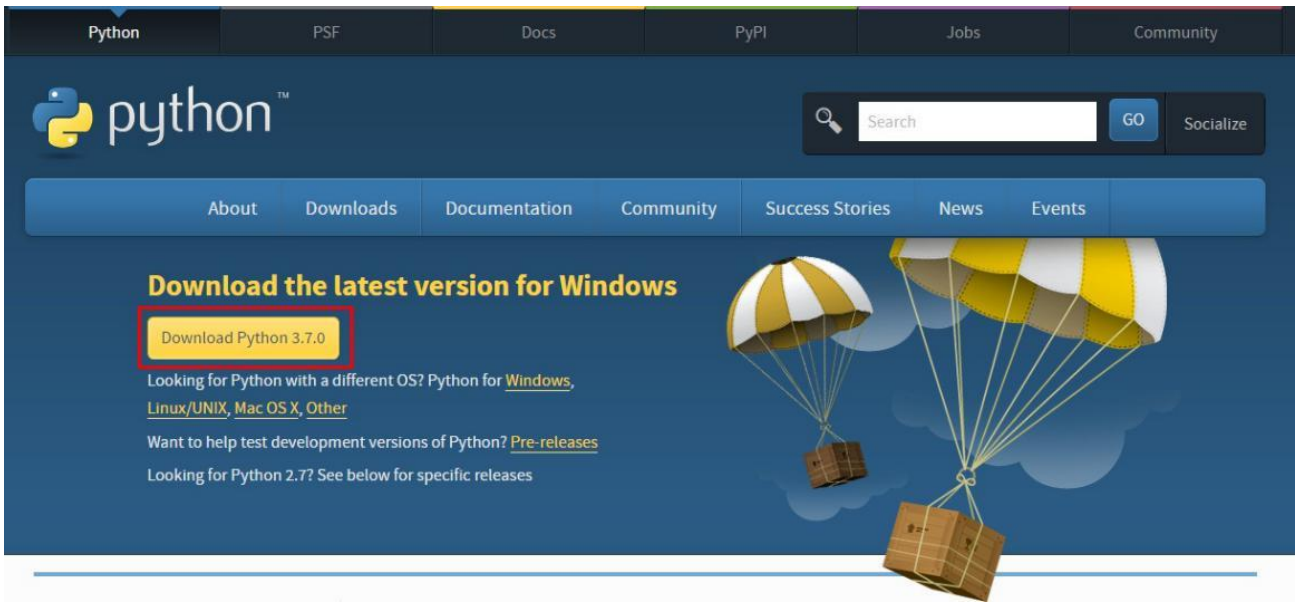
VERSÁTIL
tecnologia em informação

INSTALAÇÃO

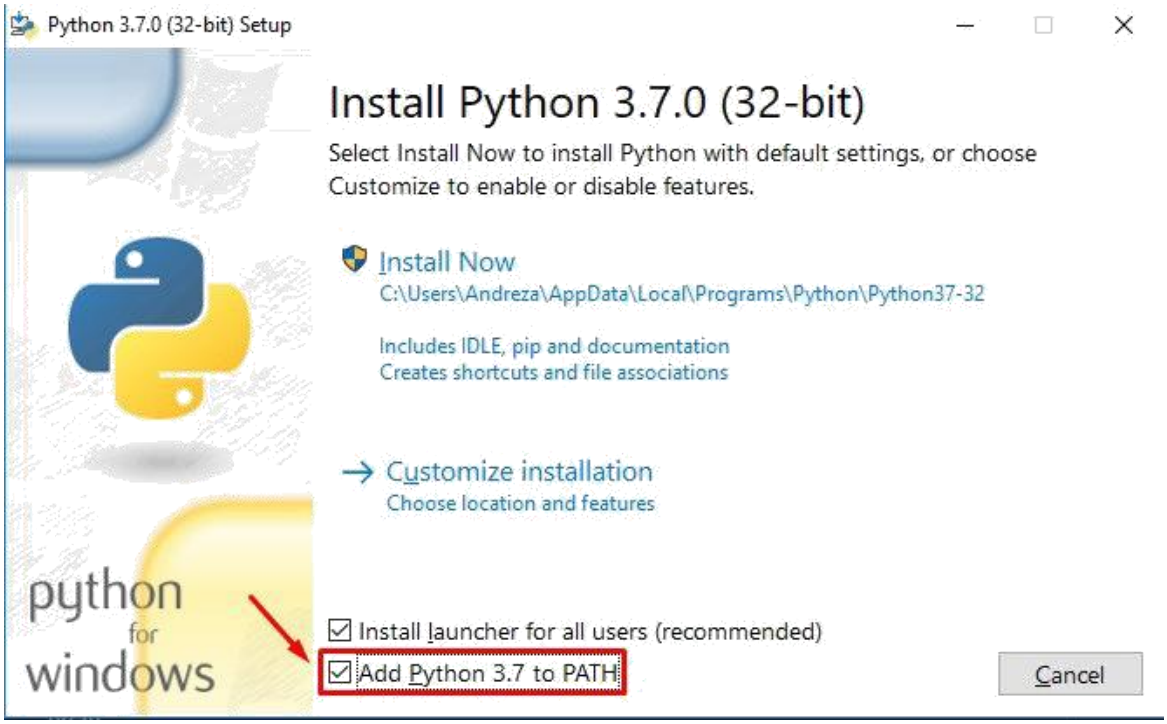
O Python já vem instalado nos sistemas Linux e Mac OS mas será necessário fazer o download da última versão (Python 3.6) para acompanhar a apostila. O Python não vem instalado por padrão no Windows e o download deverá ser feito no site https://www.python.org/ além de algumas configurações extras.

INSTALANDO O PYTHON NO WINDOWS

O primeiro passo é acessar o site do Python: https://www.python.org/. Na sessão de **Downloads** já será disponibilizado o instalador específico do Windows automaticamente, portanto é só baixar o Python3, na sua versão mais atual.

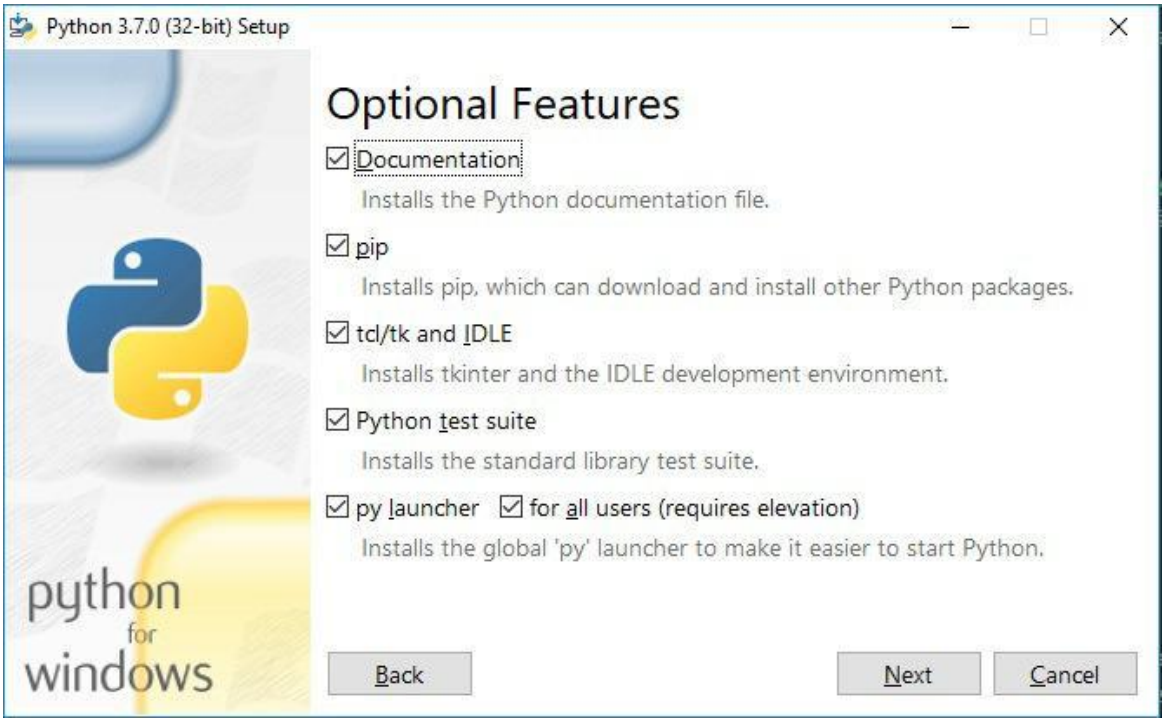


Após o download ser finalizado, abra-o e na primeira tela marque a opção Add Python 3.X to PATH. Essa opção é importante para conseguirmos executar o Python dentro do Prompt de Comando do Windows. Caso você não tenha marcado esta opção, terá que configurar a variável de ambiente no Windows de forma manual.

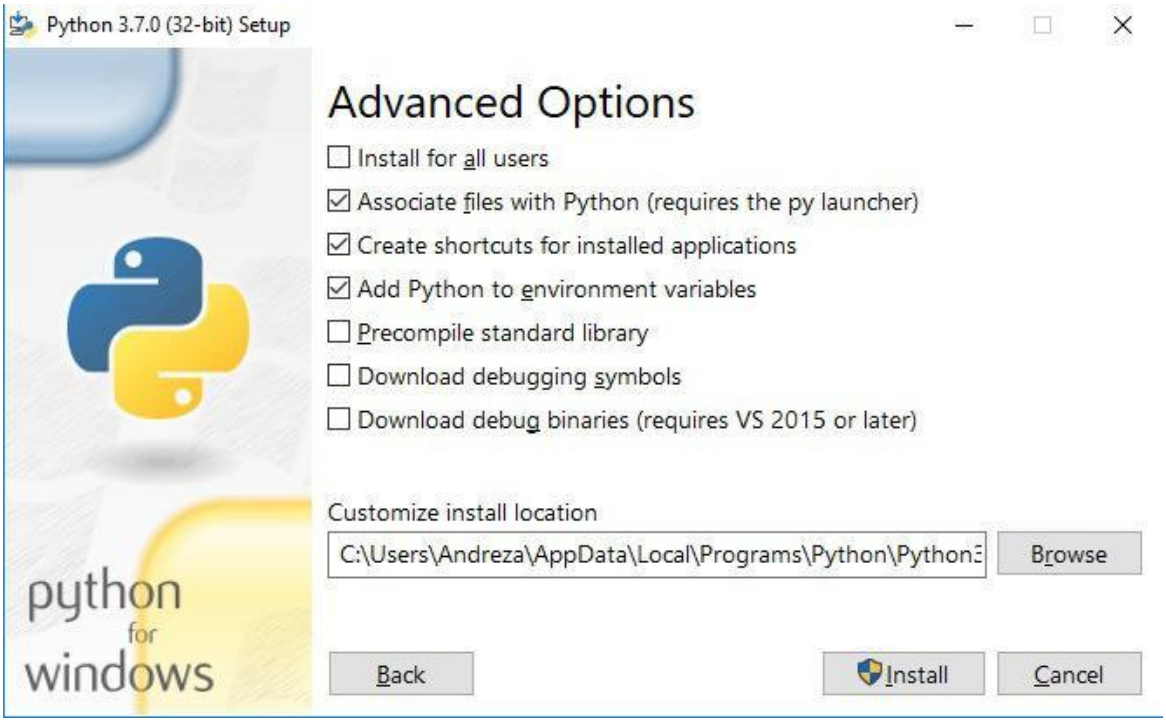


Selecione a instalação customizada somente para ver a instalação com mais detalhes.

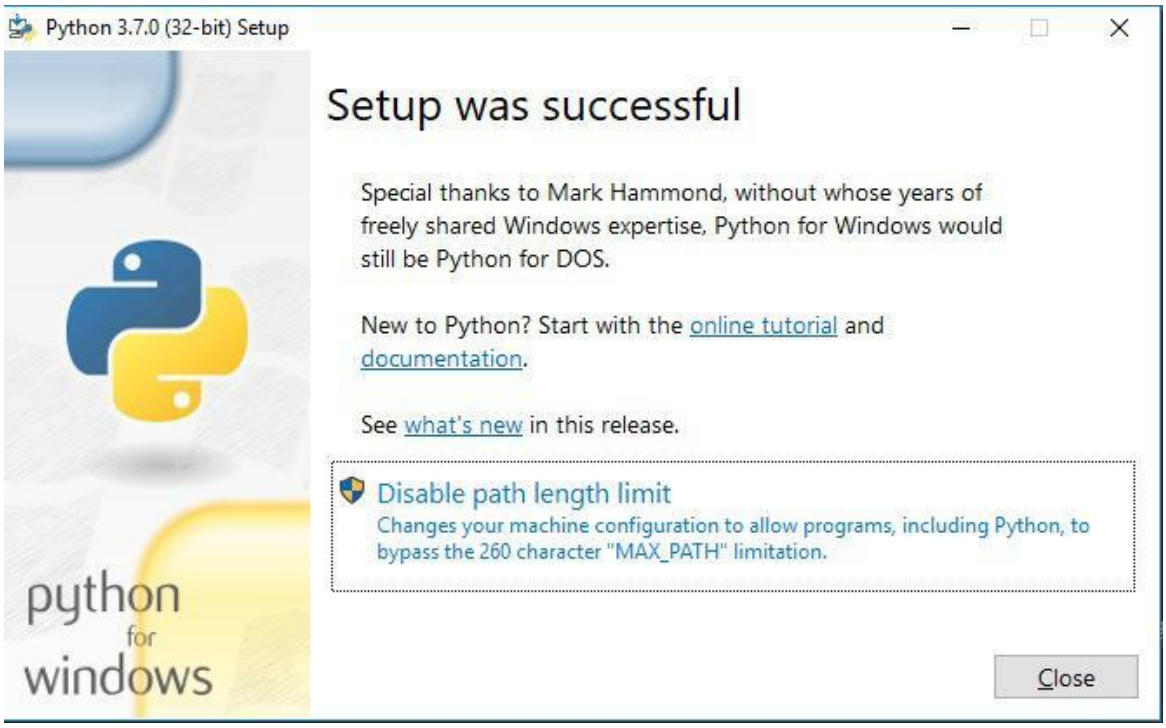
Na tela seguinte são as features opcionais, se certifique que o gerenciador de pacotes pip esteja selecionado, ele que permite instalar pacotes e bibliotecas no Python. Clique em Next para dar seguimento na instalação.



Já na terceira tela, deixe tudo como está, mas se atente ao diretório de instalação do Python, para caso queira procurar o executável ou algo que envolva o seu diretório.



Por fim, basta clicar em Install e aguardar o término da instalação.



Terminada a instalação, teste se o Python foi instalado corretamente. Abra o Prompt de Comando e execute:

```
python -V
```

OBS: Para que funcione corretamente é necessário que seja no Prompt de Comando e não em algum programa Git Bash instalado em sua máquina. E o comando python -V é importante que esteja com o V com letra maiúscula.

Esse comando imprime a versão do Python instalada no Windows. Se a versão for impressa, significa que o Python foi instalado corretamente. Agora, rode o comando python **:**

```
python
```

Assim você terá acesso ao console do próprio Python, conseguindo assim utilizá-lo.

INSTALANDO O PYTHON NO LINUX

Os sistemas operacionais baseados no Debian já possuem o Python3 pré-instalado. Verifique se o seu sistema já possui o Python3 instalado executando o seguinte comando no terminal:

```
python3 -V
```

OBS: O comando python3 -V é importante que esteja com o V com letra maiúscula.

Este comando retorna a versão do Python3 instalada. Se você ainda não o tiver instalado, digite os seguintes comandos no terminal:

```
sudo apt-get update
```

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

INSTALANDO O PYTHON NO MACOS

A maneira mais fácil de instalar o Python3 no MacOS é utilizando o Homebrew **.**

Com o Homebrew instalado, abra o terminal e digite os seguintes comandos:

```
brew update
```

```
brew install python3
```

OUTRAS FORMAS DE UTILIZAR O PYTHON

Podemos rodar o Python diretamente do seu próprio Prompt.

Podemos procurar pelo Python na caixa de pesquisa do Windows e abri-lo, assim o seu console próprio será aberto. Uma outra forma é abrir a IDLE do Python, que se parece muito com o console mas vem com um menu que possui algumas opções extras.

CAPÍTULO 01

1. Sobre o Python

Se você trabalha muito com computadores, acabará encontrando alguma tarefa que gostaria de automatizar. Por exemplo, você pode querer fazer busca-e-troca em um grande número de arquivos de texto, ou renomear e reorganizar um monte de arquivos de fotos de uma maneira complicada. Talvez você gostaria de escrever um pequeno banco de dados personalizado, ou um aplicativo GUI especializado, ou um jogo simples.

Se você é um desenvolvedor de software profissional, pode ter que trabalhar com várias bibliotecas C/C++/Java, mas o tradicional ciclo escrever/compilar/testar/recompilar é muito lento. Talvez você esteja escrevendo um conjunto de testes para uma biblioteca e está achando tedioso codificar os testes. Ou talvez você tenha escrito um programa que poderia utilizar uma linguagem de extensão, e você não quer conceber e implementar toda uma nova linguagem para sua aplicação.

Python é a linguagem para você.

Você poderia escrever um script de shell do Unix ou arquivos em lote do Windows para algumas dessas tarefas, mas os scripts de shell são melhores para mover arquivos e alterar dados de texto, não sendo adequados para jogos ou aplicativos GUI. Você poderia escrever um programa C / C ++ / Java, mas pode levar muito tempo de desenvolvimento para obter até mesmo um programa de primeiro rascunho. Python é mais simples de usar, está disponível nos sistemas operacionais Windows, macOS e Unix e o ajudará a fazer o trabalho mais rapidamente.

Python é fácil de usar, sem deixar de ser uma linguagem de programação de verdade, oferecendo muito mais estruturação e suporte para programas extensos do que shell scripts ou arquivos de lote oferecem. Por outro lado, Python também oferece melhor verificação de erros do que C, e por ser uma linguagem de *muito alto nível*, ela possui tipos nativos de alto nível, tais como dicionários e vetores (arrays) flexíveis. Devido ao suporte nativo a uma variedade de tipos de dados, Python é aplicável a um domínio de problemas muito mais vasto do que Awk ou até mesmo Perl, ainda assim muitas tarefas são pelo menos tão fáceis em Python quanto nessas linguagens.

Python permite que você organize seu programa em módulos que podem ser reutilizados em outros programas escritos em Python. A linguagem provê uma vasta coleção de módulos que podem ser utilizados como base para sua aplicação — ou como exemplos para estudo e aprofundamento. Alguns desses módulos implementam manipulação de arquivos, chamadas do sistema, sockets, e até mesmo acesso a bibliotecas de construção de interfaces gráficas, como Tk.

Python é uma linguagem interpretada, por isso você pode economizar um tempo considerável durante o desenvolvimento, uma vez que não há necessidade de compilação e vinculação (*linking*). O interpretador pode ser usado interativamente, o que torna fácil experimentar diversas características da linguagem, escrever programas “descartáveis”, ou testar funções em um desenvolvimento debaixo para cima (*bottom-up*). É também uma útil calculadora de mesa.

Python permite a escrita de programas compactos e legíveis. Programas escritos em Python são tipicamente mais curtos do que seus equivalentes em C, C++ ou Java, por diversas razões:

- os tipos de alto nível permitem que você expresse operações complexas em um único comando;
- a definição de bloco é feita por indentação ao invés de marcadores de início e fim de bloco;
- não há necessidade de declaração de variáveis ou parâmetros formais;

Python é *extensível*: se você sabe como programar em C, é fácil adicionar funções ou módulos diretamente no interpretador, seja para desempenhar operações críticas em máxima velocidade, ou para vincular programas Python a bibliotecas que só estejam disponíveis em formato binário (como uma biblioteca gráfica de terceiros). Uma vez que você tenha sido fisgado, você pode vincular o interpretador Python a uma aplicação escrita em C e utilizá-la como linguagem de comandos ou extensão para esta aplicação.

A propósito, a linguagem foi batizada a partir do famoso programa da BBC “Monty Python’s Flying Circus” e não tem nada a ver com répteis. Fazer referências a citações do programa na documentação não é só permitido, como também é encorajado!

Agora que você está entusiasmado com Python, vai querer conhecê-la com mais detalhes. Partindo do princípio que a melhor maneira de aprender uma linguagem é usando-a, você está agora convidado a fazê-lo com este tutorial.

No próximo capítulo, a mecânica de utilização do interpretador é explicada. Essa informação, ainda que mundana, é essencial para a experimentação dos exemplos apresentados mais tarde.

O resto do tutorial introduz diversos aspectos do sistema e linguagem Python por intermédio de exemplos. Serão abordadas expressões simples, comandos, tipos, funções e módulos. Finalmente, serão explicados alguns conceitos avançados como exceções e classes definidas pelo usuário.

CAPÍTULO 02

2. Utilizando o interpretador Python

2.1. Chamando o interpretador

O interpretador Python é frequentemente instalado como `/usr/local/bin/python3.10` nas máquinas onde está disponível; adicionando `/usr/local/bin` ao caminho de busca da shell de seu Unix torna-se possível iniciá-lo digitando o comando:

```
python3.10
```

no shell. 1 Considerando que a escolha do diretório onde o interpretador está é uma opção de instalação, outros locais são possíveis; verifique com seu guru local de Python ou administrador do sistema local. (Por exemplo, `/usr/local/python` é um local alternativo popular.)

Em máquinas Windows onde você instalou Python a partir da Microsoft Store, o comando `python3.10` estará disponível. Se você tem o lançador py.exe instalado, você pode usar o comando `py`. Veja Excursus: Configurando variáveis de ambiente para outras maneiras de executar o Python.

Digitando um caractere de fim-de-arquivo (`Control-D` no Unix, `Control-Z` no Windows) diretamente no prompt força o interpretador a sair com status de saída zero. Se isso não funcionar, você pode sair do interpretador digitando o seguinte comando: `quit()`.

Os recursos de edição de linha do interpretador incluem edição interativa, substituição de histórico e complemento de código, em sistemas com suporte à biblioteca GNU Readline. Talvez a verificação mais rápida para ver se o suporte à edição de linha de comando está disponível é digitando `Control-P` no primeiro prompt oferecido pelo Python. Se for emitido um bipe, você terá a edição da linha de comando; veja Apêndice Edição de entrada interativa e substituição de histórico para uma introdução às combinações. Se nada acontecer, ou se `^P` aparecer na tela, a edição da linha de comando não está disponível; você só poderá usar backspace para remover caracteres da linha atual.

O interpretador trabalha de forma semelhante a uma shell de Unix: quando chamado com a saída padrão conectada a um console de terminal, ele lê e executa comandos interativamente; quando chamado com um nome de arquivo como argumento, ou com redirecionamento da entrada padrão para ler um arquivo, o interpretador lê e executa o *script* contido no arquivo.

Uma segunda forma de iniciar o interpretador é `python -c comando [arg] ...`, que executa uma ou mais instruções especificadas na posição *comando*, analogamente à opção de shell `-c`. Considerando que

comandos Python frequentemente têm espaços em branco (ou outros caracteres que são especiais para a shell) é aconselhável que o *comando* esteja dentro de aspas duplas.

Alguns módulos Python são também úteis como scripts. Estes podem ser chamados usando `python -m módulo [arg] ...` que executa o arquivo fonte do *módulo* como se você tivesse digitado seu caminho completo na linha de comando.

Quando um arquivo de script é utilizado, às vezes é útil executá-lo e logo em seguida entrar em modo interativo. Isto pode ser feito acrescentando o argumento `-i` antes do nome do script.

Todas as opções de linha de comando são descritas em Linha de comando e ambiente.

2.2. Passagem de argumentos

Quando são de conhecimento do interpretador, o nome do script e demais argumentos da linha de comando da shell são acessíveis ao próprio script através da variável `argv` do módulo `sys`. Pode-se acessar essa lista executando `import sys`. Essa lista tem sempre ao menos um elemento; quando nenhum script ou argumento for passado para o interpretador, `sys.argv[0]` será uma string vazia. Quando o nome do script for `'-'` (significando entrada padrão), o conteúdo de `sys.argv[0]` será `'-'`. Quando for utilizado `-c` *comando*, `sys.argv[0]` conterá `'-c'`. Quando for utilizado `-m` *módulo*, `sys.argv[0]` conterá o caminho completo do módulo localizado. Opções especificadas após `-c` *comando* ou `-m` *módulo* não serão consumidas pelo interpretador mas deixadas em `sys.argv` para serem tratadas pelo comando ou módulo.

2.3. Modo interativo

Quando os comandos são lidos a partir do console, diz-se que o interpretador está em modo interativo. Nesse modo ele solicita um próximo comando através do *prompt primário*, tipicamente três sinais de maior (`>>>`); para linhas de continuação do comando atual, o *prompt secundário* padrão é formado por três pontos (`...`). O interpretador exibe uma mensagem de boas vindas, informando seu número de versão e um aviso de copyright antes de exibir o primeiro prompt:

```
$ python3.10
Python 3.10 (default, June 4 2019, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Linhas de continuação são necessárias em construções multi-linha. Como exemplo, dê uma olhada nesse comando `if`:

```
>>> existe_10_tipos_de_pessoas = True
>>> if existe_10_tipos_de_pessoas:
...     print("As que entendem e não entendem binário.”)
...
As que entendem e não entendem binário.
```

Para mais informações sobre o modo interativo, veja Modo interativo.

CAPÍTULO 03

3. Uma introdução informal ao Python

Nos exemplos seguintes, pode-se distinguir entrada e saída pela presença ou ausência dos prompts (>>> e …): para repetir o exemplo, você deve digitar tudo após o prompt, quando o mesmo aparece; linhas que não começarem com um prompt são na verdade as saídas geradas pelo interpretador. Observe que quando aparece uma linha contendo apenas o prompt secundário você deve digitar uma linha em branco; é assim que se encerra um comando de múltiplas linhas.

Você pode alternar a exibição de prompts e saída clicando em >>> no canto superior direito de uma caixa de exemplo. Se você ocultar os prompts e a saída de um exemplo, poderá copiar e colar facilmente as linhas de entrada em seu interpretador.

Muitos exemplos neste manual, mesmo aqueles inscritos na linha de comando interativa, incluem comentários. Comentários em Python começam com o caractere cerquilha '#' e estende até o final da linha. Um comentário pode aparecer no inicio da linha ou após espaço em branco ou código, mas não dentro de uma string literal. O caracterer cerquilha dentro de uma string literal é apenas uma cerquilha. Como os comentários são para esclarecer o código e não são interpretados pelo Python, eles podem ser omitidos ao digitar exemplos.

Alguns exemplos:

```
# este é o primeiro comentário
spam = 1  # e este é o segundo comentário
          # ... e agora um terceiro!
text = "# Isso não é um comentário porque está entre aspas."
```

3.1. Usando Python como uma calculadora

Vamos experimentar alguns comandos simples em Python. Inicie o interpretador e aguarde o prompt primário, `>>>`. (Não deve demorar muito.)

3.1.1. Números

O interpretador funciona como uma calculadora bem simples: você pode digitar uma expressão e o resultado será apresentado. A sintaxe de expressões é a usual: operadores `+`, `-`, `*` e `/` funcionam da mesma forma que em outras linguagens tradicionais (por exemplo, Pascal ou C); parênteses (`()`) podem ser usados para agrupar expressões. Por exemplo:

```
>>> 2 + 2
```

```
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # divisão sempre retorna um número de ponto flutuante
1.6
```

Os números inteiros (ex. `2`, `4`, `20`) são do tipo `int`, aqueles com parte fracionária (ex. `5.0`, `1.6`) são do tipo `float`. Veremos mais sobre tipos numéricos posteriormente neste tutorial.

Divisão (`/`) sempre retorna ponto flutuante (float). Para fazer uma divisão pelo piso e receber um inteiro como resultado (descartando a parte fracionária) você pode usar o operador `//`; para calcular o resto você pode usar o `%`:

```
>>> 17 / 3  # clássica divisão que retorna um float
5.666666666666667
>>>
>>> 17 // 3  # a divisão do piso descarta a parte fracionária
5
>>> 17 % 3  # o operador % retorna o restante da divisão
2
>>> 5 * 3 + 2  # quociente mínimo * divisor + resto
17
```

Com Python, é possível usar o operador `**` para calcular potências [1]:

```
>>> 5 ** 2  # 5 quadrado
25
>>> 2 ** 7  # 2 elevado a 7
128
```

O sinal de igual (`'='`) é usado para atribuir um valor a uma variável. Depois de uma atribuição, nenhum resultado é exibido antes do próximo prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Se uma variável não é “definida” (não tem um valor atribuído), tentar utilizá-la gerará um erro:

```
>>> n  # tente acessar uma variável indefinida
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined.
```

Há suporte completo para ponto flutuante (*float*); operadores com operandos de diferentes tipos convertem o inteiro para ponto flutuante:

```
>>> 4 * 3.75 - 1
14.0
```

No modo interativo, o valor da última expressão exibida é atribuída a variável `_`. Assim, ao utilizar Python como uma calculadora, fica mais fácil prosseguir com os cálculos, por exemplo:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Essa variável especial deve ser tratada como *somente para leitura* pelo usuário. Nunca lhe atribua explicitamente um valor — do contrário, estaria criando uma outra variável (homônima) independente, que mascararia a variável especial com seu comportamento mágico.

Além de `int` e `float`, o Python suporta outros tipos de números, tais como `Decimal` e `Fraction`. O Python também possui suporte nativo a números complexos, e usa os sufixos `j` ou `J` para indicar a parte imaginária (por exemplo, `3+5j`).

3.1.2. Strings

Além de números, Python também pode manipular strings (sequências de caracteres), que podem ser expressas de diversas formas. Elas podem ser delimitadas por aspas simples (`'...'`) ou duplas (`"..."`) e teremos o mesmo resultado 2. `\` pode ser usada para escapar aspas:

```
>>> 'spam eggs'  # aspas simples
'spam eggs'
>>> 'doesn\'t'  # use \' para escapar das aspas simples...
"doesn't"
>>> "doesn't"  # ...ou use aspas duplas
```

```
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Na interpretação interativa, a string de saída é delimitada com aspas e caracteres especiais são escapados com barras invertidas. Embora isso possa às vezes parecer diferente da entrada (as aspas podem mudar), as duas strings são equivalentes. A string é delimitada com aspas duplas se a string contiver uma única aspa simples e nenhuma aspa dupla, caso contrário, ela é delimitada com aspas simples. A função `print()` produz uma saída mais legível, ao omitir as aspas e ao imprimir caracteres escapados e especiais:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n significa nova linha
>>> s  # sem print(), \n é incluído na saída
'First line.\nSecond line.'
>>> print(s)  # com print(), \n produz uma nova linha
First line.
Second line.
```

Se não quiseres que os caracteres sejam precedidos por `\` para serem interpretados como caracteres especiais, poderás usar *strings raw* (N.d.T: “crua” ou sem processamento de caracteres de escape) adicionando um `r` antes da primeira aspa:

```
>>> print('C:\some\n')  # aqui \n significa nova linha!
C:\some
ame
>>> print(r'C:\some\n')  # observe o r antes da citação
C:\some\name
```

As strings literais podem abranger várias linhas. Uma maneira é usar as aspas triplas: `"""..."""` ou `'''...'''`. O fim das linhas é incluído automaticamente na string, mas é possível evitar isso adicionando uma `\` no final. O seguinte exemplo:

```
print("""\
Usage: thingy [OPTIONS]
     -h                        Display this usage message
```

```
     -H hostname               Hostname to connect to
""")
```

produz a seguinte saída (observe que a linha inicial não está incluída):

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Strings podem ser concatenadas (coladas) com o operador `+`, e repetidas com `*`:

```
>>> # 3 vezes 'un', seguido de 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Duas ou mais *strings literais* (ou seja, entre aspas) ao lado da outra são automaticamente concatenados.

```
>>> 'Py' 'thon'
'Python'
```

Esse recurso é particularmente útil quando você quer quebrar strings longas:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined
together.'
```

Isso só funciona com duas strings literais, não com variáveis ou expressões:

```
>>> prefix = 'Py'
>>> prefix 'thon'   # não pode concatenar uma variável e uma
string literal
  File "<stdin>", line 1
    prefix 'thon'
                ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
                   ^
SyntaxError: invalid syntax
```

Se você quiser concatenar variáveis ou uma variável e uma literal, use `+`:

```
>>> prefix + 'thon'
'Python'
```

As strings podem ser *indexadas* (subscritas), com o primeiro caractere como índice 0. Não existe um tipo específico para caracteres; um caractere é simplesmente uma string cujo tamanho é 1:

```
>>> word = 'Python'
>>> word[0]  # caractere na posição 0
'P'
>>> word[5]  # caractere na posição 5
'n'
```

Índices também podem ser números negativos para iniciar a contagem pela direita:

```
>>> word[-1]  # último caractere
'n'
>>> word[-2]  # penúltimo caractere
'o'
>>> word[-6]  # primeiro caractere
'P'
```

Note que dado que -0 é o mesmo que 0, índices negativos começam em -1.

Além da indexação, o *fatiamento* também é permitido. Embora a indexação seja usada para obter caracteres individuais, *fatiar* permite que você obtenha substring:

```
>>> word[0:2]    # caracteres da posição 0 (incluído) a 2 (excluído)
'Py'
>>> word[2:5]    # caracteres da posição 2 (incluídos) a 5 (excluídos)
'tho'
```

Os índices do fatiamento possuem padrões úteis; um primeiro índice omitido padrão é zero, um segundo índice omitido é por padrão o tamanho da string sendo fatiada:

```
>>> word[:2]   # caractere do início até a posição 2 (excluído)
'Py'
>>> word[4:]   # caracteres da posição 4 (incluída) até o final
'on'
>>> word[-2:]  # caracteres do penúltimo (incluído) ao final
```

```
'on'
```

Observe como o início sempre está incluído, e o fim sempre é excluído. Isso garante que `s[:i] + s[i:]` seja sempre igual a `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Uma maneira de lembrar como fatias funcionam é pensar que os índices indicam posições *entre* caracteres, onde a borda esquerda do primeiro caractere é 0. Assim, a borda direita do último caractere de uma string de comprimento *n* tem índice *n*, por exemplo:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

A primeira fileira de números indica a posição dos índices 0…6 na string; a segunda fileira indica a posição dos respectivos índices negativos. Uma fatia de *i* a *j* consiste em todos os caracteres entre as bordas *i* e *j*, respectivamente.

Para índices positivos, o comprimento da fatia é a diferença entre os índices, se ambos estão dentro dos limites da string. Por exemplo, o comprimento de `word[1:3]` é 2.

A tentativa de usar um índice que seja muito grande resultará em um erro:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

No entanto, os índices de fatiamento fora do alcance são tratados graciosamente (N.d.T: o termo original “gracefully” indica robustez no tratamento de erros) quando usados para fatiamento. Um índice maior que o comprimento é trocado pelo comprimento, um limite superior menor que o limite inferior produz uma string vazia:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

As strings do Python não podem ser alteradas — uma string é imutável. Portanto, atribuir a uma posição indexada na sequência resulta em um erro:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Se você precisar de uma string diferente, deverá criar uma nova:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

A função embutida `len()` devolve o comprimento de uma string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Ver também

Tipo sequência de texto — str

As strings são exemplos de *tipos de sequências* e suportam as operações comumente suportadas por esses tipos.

Métodos de string

As strings suportam uma grande quantidade de métodos para transformações básicas e busca.

Literais de string formatados

Strings literais que possuem expressões embutidas.

Sintaxe das strings de formato

Informações sobre formatação de string com o método `str.format()`.

Formatação de String no Formato printf-style

As antigas operações de formatação invocadas quando as strings são o operando esquerdo do operador `%` são descritas com mais detalhes aqui.

ATIVIDADES

1. Faça um Programa que mostre a mensagem "Alo mundo" na tela

2. Faça um Programa que peça um número e então mostre a mensagem *O número informado foi [número]*.

3. Faça um Programa que peça as 4 notas bimestrais e mostre a média.

4. Faça um Programa que peça o raio de um círculo, calcule e mostre sua área.

5. Faça um Programa que peça a temperatura em graus Fahrenheit, transforme e mostre a temperatura em graus Celsius.

a. C = 5 * ((F-32) / 9).

3.1.3. Listas

Python inclui diversas estruturas de dados *compostas*, usadas para agrupar outros valores. A mais versátil é *list* (lista), que pode ser escrita como uma lista de valores (itens) separados por vírgula, entre colchetes. Os valores contidos na lista não precisam ser todos do mesmo tipo.

```
>>> nome_lista = [1, 4, 9, 16, 25]
>>> nome_lista
[1, 4, 9, 16, 25]
```

Como strings (e todos os tipos embutidos de sequência), listas podem ser indexadas e fatiadas:

```
>>> nome_lista[0]  # indexação retorna o item
1
>>> nome_lista[-1]
25
>>> nome_lista[-3:]  # fatiar retorna uma nova lista
[9, 16, 25]
```

Todas as operações de fatiamento devolvem uma nova lista contendo os elementos solicitados. Isso significa que o seguinte fatiamento devolve uma cópia rasa da lista:

```
>>> nome_lista[:]
[1, 4, 9, 16, 25]
```

As listas também suportam operações como concatenação:

```
>>> nome_lista + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Diferentemente de strings, que são imutáveis, listas são mutáveis, ou seja, é possível alterar elementos individuais de uma lista:

```
>>> cubes = [1, 8, 27, 65, 125]  # algo está errado aqui
>>> 4 ** 3  # o cubo de 4 é 64, não 65!
64
>>> cubes[3] = 64  # substitua o valor errado
>>> cubes
[1, 8, 27, 64, 125]
```

Você também pode adicionar novos itens no final da lista, usando o *método* `append()` (estudaremos mais a respeito dos métodos posteriormente):

```
>>> cubes.append(216)  # adicione o cubo de 6
>>> cubes.append(7 ** 3)  # e o cubo de 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Atribuição a fatias também é possível, e isso pode até alterar o tamanho da lista ou remover todos os itens dela:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # substituir alguns valores
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # agora removê-los
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # limpe a lista substituindo todos os elementos por uma
lista vazia
>>> letters[:] = []
>>> letters
[]
```

A função embutida `len()` também se aplica a listas:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

É possível aninhar listas (criar listas contendo outras listas), por exemplo:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. Primeiros passos para a programação

Podemos usar o Python para tarefas mais complicadas do que somar 2+2. Com o loop while, podemos executar um conjunto de instruções, desde que uma condição seja verdadeira.

Por exemplo, podemos escrever o início da sequência de Fibonacci assim:

```
>>> # série Fibonacci:
... # a soma de dois elementos define o próximo
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

Este exemplo introduz diversas características ainda não mencionadas.

- A primeira linha contém uma atribuição múltipla: as variáveis `a` e `b` recebem simultaneamente os novos valores 0 e 1. Na última linha há outro exemplo de atribuição múltipla demonstrando que expressões do lado direito são sempre

avaliadas primeiro, antes da atribuição. As expressões do lado direito são avaliadas da esquerda para a direita.

- O laço de repetição `while` executa enquanto a condição (aqui: `a < 10`) permanece verdadeira. Em Python, como em C, qualquer valor inteiro que não seja zero é considerado verdadeiro; zero é considerado falso. A condição pode também ser uma cadeia de caracteres ou uma lista, ou qualquer sequência; qualquer coisa com um tamanho maior que zero é verdadeiro, enquanto sequências vazias são falsas. O teste usado no exemplo é uma comparação simples. Os operadores padrões de comparação são os mesmos de C: `<` (menor que), `>` (maior que), `==` (igual), `<=` (menor ou igual), `>=` (maior ou igual) e `!=` (diferente).

  A seguir, teremos um exemplo de um loop while que é iniciado com a condição "a <= 10", o que significa que o loop será executado enquanto o valor de "a" for menor ou igual a 10. Dentro do loop, a linha "print(a)" imprime o valor atual de "a" na tela. Em seguida, "a = a + 1" adiciona 1 ao valor de "a". Este processo é repetido até que o valor de "a" seja maior do que 10, e então o loop é finalizado

```
>>> a = 0
>>> while a <= 10:
...     print(a)
        a = a+1

0 1 2 3 4 5 6 7 8 9 10
```

- O *corpo* do laço é *indentado*: indentação em Python é a maneira de agrupar comandos em blocos. No console interativo padrão você terá que digitar tab ou espaços para indentar cada linha. Na prática você vai preparar scripts Python mais complicados em um editor de texto; a maioria dos editores de texto tem facilidades de indentação automática. Quando um comando composto é digitado interativamente, deve ser finalizado por uma linha em branco (já que o interpretador não tem como adivinhar qual é a última linha do comando). Observe que toda linha de um mesmo bloco de comandos deve ter a mesma indentação.
- A função `print()` escreve o valor dos argumentos fornecidos. É diferente de apenas escrever a expressão no interpretador (como fizemos anteriormente nos exemplos da calculadora) pela forma como lida com múltiplos argumentos, quantidades de ponto flutuante e strings. As strings são impressas sem aspas, e um espaço é inserido entre os itens, assim você pode formatar bem o resultado, dessa forma:

```
>>> i = 256*256
>>> print('The value of i is', i)
O valor de i é 65536
```

O argumento *end* pode ser usado para evitar uma nova linha após a saída ou finalizar a saída com uma string diferente:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

ATIVIDADES

1. Faça um Programa que leia um vetor de 5 números inteiros e mostre-os.
2. Faça um programa que leia uma variável numeral de limite, mostrando todos os números que o precedem.

CAPÍTULO 04

4. Mais ferramentas de controle de fluxo

Além do comando `while` recém apresentado, Python tem as estruturas usuais de controle de fluxo conhecidas em outras linguagens, com algumas particularidades.

4.1. Comandos `if e else`

Provavelmente o mais conhecido comando de controle de fluxo é o `if`. Por exemplo:

```
>>> x = int(input("Please enter an integer: ")) # Por favor insira um número inteiro:

>>> if x < 0:
...      x = 0
...      print('Negative changed to zero') # Se 'x' menor que '0', negativos vão passar a ser 0.
... elif x == 0:
...      print('Zero')   # Se x igual a 0, printar 'Zero'
... elif x == 1:
...      print('Single') # Se x igual a 1, printar 'Single'
... else:
...      print('More') # Se nenhuma das condições forem verdadeiras, printar 'More'
...
```

Pode haver zero ou mais partes `elif`, e a parte `else` é opcional. A palavra-chave '`elif`' é uma abreviação para 'else if', e é útil para evitar indentação excessiva. Uma sequência `if` ... `elif` ... `elif` ... substitui os comandos `switch` ou `case`, encontrados em outras linguagens.

Se você está comparando o mesmo valor com várias constantes, ou verificando por tipos ou atributos específicos, você também pode achar a instrução `match` útil. Para mais detalhes veja Instruções match.

4.2. Comandos `for`

O comando `for` em Python é um pouco diferente do que costuma ser em C ou Pascal. Ao invés de sempre iterar sobre uma progressão aritmética de números (como no Pascal), ou permitir ao usuário definir o passo de iteração e a condição de parada (como C), o comando `for` do Python itera sobre os itens de qualquer sequência (seja uma lista ou uma string), na ordem que aparecem na sequência. Por exemplo:

```
>>> # Meça algumas cordas:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Código que modifica uma coleção sobre a qual está iterando pode ser inseguro. No lugar disso, usualmente você deve iterar sobre uma cópia da coleção ou criar uma nova coleção:

```
# Criar uma coleção de amostras
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎':
'active'}

# Estratégia: iterar sobre uma cópia
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Estratégia: Criar uma nova coleção
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

4.3. A função `range()`

Se você precisa iterar sobre sequências numéricas, a função embutida `range()` é a resposta. Ela gera progressões aritméticas:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

O ponto de parada fornecido nunca é incluído na lista; `range(10)` gera uma lista com 10 valores, exatamente os índices válidos para uma sequência de comprimento 10. É possível iniciar o intervalo com outro número, ou alterar a razão da progressão (inclusive com passo negativo):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Para iterar sobre os índices de uma sequência, combine `range()` e `len()` da seguinte forma:

```
>>> a = ['Mary', 'tinha', 'um', 'pequeno', 'cordeirinho']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 tinha
2 um
3 pequeno
4 cordeirinho
```

Na maioria dos casos, porém, é mais conveniente usar a função `enumerate()`, veja Técnicas de iteração.

Uma coisa estranha acontece se você imprime um intervalo:

```
>>> range(10)
range(0, 10)
```

Em muitos aspectos, o objeto retornado pela função `range()` se comporta como se fosse uma lista, mas na verdade não é. É um objeto que retorna os itens sucessivos da sequência desejada quando você itera sobre a mesma, mas na verdade ele não gera a lista, economizando espaço.

Dizemos que um objeto é iterável, isso é, candidato a ser alvo de uma função ou construção que espera alguma coisa capaz de retornar sucessivamente seus elementos um de cada vez. Nós vimos que o comando `for` é um exemplo de construção, enquanto que um exemplo de função que recebe um iterável é `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
```

```
6
```

Mais tarde, veremos mais funções que retornam iteráveis e tomam iteráveis como argumentos. No capítulo Estruturas de dados, iremos discutir em mais detalhes sobre `list()`.

4.4. Comandos `break` e `continue`, nos laços de repetição

Durante uma execução de while, é possível programar um break (comando que para a execução de um determinado bloco de código ou instrução) que acontece se determinada condição for atingida.

Normalmente o uso de break se dá quando colocamos mais de uma condição que, se a instrução do código atingir qualquer uma dessas condições (uma delas), ele para sua execução para que não entre em um loop infinito de repetições. Por exemplo:

```
a = 1
while a < 10:
    print(a)
    a += 1
    if a == 4:
        break

1
2
3
```

Enquanto a variável a for menor que 10, continue imprimindo-a e acrescentando 1 ao seu valor. Mas se em algum momento ela for igual a 4, pare a repetição.

A instrução `continue`, também emprestada da linguagem C, continua com a próxima iteração do laço:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Encontrou um número par ", num)
...         continue
...     print("Encontrou um número ímpar ", num)
...
Encontrou um número par 2
Encontrou um número ímpar 3
Encontrou um número par 4
```

```
Encontrou um número ímpar 5
Encontrou um número par 6
Encontrou um número ímpar 7
Encontrou um número par 8
Encontrou um número ímpar 9
```

4.6. Instruções `match`

Uma instrução de correspondência pega uma expressão e compara seu valor com padrões sucessivos fornecidos como um ou mais blocos de case. Isso é superficialmente semelhante a uma instrução switch em C, Java ou JavaScript (e muitas outras linguagens), mas também pode extrair componentes (elementos de sequência ou atributos de objeto) do valor em variáveis.

A forma mais simples compara um valor de assunto com um ou mais literais:

```
def http_error(status):
    match status:
        case 400:
            return " Pedido ruim "
        case 404:
            return " Não encontrado "
        case 418:
            return " eu sou um bule "
        case _:
            return " Algo está errado com a internet "
```

Observe o último bloco: o “nome da variável” `_` atua como um *curinga* e nunca falha em corresponder. Se nenhum caso corresponder, nenhuma das ramificações será executada.

Você pode combinar vários literais em um único padrão usando `|` (“ou”):

```
case 401 | 403 | 404:
    return " Não permitido "
```

Os padrões podem se parecer com atribuições de desempacotamento e podem ser usados para vincular variáveis:

```
# ponto é uma tupla (x, y)
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
```

```
            print(f"Y={y}")
        case (x, 0):
            print(f"X={x}")
        case (x, y):
            print(f"X={x}, Y={y}")
        case _:
            raise ValueError("Não é um ponto")
```

Estude isso com cuidado! O primeiro padrão tem dois literais e pode ser considerado uma extensão do padrão literal mostrado acima. Mas os próximos dois padrões combinam um literal e uma variável, e a variável *vincula* um valor do assunto (`point`). O quarto padrão captura dois valores, o que o torna conceitualmente semelhante à atribuição de desempacotamento `(x, y) = point`.

ATIVIDADES

1. Faça um programa que peça uma nota, entre zero e dez. Mostre uma mensagem caso o valor seja inválido e continue pedindo até que o usuário informe um valor válido.
2. Utilizando listas faça um programa que faça 5 perguntas para uma pessoa sobre um crime. As perguntas são:
    a. "Telefonou para a vítima?"
    b. "Esteve no local do crime?"
    c. "Mora perto da vítima?"
    d. "Devia para a vítima?"
    e. "Já trabalhou com a vítima?" O programa deve no final emitir uma classificação sobre a participação da pessoa no crime. Se a pessoa responder positivamente a 2 questões ela deve ser classificada

como "Suspeita", entre 3 e 4 como "Cúmplice" e 5 como "Assassino". Caso contrário, ele será classificado como "Inocente".

3. Faça um programa que leia 5 números e informe o maior número.
4. Faça um programa que imprima na tela apenas os números ímpares entre 1 e 50.
5. Numa eleição existem três candidatos. Faça um programa que peça o número total de eleitores. Peça para cada eleitor votar e ao final mostrar o número de votos de cada candidato.

4.7. Definindo funções

Podemos criar uma função que escreve a série de Fibonacci até um limite arbitrário:

```
>>> def fib(n):    # escreva a série de Fibonacci até n
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Agora chame a função que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada `def` inicia a *definição* de uma função. Ela deve ser seguida do nome da função e da lista de parâmetros formais entre parênteses. Os comandos que formam o corpo da função começam na linha seguinte e devem ser indentados.

Opcionalmente, a primeira linha do corpo da função pode ser uma literal string, cujo propósito é documentar a função. Se presente, essa string chama-se *docstring*. (Há mais informação sobre docstrings na seção Strings de documentação.) Existem ferramentas que utilizam docstrings para produzir automaticamente documentação online ou para imprimir, ou ainda, permitir que o usuário navegue interativamente pelo código. É uma boa prática incluir sempre docstrings em suas funções, portanto, tente fazer disso um hábito.

A *execução* de uma função cria uma nova tabela de símbolos para as variáveis locais da função. Mais precisamente, todas as atribuições de variáveis numa função são armazenadas na tabela de símbolos local; referências a variáveis são buscadas primeiro na tabela de símbolos local, em seguida na tabela de símbolos locais de funções delimitadoras ou circundantes, depois na tabela de símbolos global e, finalmente, na tabela de nomes da própria linguagem. Embora possam ser referenciadas, variáveis globais e de funções externas não podem ter atribuições (a menos que seja utilizado o comando `global`, para variáveis globais, ou `nonlocal`, para variáveis de funções externas).

Os parâmetros reais (argumentos) de uma chamada de função são introduzidos na tabela de símbolos local da função no momento da chamada; portanto, argumentos são passados *por valor* (onde o *valor* é sempre uma *referência* para objeto, não o valor do objeto). 1 Quando uma função chama outra função, ou chama a si mesma recursivamente, uma nova tabela de símbolos é criada para tal chamada.

Uma definição de função associa o nome da função com o objeto função na tabela de símbolos atual. O interpretador reconhece o objeto apontado pelo nome como uma função definida pelo usuário. Outros nomes também podem apontar para o mesmo objeto função e também pode ser usados pra acessar a função:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Conhecendo outras linguagens, pode-se questionar que `fib` não é uma função, mas um procedimento, pois ela não devolve um valor. Na verdade, mesmo funções que não usam o comando `return` devolvem um valor, ainda que pouco interessante. Esse valor é chamado `None` (é um nome embutido). O interpretador interativo evita escrever `None` quando ele é o único resultado de uma expressão. Mas se quiser vê-lo pode usar a função `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

É fácil escrever uma função que retorna uma lista de números da série de Fibonacci, ao invés de exibi-los:

```
>>> def fib2(n):  # retorna a série de Fibonacci até n
...     result = []
...     a, b = 0, 1
...     while a < n:
```

```
...         result.append(a)    # Veja abaixo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # chame-o
>>> f100                # escreva o resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este exemplo demonstra novos recursos de Python:

- A instrução `return` finaliza a execução e retorna um valor da função. `return` sem qualquer expressão como argumento retorna `None`. Atingir o final da função também retorna `None`.
- A instrução `result.append(a)` chama um *método* do objeto lista `result`. Um método é uma função que 'pertence' a um objeto, e é chamada `obj.nomemetodo`, onde `obj` é um objeto qualquer (pode ser uma expressão), e `nomemetodo` é o nome de um método que foi definido pelo tipo do objeto. Tipos diferentes definem métodos diferentes. Métodos de diferentes tipos podem ter o mesmo nome sem ambiguidade. (É possível definir seus próprios tipos de objetos e métodos, utilizando *classes*, veja em Classes) O método `append()`, mostrado no exemplo é definido para objetos do tipo lista; adiciona um novo elemento ao final da lista. Neste exemplo, ele equivale a `result = result + [a]`, só que mais eficiente.

4.8. Mais sobre definição de funções

É possível definir funções com um número variável de argumentos. Existem três formas, que podem ser combinadas.

4.8.1. Argumentos com valor padrão

A mais útil das três é especificar um valor padrão para um ou mais argumentos. Isso cria uma função que pode ser invocada com menos argumentos do que os que foram definidos. Por exemplo:

```
def ask_ok(prompt,retries=4,reminder='Por favor, tente
novamente!'):
    for i in range(retries):
        ok = input(prompt)
        retries = retries - 1
        if ok in ('s', 'sim'):
            print('sim/s ok')
        elif ok in ('n', 'não', 'nao'):
```

```
            print('nao/n ok')
        elif retries < 0:
            raise ValueError('resposta de usuário inválida')
        print(reminder)
```

Essa função pode ser chamada de várias formas:

- fornecendo apenas o argumento obrigatório: `ask_ok('pergunta do prompt')`
- fornecendo um dos argumentos opcionais: `ask_ok('pergunta do prompt', 'tentativas', 'resposta para erro')`
- ou fornecendo todos os argumentos: `ask_ok('pergunta do prompt', 'tentativas', 'resposta para erro')`

Este exemplo também introduz a palavra-chave `in`, que verifica se uma sequência contém ou não um determinado valor.

Os valores padrões são avaliados no momento da definição da função, e no escopo em que a função foi *definida*, portanto:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

irá exibir `5`.

Aviso importante: Valores padrões são avaliados apenas uma vez. Isso faz diferença quando o valor é um objeto mutável, como uma lista, dicionário, ou instâncias de classes. Por exemplo, a função a seguir acumula os argumentos passados, nas chamadas subsequentes:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Isso exibirá:

```
[1]
```

```
[1, 2]
[1, 2, 3]
```

Se não quiser que o valor padrão seja compartilhado entre chamadas subsequentes, pode reescrever a função assim:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.8.2. Argumentos nomeados

Funções também podem ser chamadas usando argumentos nomeados da forma `chave=valor`. Por exemplo, a função a seguir:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

aceita um argumento obrigatório (`voltage`) e três argumentos opcionais (`state`, `action`, e `type`). Esta função pode ser chamada de qualquer uma dessas formas:

```
parrot(1000)
# 1 argumento posicional
parrot(voltage=1000)
# 1 argumento de palavra-chave
parrot(voltage=1000000, action='VOOOOOM')
# 2 argumentos de palavra-chave
parrot(action='VOOOOOM', voltage=1000000)
# 2 argumentos de palavra-chave
parrot('a million', 'bereft of life', 'jump')
# 3 argumentos posicionais
parrot('a thousand', state='pushing up the daisies')
# 1 posicional, 1 palavra-chave
```

mas todas as formas a seguir seriam inválidas:

```
parrot()                     # Argumento requerido faltando
parrot(voltage=5.0, 'dead')  # argumento sem palavra-chave após um argumento com palavra-chave
parrot(110, voltage=220)     # valor duplicado para o mesmo argumento
parrot(actor='John Cleese')  # argumento de palavra-chave desconhecida
```

Em uma chamada de função, argumentos nomeados devem vir depois dos argumentos posicionais. Todos os argumentos nomeados passados devem corresponder com argumentos aceitos pela função (ex. `actor` não é um argumento nomeado válido para a função `parrot`), mas sua ordem é irrelevante. Isto também inclui argumentos obrigatórios (ex.: `parrot(voltage=1000)` funciona). Nenhum argumento pode receber mais de um valor. Eis um exemplo que não funciona devido a esta restrição:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Quando um último parâmetro formal no formato `**nome` está presente, ele recebe um dicionário (veja Tipo mapeamento — dict) contendo todos os argumentos nomeados, exceto aqueles que correspondem a um parâmetro formal. Isto pode ser combinado com um parâmetro formal no formato `*nome` (descrito na próxima subseção) que recebe uma tupla contendo os argumentos posicionais, além da lista de parâmetros formais. (`*nome` deve ocorrer antes de `**nome`.) Por exemplo, se definirmos uma função assim:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Pode ser chamada assim:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
```

```
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

e, claro, exibiria:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Observe que a ordem em que os argumentos nomeados são exibidos é garantida para corresponder à ordem em que foram fornecidos na chamada da função.

4.8.3. Parâmetros especiais

Por padrão, argumentos podem ser passadas para uma função Python tanto por posição quanto explicitamente pelo nome. Para uma melhor legibilidade e desempenho, faz sentido restringir a maneira pelo qual argumentos possam ser passados, assim um desenvolvedor precisa apenas olhar para a definição da função para determinar se os itens são passados por posição, por posição e nome, ou por nome.

A definição de uma função pode parecer com:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                   |
        |        Posicional ou palavra-chave   |
        |                                  - Somente palavra-
chave
         -- Somente posicional
```

onde `/` e `*` são opcionais. Se usados, esses símbolos indicam o tipo de parâmetro pelo qual os argumentos podem ser passados para as funções: somente-posicional, posicional-ou-nomeado, e somente-nomeado. Parâmetros nomeados são também conhecidos como parâmetros palavra-chave.

4.8.3.1. Argumentos posicional-ou-nomeados

Se `/` e `*` não estão presentes na definição da função, argumentos podem ser passados para uma função por posição ou por nome.

4.8.3.2. Parâmetros somente-posicionais

Olhando com um pouco mais de detalhes, é possível definir certos parâmetros como *somente-posicional*. Se *somente-posicional*, a ordem do parâmetro importa, e os parâmetros não podem ser passados por nome. Parâmetros somente-posicional são colocados antes de `/` (barra). A `/` é usada para logicamente separar os argumentos somente-posicional dos demais parâmetros. Se não existe uma `/` na definição da função, não existe parâmetros somente-posicionais.

Parâmetros após a `/` podem ser *posicionais-ou-nomeados* ou *somente-nomeado*.

4.8.3.3. Argumentos somente-nomeados

Para definir parâmetros como *somente-nomeado*, indicando que o parâmetro deve ser passado por argumento nomeado, colocamos um `*` na lista de argumentos imediatamente antes do primeiro parâmetro *somente-nomeado*.

4.8.3.4. Exemplos de funções

Considere o seguinte exemplo de definição de função com atenção redobrada para os marcadores `/` e `*`:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

A definição da primeira função, `standard_arg`, a forma mais familiar, não coloca nenhuma restrição para a chamada da função e argumentos podem ser passados por posição ou nome:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

A segunda função `pos_only_arg` está restrita ao uso de parâmetros somente posicionais, uma vez que existe uma `/` na definição da função:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() obtive alguns argumentos somente posicionais passados como argumentos de palavra-chave: 'arg'
```

A terceira função `kwd_only_args` permite somente argumentos nomeados como indicado pelo `*` na definição da função:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() leva 0 argumentos posicionais, mas 1 foi dado

>>> kwd_only_arg(arg=3)
3
```

E a última usa as três convenções de chamada na mesma definição de função:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() leva 2 argumentos posicionais, mas 3 foram dados

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() obtive alguns argumentos somente posicionais passados como argumentos de palavra-chave: 'pos_only'
```

Finalmente, considere essa definição de função que possui uma potencial colisão entre o argumento posicional `name` e `**kwds` que possui `name` como uma chave:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Não é possível essa chamada devolver `True`, uma vez que a chave `'name'` sempre será aplicada para o primeiro parâmetro. Por exemplo:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() tem vários valores para o argumento 'name'
>>>
```

Mas usando `/` (somente argumentos posicionais), isso é possível já que permite `name` como um argumento posicional e `'name'` como uma chave nos argumentos nomeados:

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

Em outras palavras, o nome de parâmetros somente-posicional podem ser usados em `**kwds` sem ambiguidade.

4.8.3.5. Recapitulando

A situação irá determinar quais parâmetros usar na definição da função:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Como guia:

- Use somente-posicional se você não quer que o nome do parâmetro esteja disponível para o usuário. Isso é útil quando nomes de parâmetros não tem um significado real, se você quer forçar a ordem dos argumentos da função quando ela é chamada ou se você precisa ter alguns parâmetros posicionais e alguns nomeados.
- Use somente-nomeado quando os nomes tem significado e a definição da função fica mais clara deixando esses nomes explícitos ou se você quer evitar que usuários confiem na posição dos argumentos que estão sendo passados.

- Para uma API, use somente-posicional para evitar quebras na mudança da API se os nomes dos parâmetros forem alterados no futuro.

ATIVIDADES

1. Faça um programa, com uma função que necessite de três argumentos, e que forneça a soma desses três argumentos.
2. Faça um programa que converta da notação de 24 horas para a notação de 12 horas. Por exemplo, o programa deve converter 14:25 em 2:25 P.M. A entrada é dada em dois inteiros. Deve haver pelo menos duas funções: uma para fazer a conversão e uma para a saída. Registre a informação A.M./P.M. como um valor ‘A’ para A.M. e ‘P’ para P.M. Assim, a função para efetuar as conversões terá um parâmetro formal para registrar se é A.M. ou P.M. Inclua um loop que permita que o usuário repita esse cálculo para novos valores de entrada todas as vezes que desejar.
3. **Reverso do número.** Faça uma função que retorne o reverso de um número inteiro informado. Por exemplo: 127 -> 721.
4. **Data com mês por extenso**. Construa uma função que receba uma data no formato *DD/MM/AAAA* e devolva uma string no formato *D de*

mesPorExtenso de AAAA. Opcionalmente, valide a data e retorne NULL caso a data seja inválida.

5. Faça uma função que informe a quantidade de dígitos de um determinado número inteiro informado.

CAPÍTULO 05

5. Estruturas de dados

Esse capítulo descreve algumas coisas que você já aprendeu em detalhes e adiciona algumas coisas novas também.

5.1. Mais sobre listas

O tipo de dado lista tem ainda mais métodos. Aqui estão apresentados todos os métodos de objetos do tipo lista:

`list.`**`append`**(*x*)

Adiciona um item ao fim da lista. Equivalente a `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)

Prolonga a lista, adicionando no fim todos os elementos do argumento *iterable* passado como parâmetro. Equivalente a `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)

Insere um item em uma dada posição. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim `a.insert(0, x)` insere um elemento na frente da lista e `a.insert(len(a), x)` e equivale a `a.append(x)`.

`list.`**`remove`**(*x*)

Remove o primeiro item encontrado na lista cujo valor é igual a *x*. Se não existir valor igual, uma exceção `ValueError` é levantada.

`list.`**`pop`**([*i*])

Remove um item em uma dada posição na lista e o retorna. Se nenhum índice é especificado, `a.pop()` remove e devolve o último item da lista. (Os colchetes ao redor do *i* na demonstração do método indica que o parâmetro é opcional, e não que é necessário escrever estes colchetes ao chamar o método. Você verá este tipo de notação frequentemente na Biblioteca de Referência Python.)

`list.`**`clear`**()

Remove todos os itens de uma lista. Equivalente a `del a[:]`.

`list.`**`index`**(*x*[, *start*[, *end*]])

Devolve o índice base-zero do primeiro item cujo valor é igual a *x*, levantando `ValueError` se este valor não existe.

Os argumentos opcionais *start* e *end* são interpretados como nas notações de fatiamento e são usados para limitar a busca para uma subsequência específica da lista. O índice retornado é calculado relativo ao começo da sequência inteira e não referente ao argumento *start*.

`list.`**`count`**(*x*)

Devolve o número de vezes em que *x* aparece na lista.

`list.`**`sort`**(**, key=None, reverse=False*)

Ordena os itens na lista (os argumentos podem ser usados para personalizar a ordenação, veja a função `sorted()` para maiores explicações).

`list.`**`reverse`**()

Inverte a ordem dos elementos na lista.

`list.`**`copy`**()

Devolve uma cópia rasa da lista. Equivalente a `a[:]`.

Um exemplo que usa a maior parte dos métodos das listas:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi',
'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)   # Encontre a próxima banana
começando na posição 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi',
'orange', 'pear']
>>> fruits.pop()
'pear'
```

Você pode ter percebido que métodos como `insert`, `remove` ou `sort`, que apenas modificam a lista, não têm valor de retorno impresso – eles retornam o `None` padrão. [1] Isto é um princípio de design para todas as estruturas de dados mutáveis em Python.

Outra coisa que você deve estar atento é que nem todos os dados podem ser ordenados ou comparados. Por exemplo, , `[None, 'hello', 10]` não podem ser ordenados, pois inteiros não podem ser comparados a strings e *None* não pode ser comparado a nenhum outro tipo. Além disso, existem alguns tipos de dados que não possuem uma relação de ordem definida. Por exemplo, `3+4j < 5+7j` não é uma comparação válida.

5.1.1. Usando listas como pilhas

Os métodos de lista tornam muito fácil utilizar listas como pilhas, onde o item adicionado por último é o primeiro a ser recuperado (política "último a entrar, primeiro a sair"). Para adicionar um item ao topo da pilha, use `append()`. Para recuperar um item do topo da pilha use `pop()` sem nenhum índice. Por exemplo:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Usando listas como filas

Você também pode usar uma lista como uma fila, onde o primeiro item adicionado é o primeiro a ser recuperado (política "primeiro a entrar, primeiro a sair"); porém, listas não são eficientes para esta finalidade. Embora *appends* e *pops* no final da lista sejam rápidos, fazer *inserts* ou *pops* no início da lista é lento (porque todos os demais elementos têm que ser deslocados).

Para implementar uma fila, use a classe `collections.deque` que foi projetada para permitir *appends* e *pops* eficientes nas duas extremidades. Por exemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry entra
>>> queue.append("Graham")          # Graham entra
```

```
>>> queue.popleft()                # O primeiro a entrar agora
sai
'Eric'
>>> queue.popleft()                 # O segundo a entrar agora
sai
'John'
>>> queue                           # Fila restante por ordem
de entrada
deque(['Michael', 'Terry', 'Graham'])
```

5.2. A instrução `del`

Existe uma maneira de remover um item de uma lista usando seu índice no lugar do seu valor: a instrução `del`. Ele difere do método `pop()` que devolve um valor. A instrução `del` pode também ser utilizada para remover fatias de uma lista ou limpar a lista inteira (que fizemos antes atribuindo uma lista vazia à fatia `a[:]`). Por exemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` também pode ser usado para remover totalmente uma variável:

```
>>> del a
```

Referenciar a variável `a` depois de sua remoção constitui erro (pelo menos até que seja feita uma nova atribuição para ela). Encontraremos outros usos para a instrução `del` mais tarde.

5.4. Conjuntos

Python também inclui um tipo de dados para conjuntos, chamado `set`. Um conjunto é uma coleção desordenada de elementos, sem elementos repetidos. Usos comuns para conjuntos incluem a verificação eficiente da existência de objetos e a eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

Chaves ou a função `set()` podem ser usados para criar conjuntos. Note: para criar um conjunto vazio você precisa usar `set()`, não `{}`; este último cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

Uma pequena demonstração:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
'banana'}
>>> print(basket)                      # mostrar que as
duplicatas foram removidas
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # teste rápido de adesão
True
>>> 'crabgrass' in basket
False

>>> # Demonstrar operações definidas em letras únicas de duas
palavras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # letras únicas em um
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letras em a mas não
em b
{'r', 'd', 'b'}
>>> a | b                              # letras em a ou b ou
ambos
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letras em a e b
{'a', 'c'}
>>> a ^ b                              # letras em a ou b, mas
não em ambos
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Da mesma forma que compreensão de listas, compreensões de conjunto também são suportadas:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dicionários

Outra estrutura de dados muito útil embutida em Python é o *dicionário*, cujo tipo é `dict` (ver Tipo mapeamento — dict). Dicionários são também chamados de

“memória associativa” ou “vetor associativo” em outras linguagens. Diferente de sequências que são indexadas por inteiros, dicionários são indexados por chaves (*keys*), que podem ser de qualquer tipo imutável (como strings e inteiros). Tuplas também podem ser chaves se contiverem apenas strings, inteiros ou outras tuplas. Se a tupla contiver, direta ou indiretamente, qualquer valor mutável, não poderá ser chave. Listas não podem ser usadas como chaves porque podem ser modificadas *internamente* pela atribuição em índices ou fatias, e por métodos como `append()` e `extend()`.

Um bom modelo mental é imaginar um dicionário como um conjunto não-ordenado de pares *chave:valor*, onde as chaves são únicas em uma dada instância do dicionário. Dicionários são delimitados por chaves: `{}`, e contém uma lista de pares chave:valor separada por vírgulas. Dessa forma também será exibido o conteúdo de um dicionário no console do Python. O dicionário vazio é `{}`.

As principais operações em um dicionário são armazenar e recuperar valores a partir de chaves. Também é possível remover um par *chave:valor* com o comando `del`. Se você armazenar um valor utilizando uma chave já presente, o antigo valor será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro.

Executar `list(d)` em um dicionário devolve a lista de todas as chaves presentes no dicionário, na ordem de inserção (se desejar ordená-las basta usar a função `sorted(d)`). Para verificar a existência de uma chave, use o operador `in`.

A seguir, um exemplo de uso do dicionário:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

O construtor `dict()` produz dicionários diretamente de sequências de pares chave-valor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Além disso, as compreensões de dicionários podem ser usadas para criar dicionários a partir de expressões arbitrárias de chave e valor:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Quando chaves são strings simples, é mais fácil especificar os pares usando argumentos nomeados no construtor:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6. Técnicas de iteração

Ao iterar sobre dicionários, a chave e o valor correspondente podem ser obtidos simultaneamente usando o método `items()`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

Ao iterar sobre sequências, a posição e o valor correspondente podem ser obtidos simultaneamente usando a função `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Para percorrer duas ou mais sequências ao mesmo tempo, as entradas podem ser pareadas com a função `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
```

```
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Para percorrer uma sequência em ordem inversa, chame a função `reversed()` com a sequência na ordem original.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Para percorrer uma sequência de maneira ordenada, use a função `sorted()`, que retorna uma lista ordenada com os itens, mantendo a sequência original inalterada.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

Usar `set()` em uma sequência elimina elementos duplicados. O uso de `sorted()` em combinação com `set()` sobre uma sequência é uma maneira idiomática de fazer um loop sobre elementos exclusivos da sequência na ordem de classificação.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
```

```
apple
banana
orange
pear
```

Às vezes é tentador alterar uma lista enquanto você itera sobre ela; porém, costuma ser mais simples e seguro criar uma nova lista.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5,
float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. Mais sobre condições

As condições de controle usadas em `while` e `if` podem conter quaisquer operadores, não apenas comparações.

Os operadores de comparação `in` e `not in` verificam se um valor ocorre (ou não ocorre) em uma dada sequência. Os operadores `is` e `is not` comparam se dois objetos são na verdade o mesmo objeto. Todos os operadores de comparação possuem a mesma precedência, que é menor do que a prioridade de todos os operadores numéricos.

Comparações podem ser encadeadas: Por exemplo `a < b == c` testa se `a` é menor que `b` e se `b` é igual a `c`.

Comparações podem ser combinadas através de operadores booleanos `and` e `or`, e o resultado de uma comparação (ou de qualquer outra expressão), pode ter seu valor booleano negado através de `not`. Estes possuem menor prioridade que os demais operadores de comparação. Entre eles, `not` é o de maior prioridade e `or` o de menor. Dessa forma, a condição `A and not B or C` é equivalente a `(A and (not B)) or C`. Naturalmente, parênteses podem ser usados para expressar o agrupamento desejado.

Os operadores booleanos `and` e `or` são operadores *curto-circuito*: seus argumentos são avaliados da esquerda para a direita, e a avaliação encerra quando o resultado é determinado. Por exemplo, se `A` e `C` são expressões verdadeiras, mas `B` é falsa, então `A and B and C` não chega a avaliar a

expressão `C`. Em geral, quando usado sobre valores genéricos e não como booleanos, o valor do resultado de um operador curto-circuito é o último valor avaliado na expressão.

É possível atribuir o resultado de uma comparação ou outra expressão booleana para uma variável. Por exemplo:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Observe que no Python, ao contrário de C, a atribuição dentro de expressões deve ser feita explicitamente com o operador morsa `:=`. Isso evita uma classe comum de problemas encontrados nos programas C: digitar `=` em uma expressão quando `==` era o planejado.

ATIVIDADES

1. Faça um programa que leia um número indeterminado de valores, correspondentes a notas, encerrando a entrada de dados quando for informado um valor igual a -1 (que não deve ser armazenado). Após esta entrada de dados, faça:
    a. Mostre a quantidade de valores que foram lidos;
    b. Exiba todos os valores na ordem em que foram informados, um ao lado do outro;
    c. Exiba todos os valores na ordem inversa à que foram informados, um abaixo do outro;
    d. Calcule e mostre a soma dos valores;
    e. Calcule e mostre a média dos valores;
    f. Calcule e mostre a quantidade de valores acima da média calculada;
    g. Calcule e mostre a quantidade de valores abaixo de sete;
    h. Encerre o programa com uma mensagem;

CAPÍTULO 06

6. Módulos

Ao sair e entrar de novo no interpretador Python, as definições anteriores (funções e variáveis) são perdidas. Portanto, se quiser escrever um programa maior, será mais eficiente usar um editor de texto para preparar as entradas para o interpretador, e executá-lo usando o arquivo como entrada. Isso é conhecido como criar um *script*. Se o programa se torna ainda maior, é uma boa prática dividi-lo em arquivos menores, para facilitar a manutenção. Também é preferível usar um arquivo separado para uma função que você escreveria em vários programas diferentes, para não copiar a definição de função em cada um deles.

Para permitir isso, o Python tem uma maneira de colocar as definições em um arquivo e então usá-las em um script ou em uma execução interativa do interpretador. Tal arquivo é chamado de *módulo*; definições de um módulo podem ser *importadas* para outros módulos, ou para o módulo *principal* (a coleção de variáveis a que você tem acesso num script executado como um programa e no modo calculadora).

Um módulo é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo acrescido do sufixo `.py`. Dentro de um módulo, o nome do módulo (como uma string) está disponível como o valor da variável global `__name__`. Por exemplo, use seu editor de texto favorito para criar um arquivo chamado `fibo.py` no diretório atual com o seguinte conteúdo:

```
# Módulo de números de Fibonacci

def fib(n):    # escreve a série de Fibonacci até n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # retorna a série de Fibonacci até n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Agora, entre no interpretador Python e importe esse módulo com o seguinte comando:

```
>>> import fibo
```

Isso não coloca os nomes das funções definidas em `fibo` diretamente na tabela de símbolos atual; isso coloca somente o nome do módulo `fibo`. Usando o nome do módulo você pode acessar as funções:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Se pretender usar uma função muitas vezes, você pode atribui-lá a um nome local:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. Mais sobre módulos

Um módulo pode conter tanto instruções executáveis quanto definições de funções e classes. Essas instruções servem para inicializar o módulo. Eles são executados somente na *primeira* vez que o módulo é encontrado em uma instrução de importação. 1 (Também rodam se o arquivo é executado como um script.)

Cada módulo tem sua própria tabela de símbolos privada, que é usada como tabela de símbolos global para todas as funções definidas no módulo. Assim, o autor de um módulo pode usar variáveis globais no seu módulo sem se preocupar com conflitos acidentais com as variáveis globais do usuário. Por outro lado, se você precisar usar uma variável global de um módulo, poderá fazê-lo com a mesma notação usada para se referir às suas funções, `nomemodulo.nomeitem`.

Módulos podem importar outros módulos. É costume, porém não obrigatório, colocar todos os comandos `import` no início do módulo (ou script , se preferir). As definições do módulo importado são colocadas na tabela de símbolos global do módulo que faz a importação.

Existe uma variante do comando `import` que importa definições de um módulo diretamente para a tabela de símbolos do módulo importador. Por exemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isso não coloca o nome do módulo de onde foram feitas as importações na tabela de símbolos local (assim, no exemplo, `fibo` não está definido).

Existe ainda uma variante que importa todos os nomes definidos em um módulo:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isso importa todos as declarações de nomes, exceto aqueles que iniciam com um sublinhado (`_`). Na maioria dos casos, programadores Python não usam esta facilidade porque ela introduz um conjunto desconhecido de nomes no ambiente, podendo esconder outros nomes previamente definidos.

Note que, em geral, a prática do `import *` de um módulo ou pacote é desaprovada, uma vez que muitas vezes dificulta a leitura do código. Contudo, é aceitável para diminuir a digitação em sessões interativas.

Se o nome do módulo é seguido pela palavra-chave `as`, o nome a seguir é vinculado diretamente ao módulo importado.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isto efetivamente importa o módulo, da mesma maneira que `import fibo` fará, com a única diferença de estar disponível com o nome `fib`.

Também pode ser utilizado com a palavra-chave `from`, com efeitos similares:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Nota

Por motivos de eficiência, cada módulo é importado apenas uma vez por sessão do interpretador. Portanto, se você alterar seus módulos, deverá reiniciar o interpretador – ou, se for apenas um módulo que deseja testar interativamente, use `importlib.reload()`, e.g. `import importlib; importlib.reload(modulename)`.

6.1.1. Executando módulos como scripts

Quando você rodar um módulo Python com

```
python fibo.py <arguments>
```

o código no módulo será executado, da mesma forma que quando é importado, mas com a variável `__name__` com valor `"__main__"`. Isto significa que adicionando este código ao final do seu módulo:

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

você pode tornar o arquivo utilizável tanto como script quanto como um módulo importável, porque o código que analisa a linha de comando só roda se o módulo é executado como arquivo “principal”:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Se o módulo é importado, o código não é executado:

```
>>> import fibo
>>>
```

Isso é frequentemente usado para fornecer uma interface de usuário conveniente para um módulo, ou para realizar testes (rodando o módulo como um script executa um conjunto de testes).

CAPÍTULO 07

7. Entrada e Saída

Existem várias maneiras de apresentar a saída de um programa; os dados podem ser exibidos em forma legível para seres humanos, ou escritos em arquivos para uso posterior. Este capítulo apresentará algumas das possibilidades.

7.1. Refinando a formatação de saída

Até agora vimos duas maneiras de exibir valores: *expressões* e a função `print()`. (Uma outra maneira é utilizar o método `write()` de objetos do tipo arquivo; o arquivo saída padrão pode ser referenciado como `sys.stdout`. Veja a Referência da Biblioteca Python para mais informações sobre isso.)

Muitas vezes se deseja mais controle sobre a formatação da saída do que simplesmente exibir valores separados por espaço. Existem várias maneiras de formatar a saída.

- Para usar strings literais formatadas, comece uma string com `f` ou `F`, antes de abrir as aspas ou aspas triplas. Dentro dessa string, pode-se escrever uma expressão Python entre caracteres `{` e `}`, que podem se referir a variáveis, ou valores literais.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- O método de strings `str.format()` requer mais esforço manual. Ainda será necessário usar `{` e `}` para marcar onde a variável será substituída e pode-se incluir diretivas de formatação detalhadas, mas também precisará incluir a informação a ser formatada.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>>  '{:-9}  YES  votes  {:2.2%}'.format(yes_votes,
percentage)
' 42572654 YES votes  49.67%'
```

- Finalmente, pode-se fazer todo o tratamento da saída usando as operações de fatiamento e concatenação de strings para criar qualquer layout que se possa imaginar. O tipo string possui alguns métodos que realizam operações úteis para preenchimento de strings para uma determinada largura de coluna.

Quando não é necessário sofisticar a saída, mas apenas exibir algumas variáveis com propósito de depuração, pode-se converter qualquer valor para uma string com as funções `repr()` ou `str()`.

A função `str()` serve para retornar representações de valores que sejam legíveis para as pessoas, enquanto `repr()` é para gerar representações que o interpretador Python consegue ler (ou levantará uma exceção `SyntaxError`, se não houver sintaxe equivalente). Para objetos que não têm uma representação adequada para consumo humano, `str()` devolve o mesmo valor que `repr()`. Muitos valores, tal como números ou estruturas, como listas e dicionários, têm a mesma representação usando quaisquer das funções. Strings, em particular, têm duas representações distintas.

Alguns exemplos:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y)
+ '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # O repr() de uma string adiciona aspas e barras invertidas:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # O argumento para repr() pode ser qualquer objeto Python:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
```

O módulo `string` contém uma classe `Template` que oferece ainda outra maneira de substituir valores em strings, usando espaços reservados como `$x` e substituindo-os por valores de um dicionário, mas oferece muito menos controle da formatação.

7.1.1. Strings literais formatadas

Strings literais formatadas (também chamadas f-strings, para abreviar) permite que se inclua o valor de expressões Python dentro de uma string, prefixando-a com `f` ou `F` e escrevendo expressões na forma `{expression}`.

Um especificador opcional de formato pode ser incluído após a expressão. Isso permite maior controle sobre como o valor é formatado. O exemplo a seguir arredonda pi para três casas decimais:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Passando um inteiro após o `':'` fará com que o campo tenha um número mínimo de caracteres de largura. Isso é útil para alinhar colunas.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Outros modificadores podem ser usados para converter o valor antes de ser formatado. `'!a'` aplica a função `ascii()`, `'!s'` aplica a função `str()` e `'!r'` aplica a função `repr()`

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

Para uma referência dessas especificações de formatos, veja o guia de referência para Minilinguagem de especificação de formato.

7.1.4. Formatação de strings à moda antiga

O operador % (módulo) também pode ser usado para formatação de string. Dado `'string' % valores`, as instâncias de `%` em `string` são substituídas por zero ou mais elementos de `valores`. Essa operação é conhecida como interpolação de string. Por exemplo:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Mais informação pode ser encontrada na seção Formatação de String no Formato printf-style.

ATIVIDADES

1. **Nome na vertical em escada.** Faça um programa que solicite o nome do usuário e mostre o nome em formato de escada.

```
F
FU
FUL
FULA
FULAN
FULANO
```

2. **Jogo de Forca.** Desenvolva um jogo da forca. O programa terá uma lista de palavras lidas de um arquivo texto e escolherá uma aleatoriamente. O jogador poderá errar 6 vezes antes de ser enforcado.

```
Digite uma letra: A
-> Você errou pela 1ª vez. Tente de novo!

Digite uma letra: O
A palavra é: _ _ _ _ O

Digite uma letra: E
A palavra é: _ E _ _ O

Digite uma letra: S
-> Você errou pela 2ª vez. Tente de novo!
```

7.2. Leitura e escrita de arquivos

A função `open()` devolve um objeto arquivo, e é frequentemente usada com dois argumentos: `open(nome_do_arquivo, modo)`.

```
>>> f = open('workfile', 'w')
```

O primeiro argumento é uma string contendo o nome do arquivo. O segundo argumento é outra string, contendo alguns caracteres que descrevem o modo como o arquivo será usado. *modo* pode ser `'r'` quando o arquivo será apenas lido, `'w'` para escrever (se o arquivo já existir seu conteúdo prévio será apagado), e `'a'` para abrir o arquivo para adição; qualquer escrita será adicionada ao final do arquivo. A opção `'r+'` abre o arquivo tanto para leitura

como para escrita. O argumento *modo* é opcional, em caso de omissão será assumido `'r'`.

Normalmente, arquivos são abertos em *modo texto*, ou seja, você lê e grava strings, de e para o arquivo, numa codificação específica. Se a codificação não for especificada, o padrão é dependente da plataforma/sistema operacional (consulte `open()`). Incluir `'b'` ao *modo* abre o arquivo em *modo binário*: os dados são lidos e escritos na forma de bytes. Esse modo deve ser usado para todos os arquivos que não contenham texto.

Em modo texto, o padrão durante a leitura é converter terminadores de linha específicos da plataforma (`\n` no Unix, `\r\n` no Windows) para apenas `\n`. Ao escrever no modo de texto, o padrão é converter as ocorrências de `\n` de volta para os finais de linha específicos da plataforma. Essa modificação de bastidores nos dados do arquivo é adequada para arquivos de texto, mas corromperá dados binários, como arquivos `JPEG` ou `EXE`. Tenha muito cuidado para só usar o modo binário, ao ler e gravar esses arquivos.

É uma boa prática usar a palavra-chave `with` ao lidar com arquivos. A vantagem é que o arquivo é fechado corretamente após o término de sua utilização, mesmo que uma exceção seja levantada em algum momento. Usar `with` também é muito mais curto que escrever seu bloco equivalente `try-finally`:

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # Podemos verificar se o arquivo foi fechado automaticamente.
>>> f.closed
True
```

Se você não está usando a palavra reservada `with`, então você deveria chamar `f.close()` para fechar o arquivo e imediatamente liberar qualquer recurso do sistema usado por ele.

Aviso

Chamar `f.write()` sem usar a palavra reservada `with` ou chamar``f.close()`` **pode** resultar nos argumentos de `f.write()` não serem completamente escritos no disco, mesmo se o programa for encerrado com êxito.

Depois que um arquivo é fechado, seja por uma instrução `with` ou chamando `f.close()`, as tentativas de usar o arquivo falharão automaticamente.

```
>>> f.close()
>>> f.read()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1. Métodos de objetos arquivo

Para simplificar, o resto dos exemplos nesta seção assumem que um objeto arquivo chamado `f` já foi criado.

Para ler o conteúdo de um arquivo, chame `f.read(tamanho)`, que lê um punhado de dados devolvendo-os como uma string (em modo texto) ou bytes (em modo binário). *tamanho* é um argumento numérico opcional. Quando *tamanho* é omitido ou negativo, todo o conteúdo do arquivo é lido e devolvido; se o arquivo é duas vezes maior que memória da máquina, o problema é seu. Caso contrário, no máximo *tamanho* caracteres (em modo texto) ou *tamanho* bytes (em modo binário) são lidos e devolvidos. Se o fim do arquivo for atingido, `f.read()` devolve uma string vazia (`''`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

O método `f.readline()` lê uma única linha do arquivo; o caractere de quebra de linha (`\n`) é mantido ao final da string, e só é omitido na última linha do arquivo, se o arquivo não terminar com uma quebra de linha. Isso elimina a ambiguidade do valor retornado; se `f.readline()` retorna uma string vazia, o fim do arquivo foi atingido. Linhas em branco são representadas por um `'\n'` – uma string contendo apenas o caractere terminador de linha.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Uma maneira alternativa de ler linhas do arquivo é iterar diretamente pelo objeto arquivo. É eficiente, rápido e resulta em código mais simples:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Se desejar ler todas as linhas de um arquivo em uma lista, pode-se usar `list(f)` ou `f.readlines()`.

`f.write(string)` escreve o conteúdo de *string* para o arquivo, retornando o número de caracteres escritos.

```
>>> f.write('This is a test\n')
15
```

Outros tipos de objetos precisam ser convertidos – seja para uma string (em modo texto) ou para bytes (em modo binário) – antes de escrevê-los:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # converter a tupla em string
>>> f.write(s)
18
```

`f.tell()` retorna um inteiro dando a posição atual do objeto arquivo, no arquivo representado, como número de bytes desde o início do arquivo, no modo binário, e um número ininteligível, quando no modo de texto.

Para mudar a posição, use `f.seek(offset, de_onde)`. A nova posição é computada pela soma do deslocamento *offset* a um ponto de referência especificado pelo argumento *de-onde*. Se o valor de *de_onde* é 0,a referência é o início do arquivo, 1 refere-se à posição atual, e 2 refere-se ao fim do arquivo. Este argumento pode ser omitido e o valor padrão é 0, usando o início do arquivo como referência.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Ir para o 6º byte no arquivo
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Ir para o 3º byte antes do final
13
>>> f.read(1)
b'd'
```

Em arquivos texto (abertos sem um `b`, em modo string), somente *seeks* relativos ao início do arquivo serão permitidos (exceto se for indicado o final do arquivo, com `seek(0, 2)`) e o único valor válido para *offset* são aqueles retornados por chamada à `f.tell()`, ou zero. Qualquer outro valor para *offset* produz um comportamento indefinido.

Objetos arquivo tem alguns método adicionais, como `isatty()` e `truncate()` que não são usados com frequência; consulte a Biblioteca de Referência para um guia completo de objetos arquivo.

ATIVIDADES

1. A ACME Inc., uma empresa de 500 funcionários, está tendo problemas de espaço em disco no seu servidor de arquivos. Para tentar resolver este problema, o Administrador de Rede precisa saber qual o espaço ocupado pelos usuários, e identificar os usuários com maior espaço ocupado. Através de um programa, baixado da Internet, ele conseguiu gerar o seguinte arquivo, chamado "usuarios.txt":

```
alexandre       456123789
anderson        1245698456
antonio         123456456
carlos          91257581
cesar           987458
rosemary        789456125
```

   Neste arquivo, o nome do usuário possui 15 caracteres. A partir deste arquivo, você deve criar um programa que gere um relatório, chamado "relatório.txt", no seguinte formato:

```
ACME Inc.               Uso do espaço em disco pelos usuários
------------------------------------------------------------------------
--------
Nr.  Usuário        Espaço utilizado     % do uso

1    alexandre       434,99 MB             16,85%
2    anderson       1187,99 MB             46,02%
3    antonio         117,73 MB              4,56%
4    carlos           87,03 MB              3,37%
5    cesar             0,94 MB              0,04%
6    rosemary        752,88 MB             29,16%

Espaço total ocupado: 2581,57 MB
Espaço médio ocupado: 430,26 MB
```

   O arquivo de entrada deve ser lido uma única vez, e os dados armazenados em memória, caso sejam necessários, de forma a agilizar a execução do programa. A conversão do espaço ocupado em disco, de bytes para megabytes deverá ser feita através de uma função separada, que será chamada pelo programa principal. O cálculo do percentual de uso também deverá ser feito através de uma função, que será chamada pelo programa principal.

CAPÍTULO 08

8. Erros e exceções

Até agora mensagens de erro foram apenas mencionadas, mas se você testou os exemplos, talvez tenha esbarrado em algumas. Existem pelo menos dois tipos distintos de erros: *erros de sintaxe* e *exceções*.

8.1. Erros de sintaxe

Erros de sintaxe, também conhecidos como erros de parse, são provavelmente os mais frequentes entre aqueles que ainda estão aprendendo Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

O parser repete a linha inválida e apresenta uma pequena 'seta' apontando para o ponto da linha em que o erro foi detectado. O erro é causado (ou ao menos detectado) pelo símbolo que *precede* a seta: no exemplo, o erro foi detectado na função `print()`, uma vez que um dois-pontos (`':'`) está faltando antes dela. O nome de arquivo e número de linha são exibidos para que você possa rastrear o erro no texto do script.

8.2. Exceções

Mesmo que um comando ou expressão estejam sintaticamente corretos, talvez ocorra um erro na hora de sua execução. Erros detectados durante a execução são chamados *exceções* e não são necessariamente fatais: logo veremos como tratá-las em programas Python. A maioria das exceções não são tratadas pelos programas e acabam resultando em mensagens de erro:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

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

A última linha da mensagem de erro indica o que aconteceu. Exceções surgem com diferentes tipos, e o tipo é exibido como parte da mensagem: os tipos no exemplo são `ZeroDivisionError`, `NameError` e `TypeError`. A string exibida como sendo o tipo da exceção é o nome da exceção embutida que ocorreu. Isso é verdade para todas exceções pré-definidas em Python, mas não é necessariamente verdade para exceções definidas pelo usuário (embora seja uma convenção útil). Os nomes das exceções padrões são identificadores embutidos (não palavras reservadas).

O resto da linha é um detalhamento que depende do tipo da exceção ocorrida e sua causa.

A parte anterior da mensagem de erro apresenta o contexto onde ocorreu a exceção. Essa informação é denominada *stack traceback* (situação da pilha de execução). Em geral, contém uma lista de linhas do código-fonte, sem apresentar, no entanto, linhas lidas da entrada padrão.

Exceções embutidas lista as exceções pré-definidas e seus significados.

8.3. Tratamento de exceções

É possível escrever programas que tratam exceções específicas. Observe o exemplo seguinte, que pede dados ao usuário até que um inteiro válido seja fornecido, ainda permitindo que o programa seja interrompido (utilizando `Control-C` ou seja lá o que for que o sistema operacional suporte); note que uma interrupção gerada pelo usuário será sinalizada pela exceção `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...             print("Oops!  That was no valid number.  Try
again...")
...
```

A instrução `try` funciona da seguinte maneira:

- Primeiramente, a *cláusula try* (o conjunto de instruções entre as palavras reservadas `try` e `except`) é executada.
- Se nenhuma exceção ocorrer, a *cláusula except* é ignorada e a execução da instrução `try` é finalizada.
- Se ocorrer uma exceção durante a execução de uma cláusura `try`, as instruções remanescentes na cláusula são ignoradas. Se o tipo da

exceção ocorrida tiver sido previsto em algum `except`, essa *cláusura except* é executada, e então depois a execução continua após o bloco try/except.

- Se a exceção levantada não corresponder a nenhuma exceção listada na *cláusula de exceção*, então ela é entregue a uma instrução `try` mais externa. Se não existir nenhum tratador previsto para tal exceção, trata-se de uma *exceção não tratada* e a execução do programa termina com uma mensagem de erro.

8.4. Levantando exceções

A instrução `raise` permite ao programador forçar a ocorrência de um determinado tipo de exceção. Por exemplo:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

O argumento de `raise` indica a exceção a ser levantada. Esse argumento deve ser uma instância de exceção ou uma classe de exceção (uma classe que deriva de `Exception`). Se uma classe de exceção for passada, será implicitamente instanciada invocando o seu construtor sem argumentos:

```
raise ValueError  # abreviação de 'raise ValueError()'
```

Caso você precise determinar se uma exceção foi levantada ou não, mas não quer manipular o erro, uma forma simples de instrução `raise` permite que você levante-a novamente:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. Encadeamento de exceções

A instrução `raise` permite um `from` opcional que torna possível o encadear exceções. Por exemplo:

```
# exc deve ser instância de exceção ou None.
raise RuntimeError from exc
```

Isso pode ser útil quando você está transformando exceções. Por exemplo:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

O encadeamento de exceções acontece automaticamente quando uma exceção é levantada dentro de uma seção `except` ou `finally`. O encadeamento de exceções pode ser desativado usando o instrução `from None`:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

Para mais informações sobre os mecanismos de encadeamento, veja Exceções embutidas.

8.6. Definindo ações de limpeza

A instrução `try` possui outra cláusula opcional, cuja finalidade é permitir a implementação de ações de limpeza, que sempre devem ser executadas independentemente da ocorrência de exceções. Como no exemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Se uma cláusula `finally` estiver presente, a cláusula `finally` será executada como a última tarefa antes da conclusão da instrução `try`. A cláusula `finally` executa se a instrução `try` produz uma exceção. Os pontos a seguir discutem casos mais complexos quando ocorre uma exceção:

- Se ocorrer uma exceção durante a execução da cláusula `try`, a exceção poderá ser tratada por uma cláusula `except`. Se a exceção não for tratada por uma cláusula `except`, a exceção será gerada novamente após a execução da cláusula: keyword:*!finally*.
- Uma exceção pode ocorrer durante a execução de uma cláusula `except` ou `else`. Novamente, a exceção é re-levantada depois que `finally` é executada.
- Se a cláusula `finally` executa uma instrução `break`, `continue` ou `return`, as exceções não são levantadas novamente.
- Se a instrução `try` atingir uma instrução `break`, `continue` ou `return`, a cláusula `finally` será executada imediatamente antes da execução da instrução `break`, `continue` ou `return`.
- Se uma cláusula `finally` incluir uma instrução `return`, o valor retornado será aquele da instrução `return` da cláusula `finally`, não o valor da instrução `return` da cláusula `try`.

Por exemplo:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
```

```
...         return False
...
>>> bool_return()
False
```

Um exemplo mais complicado:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como você pode ver, a cláusula `finally` é executada em todos os casos. A exceção `TypeError` levantada pela divisão de duas strings não é tratada pela cláusula `except` e portanto é re-levantada depois que a cláusula `finally` é executada.

Em aplicação do mundo real, a cláusula `finally` é útil para liberar recursos externos (como arquivos ou conexões de rede), independentemente do uso do recurso ter sido bem sucedido ou não.

8.7. Ações de limpeza predefinidas

Alguns objetos definem ações de limpeza padrões para serem executadas quando o objeto não é mais necessário, independentemente da operação que estava usando o objeto ter sido ou não bem sucedida. Veja o exemplo a seguir, que tenta abrir um arquivo e exibir seu conteúdo na tela.

```
for line in open("myfile.txt"):
    print(line, end="")
```

O problema com esse código é que ele deixa o arquivo aberto um período indeterminado depois que o código é executado. Isso não chega a ser problema em scripts simples, mas pode ser um problema para grandes aplicações. A palavra reservada `with` permite que objetos como arquivos sejam utilizados com a certeza de que sempre serão prontamente e corretamente finalizados.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Depois que a instrução é executada, o arquivo *f* é sempre fechado, mesmo se ocorrer um problema durante o processamento das linhas. Outros objetos que, como arquivos, fornecem ações de limpeza predefinidas as indicarão em suas documentações.

CAPÍTULO 09

9. Classes

Classes proporcionam uma forma de organizar dados e funcionalidades juntos. Criar uma nova classe cria um novo “tipo” de objeto, permitindo que novas “instâncias” desse tipo sejam produzidas. Cada instância da classe pode ter atributos anexados a ela, para manter seu estado. Instâncias da classe também podem ter métodos (definidos pela classe) para modificar seu estado.

Em comparação com outras linguagens de programação, o mecanismo de classes de Python introduz a programação orientada a objetos sem acrescentar muitas novidades de sintaxe ou semântica. É uma mistura de mecanismos equivalentes encontrados em C++ e Modula-3. As classes em Python oferecem todas as características tradicionais da programação orientada a objetos: o mecanismo de herança permite múltiplas classes base (herança múltipla), uma classe derivada pode sobrescrever quaisquer métodos de uma classe ancestral, e um método pode invocar outro método homônimo de uma classe ancestral. Objetos podem armazenar uma quantidade arbitrária de dados de qualquer tipo. Assim como acontece com os módulos, as classes fazem parte da natureza dinâmica de Python: são criadas em tempo de execução, e podem ser alteradas após sua criação.

Usando a terminologia de C++, todos os membros de uma classe (incluindo dados) são *públicos* (veja exceção abaixo Variáveis privadas), e todos as funções membro são *virtuais*. Como em Modula-3, não existem atalhos para referenciar membros do objeto de dentro dos seus métodos: o método (função definida em uma classe) é declarado com um primeiro argumento explícito representando o objeto (instância da classe), que é fornecido implicitamente pela chamada ao método. Como em Smalltalk, classes são objetos. Isso fornece uma semântica para importar e renomear. Ao contrário de C++ ou Modula-3, tipos pré-definidos podem ser utilizados como classes base para extensões por herança pelo usuário. Também, como em C++, a maioria dos operadores (aritméticos, indexação, etc) podem ser redefinidos por instâncias de classe.

(Na falta de uma terminologia universalmente aceita para falar sobre classes, ocasionalmente farei uso de termos comuns em Smalltalk ou C++. Eu usaria termos de Modula-3, já que sua semântica de orientação a objetos é mais próxima da de Python, mas creio que poucos leitores já ouviram falar dessa linguagem.)

9.1. Uma palavra sobre nomes e objetos

Objetos têm individualidade, e vários nomes (em diferentes escopos) podem ser vinculados a um mesmo objeto. Isso é chamado de apelidamento em outras linguagens. Geralmente, esta característica não é muito apreciada, e pode ser ignorada com segurança ao lidar com tipos imutáveis (números, strings, tuplas). Entretanto, apelidamento pode ter um efeito surpreendente na semântica do código Python envolvendo objetos mutáveis como listas, dicionários e a maioria

dos outros tipos. Isso pode ser usado em benefício do programa, porque os apelidos funcionam de certa forma como ponteiros. Por exemplo, passar um objeto como argumento é barato, pois só um ponteiro é passado na implementação; e se uma função modifica um objeto passado como argumento, o invocador verá a mudança — isso elimina a necessidade de ter dois mecanismos de passagem de parâmetros como em Pascal.

9.2. Escopos e espaços de nomes do Python

Antes de introduzir classes, é preciso falar das regras de escopo em Python. Definições de classe fazem alguns truques com espaços de nomes. Portanto, primeiro é preciso entender claramente como escopos e espaços de nomes funcionam, para entender o que está acontecendo. Esse conhecimento é muito útil para qualquer programador Python avançado.

Vamos começar com algumas definições.

Um *espaço de nomes* é um mapeamento que associa nomes a objetos. Atualmente, são implementados como dicionários em Python, mas isso não é perceptível (a não ser pelo desempenho), e pode mudar no futuro. Exemplos de espaços de nomes são: o conjunto de nomes pré-definidos (funções como `abs()` e as exceções pré-definidas); nomes globais em um módulo; e nomes locais na invocação de uma função. De certa forma, os atributos de um objeto também formam um espaço de nomes. O mais importante é saber que não existe nenhuma relação entre nomes em espaços de nomes distintos. Por exemplo, dois módulos podem definir uma função de nome `maximize` sem confusão — usuários dos módulos devem prefixar a função com o nome do módulo, para evitar colisão.

A propósito, utilizo a palavra *atributo* para qualquer nome depois de um ponto. Na expressão `z.real`, por exemplo, `real` é um atributo do objeto `z`. Estritamente falando, referências para nomes em módulos são atributos: na expressão `modname.funcname`, `modname` é um objeto módulo e `funcname` é um de seus atributos. Neste caso, existe um mapeamento direto entre os atributos de um módulo e os nomes globais definidos no módulo: eles compartilham o mesmo espaço de nomes! [1]

Atributos podem ser somente leitura ou para leitura e escrita. No segundo caso, é possível atribuir um novo valor ao atributo. Atributos de módulos são passíveis de atribuição: você pode escrever `modname.the_answer = 42`. Atributos que aceitam escrita também podem ser apagados através da instrução `del`. Por exemplo, `del modname.the_answer` removerá o atributo `the_answer` do objeto referenciado por `modname`.

Espaços de nomes são criados em momentos diferentes e possuem diferentes ciclos de vida. O espaço de nomes que contém os nomes embutidos é criado quando o interpretador inicializa e nunca é removido. O espaço de nomes global de um módulo é criado quando a definição do módulo é lida, e normalmente duram até a terminação do interpretador. Os comandos executados pela

invocação do interpretador, pela leitura de um script com programa principal, ou interativamente, são parte do módulo chamado __main__, e portanto possuem seu próprio espaço de nomes. (Os nomes embutidos possuem seu próprio espaço de nomes no módulo chamado builtins.).

O espaço de nomes local de uma função é criado quando a função é invocada, e apagado quando a função retorna ou levanta uma exceção que não é tratada na própria função. (Na verdade, uma forma melhor de descrever o que realmente acontece é que o espaço de nomes local é "esquecido" quando a função termina.) Naturalmente, cada invocação recursiva de uma função tem seu próprio espaço de nomes.

Um *escopo* é uma região textual de um programa Python onde um espaço de nomes é diretamente acessível. Aqui, "diretamente acessível" significa que uma referência sem um prefixo qualificador permite o acesso ao nome.

Ainda que escopos sejam determinados estaticamente, eles são usados dinamicamente. A qualquer momento durante a execução, existem 3 ou 4 escopos aninhados cujos espaços de nomes são diretamente acessíveis:

- o escopo mais interno, que é acessado primeiro, contém os nomes locais
- os escopos das funções que envolvem a função atual, que são acessados a partir do escopo mais próximo, contém nomes não-locais, mas também não-globais
- o penúltimo escopo contém os nomes globais do módulo atual
- e o escopo mais externo (acessado por último) contém os nomes das funções embutidas e demais objetos pré-definidos do interpretador

Se um nome é declarado no escopo global, então todas as referências e atribuições de valores vão diretamente para o escopo intermediário, que contém os nomes globais do módulo. Para alterar variáveis declaradas fora do escopo mais interno, a instrução nonlocal pode ser usada; caso contrário, todas essas variáveis serão apenas para leitura (a tentativa de atribuir valores a essas variáveis simplesmente criará uma *nova* variável local, no escopo interno, não alterando nada na variável de nome idêntico fora dele).

Normalmente, o escopo local referência os nomes locais da função corrente no texto do programa. Fora de funções, o escopo local referência os nomes do escopo global: espaço de nomes do módulo. Definições de classes adicionam um outro espaço de nomes ao escopo local.

É importante perceber que escopos são determinados estaticamente, pelo texto do código-fonte: o escopo global de uma função definida em um módulo é o espaço de nomes deste módulo, sem importar de onde ou por qual apelido a função é invocada. Por outro lado, a busca de nomes é dinâmica, ocorrendo durante a execução. Porém, a evolução da linguagem está caminhando para uma resolução de nomes estática, em "tempo de compilação", portanto não conte com a resolução dinâmica de nomes! (De fato, variáveis locais já são resolvidas estaticamente.)

Uma peculiaridade especial do Python é que – se nenhuma instrução global ou nonlocal estiver em vigor – as atribuições de nomes sempre entram no escopo mais interno. As atribuições não copiam dados — elas apenas vinculam nomes aos objetos. O mesmo vale para exclusões: a instrução `del x` remove a ligação de `x` do espaço de nomes referenciado pelo escopo local. De fato, todas as operações que introduzem novos nomes usam o escopo local: em particular, instruções import e definições de funções ligam o módulo ou o nome da função no escopo local.

A instrução global pode ser usada para indicar que certas variáveis residem no escopo global ao invés do local; a instrução nonlocal indica que variáveis particulares estão em um espoco mais interno e devem ser recuperadas lá.

9.2.1. Exemplo de escopos e espaço de nomes

Este é um exemplo que demonstra como se referir aos diferentes escopos e aos espaços de nomes, e como global e nonlocal pode afetar ligação entre as variáveis:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

A saída do código de exemplo é:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
```

```
In global scope: global spam
```

Observe como uma atribuição *local* (que é o padrão) não altera o vínculo de *scope_test* a *spam*. A instrução nonlocal mudou o vínculo de *scope_test* de *spam* e a atribuição global alterou a ligação para o nível do módulo.

Você também pode ver que não havia nenhuma ligação anterior para *spam* antes da atribuição global.

9.3. Uma primeira olhada nas classes

Classes introduzem novidades sintáticas, três novos tipos de objetos, e também alguma semântica nova.

9.3.1. Sintaxe da definição de classe

A forma mais simples de definir uma classe é:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definições de classe, assim como definições de função (instruções def), precisam ser executadas antes que tenham qualquer efeito. (Você pode colocar uma definição de classe dentro do teste condicional de um if ou dentro de uma função.)

Na prática, as instruções dentro da definição de classe geralmente serão definições de funções, mas outras instruções são permitidas, e às vezes são bem úteis — voltaremos a este tema depois. Definições de funções dentro da classe normalmente têm uma forma peculiar de lista de argumentos, determinada pela convenção de chamada a métodos — isso também será explicado mais tarde.

Quando se inicia a definição de classe, um novo espaço de nomes é criado, e usado como escopo local — assim, todas atribuições a variáveis locais ocorrem nesse espaço de nomes. Em particular, funções definidas aqui são vinculadas a nomes nesse escopo.

Quando uma definição de classe é finalizada normalmente (até o fim), um *objeto classe* é criado. Este objeto encapsula o conteúdo do espaço de nomes criado pela definição da classe; aprenderemos mais sobre objetos classe na próxima seção. O escopo local que estava vigente antes da definição da classe é

reativado, e o objeto classe é vinculado ao identificador da classe nesse escopo (ClassName no exemplo).

9.3.2. Objetos de Class

Objetos classe suportam dois tipos de operações: *referências a atributos* e *instanciação*.

Referências a atributos de classe utilizam a sintaxe padrão utilizada para quaisquer referências a atributos em Python: obj.nome. Nomes de atributos válidos são todos os nomes presentes dentro do espaço de nomes da classe, quando o objeto classe foi criado. Portanto, se a definição de classe tem esta forma:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

então MyClass.i e MyClass.f são referências a atributo válidas, retornando, respectivamente, um inteiro e um objeto função. Atributos de classe podem receber valores, pode-se modificar o valor de MyClass.i numa atribuição. __doc__ também é um atributo válido da classe, retornando a *documentação* associada: "A simple example class".

Para *instanciar* uma classe, usa-se a mesma sintaxe de invocar uma função. Apenas finja que o objeto classe do exemplo é uma função sem parâmetros, que devolve uma nova instância da classe. Por exemplo (assumindo a classe acima):

```
x = MyClass()
```

cria uma nova *instância* da classe e atribui o objeto resultante à variável local x.

A operação de instanciação (“invocar” um objeto classe) cria um objeto vazio. Muitas classes preferem criar novos objetos com um estado inicial predeterminado. Para tanto, a classe pode definir um método especial chamado __init__(), assim:

```
def __init__(self):
    self.data = []
```

Quando uma classe define um método __init__(), o processo de instanciação automaticamente invoca __init__() sobre a instância recém criada. Em nosso exemplo, uma nova instância já inicializada pode ser obtida desta maneira:

```
x = MyClass()
```

Naturalmente, o método __init__() pode ter parâmetros para maior flexibilidade. Neste caso, os argumentos fornecidos na invocação da classe serão passados para o método __init__(). Por exemplo,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Objetos instância

Agora o que podemos fazer com objetos de instância? As únicas operações compreendidas por objetos de instância são os atributos de referência. Existem duas maneiras válidas para nomear atributos: atributos de dados e métodos.

Atributos de dados correspondem a “variáveis de instância” em Smalltalk, e a “membros de dados” em C++. Atributos de dados não precisam ser declarados. Assim como variáveis locais, eles passam a existir na primeira vez em que é feita uma atribuição. Por exemplo, se `x` é uma instância da `MyClass` criada acima, o próximo trecho de código irá exibir o valor `16`, sem deixar nenhum rastro:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

O outro tipo de referências a atributos de instância é o “método”. Um método é uma função que “pertence” a um objeto instância. (Em Python, o termo método não é aplicado exclusivamente a instâncias de classes definidas pelo usuário: outros tipos de objetos também podem ter métodos. Por exemplo, listas possuem os métodos append, insert, remove, sort, entre outros. Porém, na discussão a seguir, usaremos o termo método apenas para se referir a métodos de classes definidas pelo usuário. Seremos explícitos ao falar de outros métodos.)

Nomes de métodos válidos de uma instância dependem de sua classe. Por definição, cada atributo de uma classe que é uma função corresponde a um método das instâncias. Em nosso exemplo, `x.f` é uma referência de método válida já que `MyClass.f` é uma função, enquanto `x.i` não é, já que `MyClass.i` não é

uma função. Entretanto, `x.f` não é o mesmo que `MyClass.f`. A referência `x.f` acessa um objeto método e a `MyClass.f` acessa um objeto função.

9.3.4. Objetos método

Normalmente, um método é chamado imediatamente após ser referenciado:

```
x.f()
```

No exemplo `MyClass` o resultado da expressão acima será a string `'hello world'`. No entanto, não é obrigatório invocar o método imediatamente: como `x.f` é também um objeto ele pode ser atribuído a uma variável e invocado depois. Por exemplo:

```
xf = x.f
while True:
    print(xf())
```

exibirá o texto `hello world` até o mundo acabar.

O que ocorre precisamente quando um método é invocado? Você deve ter notado que `x.f()` foi chamado sem nenhum argumento, porém a definição da função `f()` especificava um argumento. O que aconteceu com esse argumento? Certamente Python levanta uma exceção quando uma função que declara um argumento é invocada sem nenhum argumento — mesmo que o argumento não seja usado no corpo da função…

Na verdade, pode-se supor a resposta: a particularidade sobre os métodos é que o objeto da instância é passado como o primeiro argumento da função. Em nosso exemplo, a chamada `x.f()` é exatamente equivalente a `MyClass.f(x)`. Em geral, chamar um método com uma lista de *n* argumentos é equivalente a chamar a função correspondente com uma lista de argumentos que é criada inserindo o objeto de instância do método antes do primeiro argumento.

Se você ainda não entende como os métodos funcionam, dê uma olhada na implementação para esclarecer as coisas. Quando um atributo de uma instância, não relacionado a dados, é referenciado, a classe da instância é pesquisada. Se o nome é um atributo de classe válido, e é o nome de uma função, um método é criado, empacotando a instância e a função, que estão juntos num objeto abstrato: este é o método. Quando o método é invocado com uma lista de argumentos, uma nova lista de argumentos é criada inserindo a instância na posição 0 da lista. Finalmente, o objeto função — empacotado dentro do objeto método — é invocado com a nova lista de argumentos.

9.3.5. Variáveis de classe e instância

De forma geral, variáveis de instância são variáveis que indicam dados que são únicos a cada instância individual, e variáveis de classe são variáveis de atributos e de métodos que são comuns a todas as instâncias de uma classe:

```
class Dog:

    kind = 'canine'         # variável de classe compartilhada
por todas as instâncias

    def __init__(self, name):
        self.name = name     # variável de instância exclusiva
para cada instância

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # compartilhado por todos os cães
'canine'
>>> e.kind                  # compartilhado por todos os cães
'canine'
>>> d.name                  # único para d
'Fido'
>>> e.name                  # exclusivo para e
'Buddy'
```

Como vimos em Uma palavra sobre nomes e objetos, dados compartilhados podem causar efeitos inesperados quando envolvem objetos (mutáveis), como listas ou dicionários. Por exemplo, a lista *tricks* do código abaixo não deve ser usada como variável de classe, pois assim seria compartilhada por todas as instâncias de *Dog*:

```
class Dog:

    tricks = []          # uso errado de uma variável de classe

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
```

```
>>> d.tricks         # inesperadamente compartilhado por todos
os cães
['roll over', 'play dead']
```

Em vez disso, o modelo correto da classe deve usar uma variável de instância:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # cria uma nova lista vazia para
cada cão

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Observações aleatórias

Se um mesmo nome de atributo ocorre tanto na instância quanto na classe, a busca pelo atributo prioriza a instância:

```
>>> class Warehouse:
        purpose = 'storage'
        region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Atributos de dados podem ser referenciados por métodos da própria instância, bem como por qualquer outro usuário do objeto (também chamados “clientes” do objeto). Em outras palavras, classes não servem para implementar tipos

puramente abstratos de dados. De fato, nada em Python torna possível assegurar o encapsulamento de dados — tudo é baseado em convenção. (Por outro lado, a implementação de Python, escrita em C, pode esconder completamente detalhes de um objeto e controlar o acesso ao objeto, se necessário; isto pode ser utilizado por extensões de Python escritas em C.)

Clientes devem utilizar atributos de dados com cuidado, pois podem bagunçar invariantes assumidas pelos métodos ao esbarrar em seus atributos de dados. Note que clientes podem adicionar atributos de dados a suas próprias instâncias, sem afetar a validade dos métodos, desde que seja evitado o conflito de nomes. Novamente, uma convenção de nomenclatura poupa muita dor de cabeça.

Não existe atalho para referenciar atributos de dados (ou outros métodos!) de dentro de um método. Isso aumenta a legibilidade dos métodos: não há como confundir variáveis locais com variáveis da instância quando lemos rapidamente um método.

Frequentemente, o primeiro argumento de um método é chamado `self`. Isso não passa de uma convenção: o identificador `self` não é uma palavra reservada nem possui qualquer significado especial em Python. Mas note que, ao seguir essa convenção, seu código se torna legível por uma grande comunidade de desenvolvedores Python e é possível que alguma *IDE* dependa dessa convenção para analisar seu código.

Qualquer objeto função que é atributo de uma classe, define um método para as instâncias dessa classe. Não é necessário que a definição da função esteja textualmente embutida na definição da classe. Atribuir um objeto função a uma variável local da classe é válido. Por exemplo:

```
# Função definida fora da classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Agora `f`, `g` e `h` são todos atributos da classe `C` que referenciam funções, e consequentemente são todos métodos de instâncias da classe `C`, onde `h` é exatamente equivalente a `g`. No entanto, essa prática serve apenas para confundir o leitor do programa.

Métodos podem invocar outros métodos usando atributos de método do argumento `self`:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Métodos podem referenciar nomes globais da mesma forma que funções comuns. O escopo global associado a um método é o módulo contendo sua definição na classe (a classe propriamente dita nunca é usada como escopo global!). Ainda que seja raro justificar o uso de dados globais em um método, há diversos usos legítimos do escopo global. Por exemplo, funções e módulos importados no escopo global podem ser usados por métodos, bem como as funções e classes definidas no próprio escopo global. Provavelmente, a classe contendo o método em questão também foi definida neste escopo global. Na próxima seção veremos razões pelas quais um método pode querer referenciar sua própria classe.

Cada valor é um objeto e, portanto, tem uma *classe* (também chamada de *tipo*). Ela é armazenada como `object.__class__`.

9.5. Herança

Obviamente, uma característica da linguagem não seria digna do nome “classe” se não suportasse herança. A sintaxe para uma classe derivada é assim:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

O identificador `BaseClassName` deve estar definido no escopo que contém a definição da classe derivada. No lugar do nome da classe base, também são aceitas outras expressões. Isso é muito útil, por exemplo, quando a classe base é definida em outro módulo:

```
class DerivedClassName(modname.BaseClassName):
```

A execução de uma definição de classe derivada procede da mesma forma que a de uma classe base. Quando o objeto classe é construído, a classe base é

lembrada. Isso é utilizado para resolver referências a atributos. Se um atributo requisitado não for encontrado na classe, ele é procurado na classe base. Essa regra é aplicada recursivamente se a classe base por sua vez for derivada de outra.

Não há nada de especial sobre instanciação de classes derivadas: `DerivedClassName()` cria uma nova instância da classe. Referências a métodos são resolvidas da seguinte forma: o atributo correspondente é procurado através da cadeia de classes base, e referências a métodos são válidas se essa procura produzir um objeto função.

Classes derivadas podem sobrescrever métodos das suas classes base. Uma vez que métodos não possuem privilégios especiais quando invocam outros métodos no mesmo objeto, um método na classe base que invoca um outro método da mesma classe base pode, efetivamente, acabar invocando um método sobreposto por uma classe derivada. (Para programadores C++ isso significa que todos os métodos em Python são realmente `virtuais`.)

Um método sobrescrito em uma classe derivada, de fato, pode querer estender, em vez de simplesmente substituir, o método da classe base, de mesmo nome. Existe uma maneira simples de chamar diretamente o método da classe base: apenas chame `BaseClassName.methodname(self, arguments)`. Isso é geralmente útil para os clientes também. (Note que isto só funciona se a classe base estiver acessível como `BaseClassName` no escopo global).

Python tem duas funções embutidas que trabalham com herança:

- Use `isinstance()` para verificar o tipo de uma instância: `isinstance(obj, int)` será `True` somente se `obj.__class__` é a classe `int` ou alguma classe derivada de `int`.
- Use `issubclass()` para verificar herança entre classes: `issubclass(bool, int)` é `True` porque `bool` é uma subclasse de `int`. Porém, `issubclass(float, int)` é `False` porque `float` não é uma subclasse de `int`.

9.5.1. Herança múltipla

Python também suporta uma forma de herança múltipla. Uma definição de classe com várias classes bases tem esta forma:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Para a maioria dos casos mais simples, pense na pesquisa de atributos herdados de uma classe pai como o primeiro nível de profundidade, da esquerda para a direita, não pesquisando duas vezes na mesma classe em que há uma sobreposição na hierarquia. Assim, se um atributo não é encontrado em DerivedClassName, é procurado em Base1, depois, recursivamente, nas classes base de Base1, e se não for encontrado lá, é pesquisado em Base2 e assim por diante.

De fato, é um pouco mais complexo que isso; a ordem de resolução de métodos muda dinamicamente para suportar chamadas cooperativas para super(). Essa abordagem é conhecida em outras linguagens de herança múltipla como chamar-o-próximo-método, e é mais poderosa que a chamada à função super, encontrada em linguagens de herança única.

A ordenação dinâmica é necessária porque todos os casos de herança múltipla exibem um ou mais relacionamentos de diamante (em que pelo menos uma das classes pai pode ser acessada por meio de vários caminhos da classe mais inferior). Por exemplo, todas as classes herdam de object, portanto, qualquer caso de herança múltipla fornece mais de um caminho para alcançar object. Para evitar que as classes base sejam acessadas mais de uma vez, o algoritmo dinâmico lineariza a ordem de pesquisa, de forma a preservar a ordenação da esquerda para a direita, especificada em cada classe, que chama cada pai apenas uma vez, e que é monotônica (significando que uma classe pode ser subclassificada sem afetar a ordem de precedência de seus pais). Juntas, essas propriedades tornam possível projetar classes confiáveis e extensíveis com herança múltipla. Para mais detalhes, veja https://www.python.org/download/releases/2.3/mro/.

9.6. Variáveis privadas

Variáveis de instância “privadas”, que não podem ser acessadas, exceto em métodos do próprio objeto, não existem em Python. No entanto, existe uma convenção que é seguida pela maioria dos programas em Python: um nome prefixado com um sublinhado (por exemplo: `_spam`) deve ser tratado como uma parte não-pública da API (seja uma função, um método ou um atributo de dados). Tais nomes devem ser considerados um detalhe de implementação e sujeito a alteração sem aviso prévio.

Uma vez que existe um caso de uso válido para a definição de atributos privados em classes (especificamente para evitar conflitos com nomes definidos em subclasses), existe um suporte limitado a identificadores privados em classes, chamado *desfiguração de nomes*. Qualquer identificador no formato `__spam` (pelo menos dois sublinhados no início, e no máximo um sublinhado no final) é textualmente substituído por `_classname__spam`, onde `classname` é o nome da classe atual com sublinhado(s) iniciais omitidos. Essa desfiguração independe da posição sintática do identificador, desde que ele apareça dentro da definição de uma classe.

A desfiguração de nomes é útil para que subclasses possam sobrescrever métodos sem quebrar invocações de métodos dentro de outra classe. Por exemplo:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update    # cópia privada do método update() original

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # fornece nova assinatura para update()
        # mas não quebra __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

O exemplo acima deve funcionar mesmo se `MappingSubclass` introduzisse um identificador `__update` uma vez que é substituído por `_Mapping__update` na classe `Mapping` e `_MappingSubclass__update` na classe `MappingSubclass`, respectivamente.

Note que as regras de desfiguração de nomes foram projetadas para evitar acidentes; ainda é possível acessar ou modificar uma variável que é considerada privada. Isso pode ser útil em certas circunstâncias especiais, como depuração de código.

Código passado para `exec()` ou `eval()` não considera o nome da classe que invocou como sendo a classe corrente; isso é semelhante ao funcionamento da instrução `global`, cujo efeito se aplica somente ao código que é compilado junto. A mesma restrição se aplica às funções `getattr()`, `setattr()` e `delattr()`, e quando acessamos diretamente o `__dict__` da classe.

9.7. Curiosidades e conclusões

Às vezes, é útil ter um tipo semelhante ao “record” de Pascal ou ao “struct” de C, para agrupar alguns itens de dados. Uma definição de classe vazia funciona bem para este fim:

```
class Employee:
    pass

john = Employee()  # Criar um registro de funcionário vazio

# Preencha os campos do cadastro
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Um trecho de código Python que espera um tipo de dado abstrato em particular, pode receber, ao invés disso, uma classe que imita os métodos que aquele tipo suporta. Por exemplo, se você tem uma função que formata dados obtidos de um objeto do tipo "arquivo", pode definir uma classe com métodos read() e readline() que obtém os dados de um "buffer de caracteres" e passar como argumento.

Métodos de instância tem atributos também: m.__self__ é o objeto instância com o método m(), e m.__func__ é o objeto função correspondente ao método.

ATIVIDADES

1. **Classe Bola:** Crie uma classe que modele uma bola:

a. Atributos: Cor, circunferência, material
b. Métodos: trocaCor e mostraCor

2. **Classe Quadrado:** Crie uma classe que modele um quadrado:

a. Atributos: Tamanho do lado
b. Métodos: Mudar valor do Lado, retornar valor do Lado e calcular Área;

3. **Classe Bomba de Combustível:** Faça um programa completo utilizando classes e métodos que:

a. Possua uma classe chamada bombaCombustível, com no mínimo esses atributos:

1. tipoCombustivel.
2. valorLitro

3. quantidadeCombustivel

b. Possua no mínimo esses métodos:

1. abastecerPorValor( ) – método onde é informado o valor a ser abastecido e mostra a quantidade de litros que foi colocada no veículo
2. abastecerPorLitro( ) – método onde é informado a quantidade em litros de combustível e mostra o valor a ser pago pelo cliente.
3. alterarValor( ) – altera o valor do litro do combustível.
4. alterarCombustivel( ) – altera o tipo do combustível.
5. alterarQuantidadeCombustivel( ) – altera a quantidade de combustível restante na bomba.

OBS: Sempre que acontecer um abastecimento é necessário atualizar a quantidade de combustível total na bomba.

4. **Classe Ponto e Retangulo:** Faça um programa completo utilizando funções e classes que:

    a. Possua uma classe chamada Ponto, com os atributos x e y.
    b. Possua uma classe chamada Retangulo, com os atributos largura e altura.
    c. Possua uma função para imprimir os valores da classe Ponto
    d. Possua uma função para encontrar o centro de um Retângulo.
    e. Você deve criar alguns objetos da classe Retangulo.
    f. Cada objeto deve ter um vértice de partida, por exemplo, o vértice inferior esquerdo do retângulo, que deve ser um objeto da classe Ponto.
    g. A função para encontrar o centro do retângulo deve retornar o valor para um objeto do tipo ponto que indique os valores de x e y para o centro do objeto.
    h. O valor do centro do objeto deve ser mostrado na tela
    i. Crie um menu para alterar os valores do retângulo e imprimir o centro deste retângulo.

CAPÍTULO 10

10. Um breve passeio pela biblioteca padrão

10.1. Interface com o sistema operacional

O módulo `os` fornece dúzias de funções para interagir com o sistema operacional:

```
>>> import os
>>> os.getcwd()      # Retorna o diretório de trabalho atual
'C:\\Python310'
>>> os.chdir('/server/accesslogs')   # Alterar o diretório de trabalho atual
>>> os.system('mkdir today')   # Execute o comando mkdir no shell do sistema
0
```

Certifique-se de usar a forma `import os` ao invés de `from os import *`. Isso evitará que `os.open()` oculte a função `open()` que opera de forma muito diferente.

As funções embutidas `dir()` e `help()` são úteis como um sistema de ajuda interativa para lidar com módulos grandes como `os`:

```
>>> import os
>>> dir(os)
<retorna uma lista de todas as funções do módulo>
>>> help(os)
< retorna uma extensa página de manual criada a partir das docstrings do módulo >
```

Para tarefas de gerenciamento cotidiano de arquivos e diretórios, o módulo `shutil` fornece uma interface de alto nível que é mais simples de usar:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.3. Argumentos de linha de comando

Scripts geralmente precisam processar argumentos passados na linha de comando. Esses argumentos são armazenados como uma lista no atributo *argv* do módulo `sys`. Por exemplo, teríamos a seguinte saída executando `python demo.py one two three` na linha de comando:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

O módulo `argparse` fornece um mecanismo mais sofisticado para processar argumentos de linha de comando. O script seguinte extrai e exibe um ou mais nomes de arquivos e um número de linhas opcional:

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
    description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Quando executada a linha de comando `python top.py --lines=5 alpha.txt beta.txt`, o script define `args.lines` para `5` e `args.filenames` para `['alpha.txt', 'beta.txt']`.

10.6. Matemática

O módulo `math` oferece acesso às funções da biblioteca C para matemática de ponto flutuante:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

O módulo `random` fornece ferramentas para gerar seleções aleatórias:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
```

```
>>> random.sample(range(100), 10)   # amostragem sem reposição
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # flutuação aleatória
0.17970987693706186
>>> random.randrange(6)      # inteiro aleatório escolhido do intervalo (6)
4
```

O módulo `statistics` calcula as propriedades estatísticas básicas (a média, a mediana, a variação, etc.) de dados numéricos:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

O projeto SciPy <https://scipy.org> tem muitos outros módulos para cálculos numéricos.

10.8. Data e hora

O módulo `datetime` fornece classes para manipulação de datas e horas nas mais variadas formas. Apesar da disponibilidade de aritmética com data e hora, o foco da implementação é na extração eficiente dos membros para formatação e manipulação. O módulo também oferece objetos que levam os fusos horários em consideração.

```
>>> # as datas são facilmente construídas e formatadas
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # datas suportam aritmética do calendário
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```