 7 e 12. Esse valor é então impresso na linha 16. Um exemplo de execução do Código 8.7 é fornecido a seguir (o resultado pode variar dependendo do *hardware* e das condições do sistema):

```
Tempo de execução: 0.15823900000000002
```

É importante frisar que esse valor medido com as chamadas à função `clock` corresponde ao tempo estrito em que o processo ficou sendo executado em algum processador da máquina, e, por essa razão, esse tempo pode diferir do tempo real medido com o relógio. Por exemplo, suponha uma situação bastante simplista em que uma máquina tem um processador para executar simultaneamente dois programas que estejam fazendo cálculos intensos. Como o processador só consegue executar uma tarefa por vez, ele precisará ficar repetidamente alternando o trabalho entre os dois processos para que o usuário tenha a impressão de que ambos estão sendo feitos

simultaneamente. Em outras palavras, o processador executará um pouco do primeiro processo, para então executar um pouco do segundo processo, para então voltar ao primeiro processo e executar mais um pouco, e depois voltar ao segundo e executar mais um pouco, e assim sucessivamente, até que os processos estejam finalizados. Vamos supor que, no nosso exemplo, os processos foram simultaneamente iniciados e finalizados, e que cada um demandou cerca de 5 segundos de trabalho do processador. Nessa situação, foi preciso de cerca de 10 segundos para executar ambos os processos de forma alternada. Assim, se o usuário medir no relógio o tempo em que transcorreu entre o início e o fim do processo 1, por exemplo, ele verá que transcorreu 10 segundos. No entanto, se o tempo for medido com a função `clock`, apenas o tempo efetivo em processamento será contado, e assim, essa medição terá o valor de 5 segundos.

Também é possível que o tempo medido com chamadas à `clock` seja menor que o tempo real transcorrido. Se, por exemplo, um programa for executado em dois processadores simultaneamente, há uma tendência (teórica) de que o programa termine sua execução em cerca de metade do tempo que levaria em um único processador. Assim, um programa que necessite de 30 segundos de processamento poderá ser executado em cerca de 15 segundos do tempo real se o trabalho for muito bem dividido entre dois processadores e não houver outros programas requisitando seu uso. Nessa situação, a medição com `clock` contará o tempo de processamento em todos os processadores, que, no caso é de 30 segundos, mas só terão transcorridos 15 segundos do tempo real. É válido apontar que para que um programa seja executado em mais de um processador, é necessário usar instruções explícitas que permitam essa execução em paralelo, ou ao menos chamar alguma função que utilize essas instruções de paralelismo em seu processamento. Por essa razão, os programas que desenvolvemos até aqui são configurados, por padrão, para executar em apenas um processador da máquina, ainda que haja um número maior de processadores disponíveis. É oportuno observar também que criar programas para execução em múltiplos processadores é acentuadamente mais complicado do que programar para um processador só. Assim, o aprendizado das técnicas desse tipo de programação deve ser realizado após o desenvolvedor apresentar bom domínio da programação tradicional para processador único.

- `time()`: retorna o número de segundos transcorridos desde 01/01/1970 (marco zero Unix) até o momento atual, de acordo com a hora definida pelo sistema operacional. Com a função `time` é possível, por exemplo, medir o tempo real para o programa executar um conjunto de instruções. Por exemplo, podemos adaptar o Código 8.7 para medir o tempo real transcorrido na geração dos números simplesmente substituindo as chamadas à função `clock` por chamadas à função `time`, conforme realizado no Código 8.8:

```
1  #programa que gera 1000000 números pseudo-aleatórios
2  #e mede o tempo real
3  import random
4  import time
5
6  tempoInicio = time.time()  ###ponto de referencia inicial
7
8  nRepeticoes = 1000000
9  gerador = random.Random()
10 for k in range(0, nRepeticoes):
11     gerador.random()
12
13 tempoFim = time.time()      ###ponto de referência final
14
15 tempoExecucao = tempoFim - tempoInicio
16 print("Tempo de execução:", tempoExecucao)
```

Código 8.8: medição do tempo real da geração de um milhão de números pseudo-aleatórios.

Observe que, do ponto de vista lógico, a única diferença entre os Códigos 8.7 e 8.8 é o uso, neste último, da função `time` no lugar de `clock` nas linhas 6 e 13, que fará com que o Código 8.8 meça o tempo real para a execução do bloco de instruções nas linhas 7-12. Um exemplo de execução do Código 8.8 é fornecido a seguir (o resultado pode variar dependendo do *hardware* e das condições do sistema):

```
Tempo de execução: 0.15218186378479004
```

A função `time` também pode ser usada para fornecer sementes para a geração de números pseudo-aleatórios:

```
1 >>> import random
2 >>> import time
3 >>> gerador = random.Random()
4 >>> gerador.seed( time.time() )
```

Note que o uso da função `time` para essa finalidade possui a vantagem de, a, cada segundo, fornecer um valor diferente como semente de geração. Na maioria das situações práticas, isto é suficiente para garantir que, a cada execução do código, sempre haverá uma semente de geração diferente, conduzindo assim à sequências de números pseudo-aleatórios potencialmente diferentes e evitando que um programa sorteie sempre os mesmos números a cada execução, se assim for desejado.

- `localtime(segundos)`: converte o número de segundos desde 01/01/1970 (representado na variável `segundos`) para um objeto do tipo `struct_time` representando data/hora local e o retorna. Se o argumento de entrada `segundos` não for fornecido, a função usará o número de segundos relativo a hora corrente do sistema. Exemplo de uso:

```
>>> import time
>>> hora = time.localtime(1000000000)
>>> hora
time.struct_time(tm_year=2001, tm_mon=9, tm_mday=8, tm_hour
    =22, tm_min=46, tm_sec=40, tm_wday=5, tm_yday=251,
    tm_isdst=0)
```

```
>>> import time
>>> hora = time.localtime()  #usará a hora corrente do
    sistema
>>> hora
time.struct_time(tm_year=2019, tm_mon=7, tm_mday=24,
    tm_hour=19, tm_min=39, tm_sec=15, tm_wday=2, tm_yday
    =205, tm_isdst=0)
```

- `gmttime(segundos)`: funciona como `localtime`, mas gerando um objeto `struct_time` que representa data/hora em UTC (Tempo Universal Coordenado).

- `mktime(struct_time)`: realiza a operação reversa a `localtime` e `gmttime`, isto é, recebe um objeto `struct_time` e retorna o número de segundos transcorridos desde 01/01/1970 até a data/hora representada no objeto.

- `sleep(segundos)`: faz o programa "dormir" pela quantidade de segundos determinada em `segundos`, isto é, torna o programa inativo (paralisado) pela quantidade de tempo indicada. Note que, ao usar esta função, o programa so poderá executar a próxima operação após ser "acordado", o que ocorrerá automaticamente em algum momento após o tempo solicitado passar.

  ```
  >>> import time
  >>> time.sleep(15)   #coloca o programa para dormir por 15
      segundos. Só após esse tempo, o programa estará apto a
      executar a próxima operação.
  ```

  A primeira vista, pode parecer estranho colocar um programa para "dormir", pois em geral, deseja-se que suas operações sejam executadas o mais rápido possível. Todavia, existem situações práticas onde dar uma pausa na execução de um programa pode ser bastante útil. Suponhamos, por exemplo, que precisamos desenvolver um programa que buscará dados em um servidor através da internet. Ao tentar buscar os dados desejados, podemos obter uma mensagem de erro de conexão, caso o servidor esteja fora do ar. No lugar de deixar o programa continuamente em *loop* tentando se conectar ao servidor, podemos, por exemplo, colocar o programa para dormir por alguns minutos antes de tentar a próxima conexão. Desse modo, o programa espera um intervalo de tempo na expectativa de que o servidor volte a entrar no ar. Assim, o programa não permanece ativo gastando processamento em nossa máquina com seguidas tentativas de conexão que possuiriam alto potencial de fracassar.

É válido observar que, além da forma de medir o tempo de execução de um programa discutida aqui, muitos programadores Python tem recorrido a um módulo denominado `timeit` para obter um levantamento estatístico do tempo

de execução a partir de diversas execuções do mesmo código. Consulte a ajuda do módulo para mais detalhes.

8.4 Importando seus próprios módulos

Além de importar módulos da biblioteca padrão e de pacotes de terceiros, podemos também importar nossos próprios módulos para reaproveitar nossas definições. Vamos supor, por exemplo, que precisamos de um programa que calcule o fatorial de um número, e optamos por criar uma função, conforme realizado no Código 8.9.

```
1  #programa que calcula fatorial de um numero
2  def calculaFatorial(n):
3      r = 1
4      for k in range(n, 1, -1):
5          r = r*k
6      return r
7
8
9  #lendo entrada do usuário e chamando a função
10 num = int( input("Entre com o numero para obter o fatorial: ") )
11 vfatorial = calculaFatorial(num)
12 print("Fatorial do numero: ", vfatorial)
```

Código 8.9: cálculo do fatorial de um número (arquivo *fatorial.py*).

Vamos supor que salvamos o Código 8.9 em um arquivo de nome *fatorial.py*. Vamos supor também, que, em um momento posterior, precisamos fazer um programa que calcula $C_{n,p}$, isto é, a combinação de n elementos tomados p a p, que é dada por:

$$C_{n,p} = \frac{n!}{p!(n-p)!}$$

Note que, para o cálculo de $C_{n,p}$, precisamos calcular fatoriais. Assim, seria útil aproveitar a definição da função `calculaFatorial` para agilizar o desenvolvimento. De modo facilitar a importação, salvamos um novo código no mesmo diretório onde *fatorial.py* foi salvo. No entanto, ao importar o arquivo fatorial.py, conforme o Código 8.10:

```
1  import fatorial          #Note que não é necessário incluir a
       extensão do arquivo
```

Código 8.10: importa fatorial.py.

, a execução da linha 1 do Código 8.10 fará com que **todo** o Código 8.9 seja executado, incluindo as linhas 9-12, que não nos interessam. Assim, como resultado, a execução do Código 8.10 acabará por pedir um número ao usuário para cálculo de fatorial, conforme abaixo:

```
Entre com o numero para obter o fatorial:
```

Podemos evitar essa inconveniência alterando o Código importado, isto é o Código 8.9. Para isso, podemos nos valer de uma variável especial do interpretador Python, denominada `__name__`. Essa variável é criada automaticamente pelo interpretador da linguagem a cada execução de um módulo. A variável `__name__` apontará para uma string com o conteúdo `"__main__"` se o módulo

estiver sendo executado diretamente como módulo principal, isto é, se não estiver sendo importado a partir de algum outro módulo. Se o módulo estiver sendo importado a partir de outro, então a variável `__name__` terá, como valor, uma string com o nome usado na importação (No caso da importação na linha 1 do Código 8.10, `__name__` terá o valor `"fatorial"`). Assim, podemos acrescentar um teste na linha 8 do Código 8.9 para só permitir a execução das linhas 9-12 se o Código 8.9 estiver sendo executado como módulo principal, gerado assim o Código 8.11:

```
1  #programa que calcula fatorial de um numero
2  def calculaFatorial(n):
3      r = 1
4      for k in range(n, 1, -1):
5          r = r*k
6      return r
7
8  if __name__ == "__main__":  #este if só permite que o código
       subordinado seja executado se este módulo for executado como
       módulo principal
9      #lendo entrada do usuário e chamando a função
10     num= int(input("Entre com o numero para obter o fatorial: "))
11     vfatorial = calculaFatorial(num)
12     print("Fatorial do numero: ", vfatorial)
```

Código 8.11: cálculo do fatorial de um número - versão para importação (arquivo *fatorial.py*).

Note que a diferença entre os Códigos 8.9 e 8.11 é o teste acrescentado nesse último na linha 8 sobre a variável `__name__`. Esse teste fará com que o bloco subordinado nas linhas 9-12 só seja executado se este módulo **não** estiver sendo importado a partir de outro. Vamos supor agora que sobrescrevemos o arquivo *fatorial.py* com a nova versão mostrada no Código 8.11. Agora, a execução do Código 8.10 não imprimirá nada na tela, pois o módulo *fatorial.py* sendo importado na linha 1, apenas definirá a função `calculaFatorial` e nada mais, pois a execução do teste do `if` na linha 8 (Código 8.11) resultará em `False`. Tendo realizada essa oportuna alteração em *fatorial.py*, podemos então criar um arquivo chamado *combinacao.py*, no mesmo diretório de *fatorial.py* com o conteúdo do Código 8.12:

```
1  #programa que calcula combinacao de n elementos tomados p a p
2  import fatorial
3
4  def calculaCombinacao(n, p):
5      fatn = fatorial.calculaFatorial(n)
6      fatp = fatorial.calculaFatorial(p)
7      fatn_p = fatorial.calculaFatorial(n-p)
8
9      return fatn//(fatp*fatn_p)
10
11
12 if __name__ == "__main__":
13
14     n = int( input("Entre com o valor de n: ") )
15     p = int( input("Entre com o valor de p: ") )
16
17     combnp = calculaCombinacao(n, p)
18
19     print("Combinacao n,p: ", combnp)
```

Código 8.12: cálculo da combinacao de n elementos tomados p a p.

Note que, no Código 8.12, após importar o módulo *fatorial.py* (Código 8.11), usamos a função `calculaFatorial` definida neste último módulo nas linhas 5-7 para auxiliar o cálculo da combinação de elementos. Observe ainda que, na linha 12, fizemos novamente o teste sobre a variável `__name__` para só permitir que as linhas 14-19 só sejam executadas se o Código 8.12 estiver sendo executado como módulo principal. Dessa forma, o Código 8.12 fica adequado para ser importado em outros módulos, caso se deseje aproveitar a função `calculaCombinacao` futuramente. Um exemplo de execução do Código 8.12 é mostrado a seguir:

```
Entre com o valor de n: 10
Entre com o valor de p: 4
Combinacao n,p:  210
```

Deste ponto em diante no texto, procuraremos sempre colocar o teste sobre a variável `__name__` em nossos códigos.

8.5 Exercícios

1. Um dado viciado de 4 faces possui as seguintes probabilidades de sorteio para cada uma de suas faces:

Face	Probabilidade
1	10%
2	20%
3	30%
4	40%

   Faça um programa que simule o lançamento do dado obedecendo as suas probabilidades de sorteio.

   Exemplo:

```
Face sorteada: 4
```

2. Faça um programa que calcule o fatorial de um número a partir de três formas diferentes:

   (a) Através da definição de uma função recursiva

   (b) Através da definição de uma função não recursiva

   (c) Através do uso da função `factorial` disponível no módulo `math`

   Juntamente com o valor do fatorial, seu programa deve imprimir o tempo de processamento (CPU) gasto ao chamar cada uma das três funções (note que a medição do tempo é apenas nas chamadas às funções, e não nas definições das mesmas).

   Exemplo:

```
Entre com o numero: 30

Fatorial: 265252859812191058636308480000000
Tempo para cálculo recursivo: 4.1000000000013e-05
Tempo para cálculo não recursivo: 1.99999999999922e-05
Tempo para cáluclo com math.factorial: 1.10000000000110e-05
```

3. O valete de ferro é um jogo de cartas bastante popular na Bachatóvia. Neste jogo, um determinado jogador inicia com 7 cartas aleatórias na mão. Dado um baralho de 52 cartas com os 4 naipes padrão, escreva um programa em Python que sorteia aleatoriamente as 7 cartas que um jogador receberá. Note que todas as cartas possuem a mesma probabilidade de serem sorteadas, que a mesma carta não pode ser sorteada duas vezes, e você só deve sortear o jogo para um único jogador. Seu programa deve imprimir número e naipe das 7 cartas sorteadas.

   Exemplo:

```
Cartas sorteadas:
Carta 1: 6 de Copas
Carta 2: Valete de Copas
Carta 3: 9 de Espadas
Carta 4: 5 de Ouros
Carta 5: 5 de Espadas
Carta 6: 2 de Ouros
Carta 7: Dama de Espadas
```

4. Escreva um programa que sorteie jogos para a Mega Sena. Seu programa deverá solicitar o número de jogos a serem feitos e o número de dezenas em cada jogo. Note que os números sorteados são inteiros no intervalo [1 60]. Observe que cada dezena só pode aparecer uma única vez em cada jogo.

   Exemplo:

```
Entre com o numero de jogos: 3
Entre com o numero de dezenas em cada jogo: 6

dezenas do jogo 1: 17 19 30 33 41 49
dezenas do jogo 2: 1 8 19 35 39 48
dezenas do jogo 3: 6 14 23 35 47 58
```

5. Tia Liliane, professora do Jardim Escola Pentelho Feliz, se orgulha do seu revolucionário sistema de avaliação que incentiva seus alunos a dar o melhor de si nos estudos. Funciona da seguinte forma: periodicamente, tia Liliane aplica testes na turma para avaliar a assimilação do conteúdo ensinado, com pontuação entre 0 e 10 pontos. Nesse contexto, a cada aula, um aluno que tirou nota inferior a 5,0 no último teste é sorteado para ir ao quadro resolver um desafio. Sua tarefa então é bastante simples: fazer um programa em Python que lê as notas de um teste específico aplicado na turma e sorteia o aluno que deve ir ao quadro resolver o desafio. Note que apenas alunos com nota inferior a 5,0 podem ser sorteados. Se nenhum aluno possuir nota inferior a 5,0, seu programa deve informar essa condição e não sortear ninguém.

   Exemplos:

```
Entre com o numero de alunos: 5

Nota do aluno 1: 6
Nota do aluno 2: 2.5
Nota do aluno 3: 4.9
Nota do aluno 4: 3.2
Nota do aluno 5: 7.3

O aluno 3 foi sorteado para ir ao quadro na proxima aula.
```

```
Entre com o numero de alunos: 3

Nota do aluno 1: 9.4
Nota do aluno 2: 7.0
Nota do aluno 3: 5.3

Nenhum aluno obteve nota inferior a 5.0.
```

6. Faça um programa que simule o lançamento de um dado viciado de n faces, onde a probabilidade da face k ser sorteada é

$$p(k) = \frac{k}{\frac{n(n+1)}{2}} \tag{8.1}$$

   . Por exemplo, em um dado de 5 ($n = 5$) faces, a probabilidade de sair a face 1 é $p(1) = \frac{1}{15}$, ao passo que a probabilidade de sair a face 2 é $p(2) = \frac{2}{15}$, e assim sucessivamente. Seu programa deve iniciar perguntando o número

de faces do dado, e, em seguida, simular o lançamento de acordo com as probabilidades definidas pela expressão (8.1).

Exemplo:

```
Entre com o número de faces: 5
Face sorteada: 4
```

Índice Remissivo

__name__, 154

algoritmo, 1
Ambiente de Desenvolvimento Integrado, 9
and, 35
argumento de entrada, 93
argumento de saída, 93
arquivo fonte, 2
array, 113
ASCII, 71
atribuição, 11
atribuição ampliada, 28
atribuição múltipla, 133

biblioteca padrão, 141
bloco de repetição, 51
bool, 16
break, 58

código de byte, 5
código fonte, 2
cópia profunda, 132
cópia rasa, 130
chr, 79
comentário, 23
compilador, 3
compilador de código de byte, 4
complex, 17
concatenção de listas, 115
concatenção de strings, 73
constantes de barra invertida, 71
contêiner, 113
continue, 59
conversão de tipos, 19
copy (módulo), 132

datetime (módulo), 149
deepcopy, 132
def, 95
del, 116

docstring, 69

elif, 44
else, 42
encapsulamento, 102
engenharia reversa, 3
entidade recursiva, 105
escopo local, 98
espaço de nomes, 141

float, 17
for, 55
formatação de string, 73
fracionamento de sequências, 77
função, 93

geração de números aleatórios, 145

help, 85

IDE, 9
IDLE, 9
if, 39
import, 141
importação de módulos, 141
in, 38, 74, 116
indentação, 41
infinito, 63
input, 21
int, 16
interpretador, 4
interpretador de código de byte, 5
iteração, 52

laço infinito, 53
len, 73
linguagem compilada, 2
linguagem híbrida, 4
linguagem interpretada, 4
list, 113, 116
list.append, 118
list.count, 118

list.extend, 118
list.index, 119
list.insert, 119
list.remove, 119
list.reverse, 119
list.sort, 119
listas, 113

métodos de classe, 84
métodos de lista, 118
métodos de string, 85
módulo, 141
mais infinito, 63
marco zero Unix, 151
math (módulo), 141
math.cos, 145
math.exp, 143
math.factorial, 157
math.log10, 142
math.pi, 142
math.radians, 145
math.sin, 145
math.tan, 145
memória RAM, 10
menos infinito, 63

números de Fibonacci, 106
namespace, 141
NAN, 63
nomes de variáveis, 15
None, 18
not, 35
Not a Number, 63
not in, 39, 74, 116

objeto imutável, 14
operadores de comparação, 36
operadores lógicos, 33
or, 33
ord, 78
os, 143

pickle, 143
polimorfismo, 100
print, 20
processamento de texto, 124
programação orientada a objetos, 84
promt Python, 9

radiciação, 27
Random (classe), 146
random (módulo), 146
Random.choice, 146
Random.randint, 146
Random.random, 146
Random.seed, 146
range, 56
raw string, 71
raw_input, 21
relação de recorrência, 105
repetição de strings, 75
return, 96

semente de geração de números pseudo-aleatórios, 145
sequência de escape, 71
sequência de Fibonacci, 106
str, 18, 69
str.count, 85
str.isaplha, 86
str.isdigit, 86
str.islower, 86
str.isupper, 86
str.join, 86
str.lower, 87
str.replace, 87
str.split, 87
str.substring, 85
str.upper, 84, 87
string, 18, 69
string bruta, 71
struct_time, 149
struct_time.tm_gmtoff, 149
struct_time.tm_hour, 149
struct_time.tm_isdst, 149
struct_time.tm_mday, 149
struct_time.tm_min, 149
struct_time.tm_mon, 149
struct_time.tm_sec, 149
struct_time.tm_wday, 149
struct_time.tm_yday, 149
struct_time.tm_year, 149
struct_time.tm_zone, 149
sum, 124
sys, 143

tabela verdade, 33
tempo de processamento, 150
teste de repetição, 51
time, módulo, 149
time.clock, 150

time.gmttime, 153
time.localtime, 152
time.mktime, 153
time.sleep, 153
time.time, 151
Timestamp Unix, 151
tupla, 18, 132
tuple, 18, 133
type, 18

unicode, 72
Unix epoch, 151

variáveis, 11
variável global, 103

while, 51