Introdução à Programação em Python

Luiz Felippe S. Rodrigues

Prefácio

Este material foi preparado como base para uma disciplina de graduação (de cerca 60 horas) que visa fornecer um primeiro contato com a programação de computadores a estudantes (e/ou profissionais) das áreas de Arquitetura, Comunicação e Design.

A linguagem escolhida é o Python tanto pela relativa simplicidade de sua sintaxe como pelo fato de ser uma linguagem de propósito geral, com grande número de bibliotecas disponíveis, mantidas por uma comunidade internacional ativa e aberta, o que permite aplicação imediata às necessidades de cada aluno. O Python também tem sido adotado como linguagem de *scripting* para diversos programas já utilizados por este público.

Este material não visa expor de maneira exaustiva os recursos do Python mas deve ser suficiente para que o aluno se sinta confortável para escrever pequenos programas e estar apto a ler e modificar código escrito por outros, além de ser capaz de buscar mais através da documentação e/ou dos inúmeros tutoriais que podem ser obtidos online.

Dadas as características do público-alvo e às limitações de tempo, evitou-se alguns dos temas clássicos em cursos de Introdução à Programação tais como algoritmos de ordenação e análise de eficiência de algoritmos. Também, de maneira geral, evitou-se problemas e exemplos de natureza ou motivação matemática.

Cada capítulo é precedido de uma lista de *objetivos de aprendizagem* que informa o aluno quais habilidades espera-se que ele assimile a partir do assunto exposto. Procurou-se um tom informal e – na medida do possível – lúdico. Espera-se que os alunos digitem, executem e façam pequenos experimentos com cada exemplo de código apresentado. A resolução dos exercícios é essencial dado que o conhecimento é construido de maneira incremental a partir deles.

Os capítulos 1 e 2 exploram conceitos fundamentais da programação (como variáveis, controle de fluxo e funções), e expõem o essencial da sintaxe do Python. No capítulo 3, o aluno aprende a utilizar diversos módulos da biblioteca padrão para manipular arquivos. Ao fim do capítulo 3, há uma pequena digressão para uma introdução a expressões regulares, incluída pelo grande potencial para aplicações práticas ligadas a essas (mas no caso de limitações de tempo, esta última seção pode ser suprimida sem maiores prejuízos). O capítulo 4 introduz os conceitos de classe, objeto e herança, enfatizando o uso prático.

O capítulo 5 introduz o ambiente Processing no modo Python, que permite a produção de aplicações gráficas e interativas.

Finalmente, no capítulo 6, há sugestões de como utilizar o Python e a programação em diversos contextos interessantes a este público.

O conteúdo dos capítulos 1 a 3 (e, possivelmente, uma pequena parte do capítulo 4) pode ser utilizado para um curso intensivo de menor duração (foi utilizado com sucesso em um curso de 30 horas).

Além dos exercícios, há 3 propostas de projetos de maior duração. Além destes, sugere-se um projeto final de natureza mais livre, que utilize os demais conteúdos do curso como um todo em algo que seja do interesse de cada aluno.

São Paulo, outubro de 2016
Luiz Felippe Santiago Rodrigues

Sumário

1 De comandos a algoritmos

Ao fim deste capítulo espera-se que o aluno seja capaz de

- Utilizar o ambiente interativo para executar comandos,
- Compreender o uso de variáveis em computação,
- Utilizar os tipos de dados básicos do Python
- Conhecer e utilizar os diversos operadores básicos
- Compreender o conceito e aplicação de laços e condicionais
- Escrever um pequeno script para um problema pontual
- Escrever comentários de código e preocupar-se em manter o código legível

1.1 Prelúdio – As quatr.. muitas operações

Vamos iniciar lançando o aplicativo IDLE. Este é um ambiente de desenvolvimento integrado (conhecidos pela sigla em inglês IDE) gratuito para o Python. IDEs são programas que facilitam o desenvolvimento de softwares através de ferramentas que permitem navegar por eficientemente por códigos grandes, depuração de erros, etc, além de oferecer um editor com realce de sintaxe e uma maneira de executar o código. IDEs não são essenciais para a programação (mais adiante, vamos experimentar programar utilizando apenas um editor de texto qualquer e um terminal[1]).

Ao iniciar o IDLE, você deve se deparar com uma tela semelhante a

```
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

trata-se do Python Shell, também conhecido como "modo interativo" do Python. Neste ambiente, o código é executado pelo interpretador imediatamente, o que é muito útil

[1] Os termos 'terminal', 'console', 'shell' e 'prompt de comando' são frequentemente usados como sinônimos neste contexto. Eles referem-se à ideia de digitar comandos que são mostrados na tela executados quando pressionada a tecla *Enter*. Trata-se da maneira mais simples e direta de se interagir com um computador.

para pequenos testes. Isso também permite que o Shell seja usado como uma calculadora (potencialmente poderosa), isto é

```
>>> 42 + 17
59
```

O exemplo acima utiliza o *operador* '+' que executa uma soma. Experimente fazer contas com os operadores '-', '*', '/' e verifique se o comportamento é o esperado. Experimente também os operadores '**', '//' e '%' e tente descobrir seus significados. Em particular certifique-se que você é capaz de compreender o resultado da seguinte linha de código

```
>>> 42 // 4 * 4 + 42 % 4
```

Como deve ter sido fácil perceber, o operador '**' efetua uma exponenciação. Já o operador '//' efetua uma *divisão inteira* e o operador '%', conhecido como *modulo*, calcula o resto da divisão inteira.

Até o momento trabalhamos com dois tipos de dados: números *inteiros* e números *reais*[2]. É possível verificar com qual tipo estamos lidando através do comando `type`, por exemplo execute e compare as saídas dos comandos

```
>>> type(42)
>>> type(42.0)
```

Embora se trate do mesmo número, a representação interna no computador é diferente para cada caso. O comando `type` nos informa sobre isso, indicando que o primeiro é uma *instância* da classe `int`, i.e. um inteiro, enquanto o segundo é uma instancia da classe `float`, isto é, utiliza algo chamado *ponto flutuante* para representar números reais (números com casas depois da vírgula). Veremos o que exatamente *classe* significa mais adiante, provisoriamente a classe pode ser encarada apenas como uma categoria ou *tipo de dados*.

Há outro tipo importante para nosso uso imediato: as cadeias de caracteres, também conhecidas como *strings*, em inglês. Para especificar uma cadeia de caracteres, basta envolver um trecho de texto por aspas, que podem ser simples, `'`, ou duplas, `"`.

Alguns dos operadores discutidos anteriormente também funcionarão sobre cadeias de caracteres, experimente os dois comandos a seguir

```
>>> 'Olá mundo!' + "Meu nome é fulano."
>>> print('Isso é ' + "muito "*4 + '"facil"!')
```

Descobrimos assim que cadeias de caracteres podem ser somadas e multiplicadas por *inteiros*. O aluno aventureiro pode e deve examinar o que acontece ao trocar o 4 por 4.0 e experimentar com outros operadores.

O comando `print()`, não surpreendentemente, é usado para imprimir coisas na tela e

[2] A rigor, racionais.

será usado a exaustão no futuro (em particular, quando não se está usando o modo interativo). Na verdade, `print()` é o que chamamos de *função*, e o que aparece dentro dos parênteses é o *argumento* da função. Uma função é algo que recebe um argumento, executa um conjunto de instruções com ele e, possivelmente, retorna um valor.[3]

Sendo assim, função `print()` recebe uma variável, imprime seu valor na tela e não retorna nada.[4]

Note a curiosa combinação de aspas simples e duplas no termo `'"facil"!'` no código anterior. Neste caso, as aspas simples estão indicando que se iniciará a especificação de uma string enquanto as aspas duplas fazem *parte* da string. Execute as linhas a seguir para mais um exemplo:

```
>>> print("Olá!")
>>> print("'Olá'!")
>>> print('"Olá"!')
```

Um último tipo importante são os valores *booleanos*: representados simplesmente as palavras `True` e `False`. Booleanos costumam aparecer como resultado de *operações de comparação*, execute as linhas abaixo para um exemplo:

```
>>> 42 > 17
>>> 42 < 17
>>> 3.0 > 3.0
>>> 2.0 >= 2.0
>>> 37 != 37
>>> 37 == 37
```

note que na última linha o sinal = de igual aparece duplicado!

Também é possível comparar strings:

```
>>> "abcd" == 'abcd'
>>> 'abcd' == 'vxyz'
>>> 'José' < 'Maria'
>>> 'Ab' > 'Cd'
```

note que uma string é considerada "maior" quando ela aparece depois em ordem alfabética.

[3] Em algumas linguagens de programação as funções que não retornam valor nenhum são chamadas de *subrotinas* ou *procedimentos*. Funções pertencentes a uma classe ou objeto recebem o nome de *método*. Conceitualmente, há pouca ou nenhuma diferença entre *subrotinas*, *procedimentos*, *métodos* e *funções*, e eles podem, via de regra, ser encarados como equivalentes.

[4] Em Python, quando não há um valor explicito de retorno, a função retorna o objeto `None`. Experimente executar o curioso comando `print(print("O que retorna a função print?"))` e ver o que acontece.

Associados aos valores booleanos há mais três outros operadores que chamamos de *operadores lógicos* `and`, `or` e `not`. Para entendê-los, execute os comandos a seguir

```
>>> True and False
>>> True or False
>>> not True
```

a primeira vista isso pode parecer pouco intuitivo mas veja o resultado das linhas a seguir

```
>>> (17 > 10) and (42 < 9)
>>> (17 > 10) or (42 < 9)
```

o valor na saída do primeiro comando é a resposta à pergunta: "é verdade que $(17 > 10)$ e que $(42 < 9)$?", cuja a resposta é "não", i.e. `False`. O segundo comando é a resposta para: "é verdade que $(17 > 10)$ ou que $(42 < 9)$?", cuja resposta é "sim".

1.2 Caminhando pelo labirinto – comandos e scripts

A esta altura espera-se que, por conta quantidade de jargão (operadores, tipos de dados, cadeias de caracteres, funções...) e da aparente irrelevância do assuntos da seção anterior, parte dos alunos esteja se sentindo perdida em um labirinto (ou simplesmente enfadada). Tirando proveito disto, o objetivo desta atividade é navegar através do labirinto da figura 1.1. Para isto é necessário executar os seguintes comandos:

```
>>> from labirinto import Labirinto
>>> Lab = Labirinto()
```

note que o Python diferencia letras maiúsculas de letras minúsculas, isto é, ele é *"case-sensitive"*.

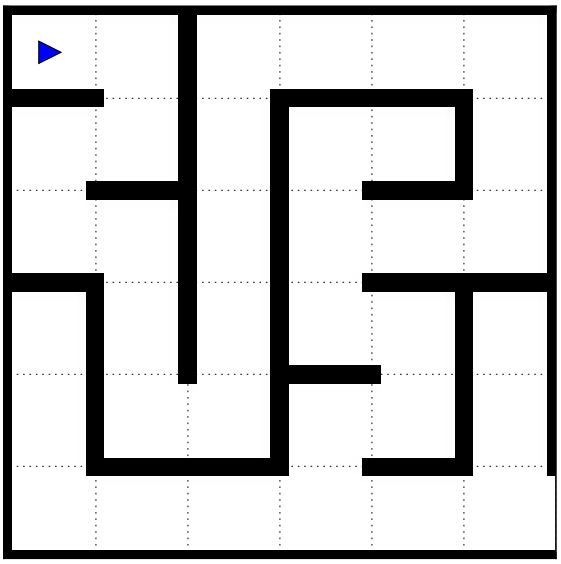


Figura 1.1: O Labirinto e a posição inicial do explorador

A primeira linha de código pode ser ignorada por enquanto. Na segunda linha o labirinto está sendo (de alguma maneira) produzido e armazenado[5] na variável `Lab`. Esta contém não apenas o labirinto em si, mas também a posição de um aventureiro que, iniciando seu trajeto no canto superior esquerdo da figura 1.1 tem a intenção de chegar na saída no canto inferior direito.[6]

[5] Tecnicamente, em Python, `Lab` não armazena o labirinto em si, mas sim uma referência para ele. Mas esta distinção é sutil e será ignorada por enquanto.

[6] Alguém pode estar se perguntando por que razão este pequeno joguinho não mostra o labirinto e o estado explorador na tela a cada movimento. Além de tornar o jogo ligeiramente mais desafiador, isto nos força a manter um modelo mental do que está acontecendo no computador, uma habilidade essencial a um bom programador.

Para a navegar por este labirinto temos à disposição a função `Lab.movimenta()` que pode receber os valores: `"avançar"`, que movimenta o explorador uma casa para frente; `"direita"`, que gira explorador 90º para a direita, e `"esquerda"`, que gira o explorador 90º para a esquerda. Quando um movimento é executado com êxito, a função retorna `True`, caso contrário, quando a instrução enviar o pobre explorador contra a parede, a função retorna `False`, indicando que o movimento foi mal sucedido. A qualquer momento, é possível saber se o aventureiro finalmente conseguiu alcançar saída, utilizando o comando `Lab.livre()` que retorna `True` se o explorador saiu do labirinto e `False` caso não tenha conseguido.

Exercício 1.1. *Utilizando os comandos descritos acima, "resolva" o labirinto, isto é, movimente o explorador com segurança em direção à saída no canto inferior direito.*

A fuga do labirinto no exercício 1.1 foi provavelmente cansativa, envolvendo quantidade significativa de digitação (ou de 'Ctrl+C','Ctrl+V'). Se alguém pedir para fazermos tudo de novo, esta será uma atividade bastante desagradável. É possível, porém, simplificar a vida gravando-se todos os comandos em um arquivo que pode ser executado sempre que necessário. Um arquivo contendo uma sequência de comandos é o que chamamos de *script*, e no contexto do Python usamos este termo como sinônimo de "pequeno programa".

O editor do IDLE pode ser iniciado através do menu 'File'→'New File', vamos utilizá-lo para resolver o labirinto na próxima tarefa. O script pode ser executado a partir do menu 'Run'→'Run module' na janela do editor. Antes da execução, editor irá solicitar que se grave o arquivo. Tradicionalmente, utiliza-se a extensão "`.py`" para scripts em Python.

Exercício 1.2. *Prepare um script que resolve o exercício 1.1 e execute-o.*

O script do exercício 1.2 ainda é claramente precário. De fato, ele poderia ser simplificado significativamente se tivéssemos à nossa disposição uma maneira de fazer o seguinte

```
Seja L uma lista de movimentos que resolvem o labirinto, isto é:
`avançar', `direita', `avançar', etc.

Para cada um das instruções na lista L:
    Execute Lab.movimenta() com a instrução
```

É possível "traduzir" o texto acima diretamente em um script em Python. Veja o código 1.1.

```
1  from labirinto import Labirinto
2
3  Lab = Labirinto() # Inicializa o labirinto
4
5  # Uma lista contendo os movimentos necessários para resolver Lab
6  L = ["avançar","direita","avançar", "direita",
7       "avançar","esquerda", ...]
8
9  # Imprime, uma única vez, a mensagem de boas vindas
10 print("Bem-vindo. Vamos resolver o labirinto.")
11
12 # Resolve o labirinto
13 for instrucao in L:
14     Lab.movimenta(instrucao)
15
16 # Verifica se o explorador saiu e imprime na tela (uma única vez!)
17 # o resultado e uma mensagem de despedida
18 print("O labirinto foi resolvido com sucesso?")
19 print(Lab.livre())
20 print("Tudo terminado agora. Tchau!")
```

Código 1.1: Um primeiro exemplo de uso do laço *for*. Note que as reticências na linha 7 devem ser substituídas pelas demais instruções.

Há várias coisas acontecendo neste trecho de código. Nas linhas 3, 5, 9, 12, 16 e 17 tudo que aparece após o símbolo '#' é ignorado pelo interpretador, trata-se do do que chamamos de *comentários*. Eles servem facilitar o entendimento do código quando ele é lido por outra pessoa ou, talvez mais importante ainda, pelo próprio criador após muito tempo.

Na linha 6 há a definição de uma *lista*. Em Python esta estrutura de dados pode conter quaisquer objetos, podemos definir uma lista simplesmente colocando coisas separadas por vírgulas entre colchetes, por exemplo

```
minha_lista = ["algum texto", 17, 42.42, "se há muitos elementos",
'pode continuar na outra linha', True, False]
```

O elemento número i+1 de uma lista pode ser acessado usando a sintaxe `minha_lista[i]`, onde `i` é o *índice*. Para entender isso melhor, execute o exemplo a seguir

```
print("O primeiro elemento da lista é", minha_lista[0])
print("O terceiro elemento da lista é", minha_lista[2])
print("O penúltimo elemento da lista é", minha_lista[-2])
print("O último elemento da lista é", minha_lista[-1])
```

note que *o índice do primeiro elemento é 0 (zero)*, e que é possível também acessar os

itens to fim para o começo usando índices negativos.

Voltando ao código 1.1, é na linha 9 que está a inovação mais importante: o *laço for* (em inglês, *for-loop*). Laços permitem que um conjunto de linhas de código, chamado de *bloco*, seja executado várias vezes. Cada execução é chamada de uma *iteração*.

Alguns pontos da sintaxe a destacar: na linha 9 palavra `for` marca o início do laço e a cada iteração a variável `instrucao` recebe um dos elementos da lista `L`. O *recuo* na linha 10 é importantíssimo: ele marca o inicio do *bloco* de código que será executado. Neste caso específico, o bloco é uma única linha, que movimenta o explorador sucessivas vezes. Após o fim do laço, as linha 18 imprime mais mensagens na tela e a linha 19 verifica se o labirinto foi resolvido, mostrando o resultado.

```
1  print("Neste código, vou olhar cada um dos itens de minha_lista")
2
3  # Define a variável minha_lista
4  minha_lista = ["algum texto", 17, 42.42, "se há muitos elementos",
5  'pode continuar na outra linha', True, False]
6
7  # Especifica o laço
8  for item in minha_lista:
9      # A ``indentação''/recuo indica que estamos dentro de um bloco
10     print('Estou olhando para o item: "{0}".'.format(item))
11     # Aqui poderíamos fazer mais alguma coisa com o item
12
13     print("Vou agora contar até 3")
14     for i in [0,1,2]:
15         # O recuo indica que estamos dentro de outro bloco
16         print('Contando: {0}'.format(i+1))
17
18     print("\n") # Pula duas linhas
19
20 # Aqui já saímos do bloco
21 print("Terminei de olhar para os itens")
```

Código 1.2: Laços dentro de laços.

É possível fazer uma composição de laços, isto é, a cada iteração, executar um outro laço. Para isso, basta *incrementar o recuo* como pode ser visto nas linhas 15 e 16 do código 1.2.

Nas linhas 10 e 16 do código 1.2, há uma outra novidade: o comando `format`, que permite que variáveis sejam incluídas em cadeias de caracteres. Execute o seguinte exemplo

```
a = 17
b = 42.42
frase = "a = {0}; b= {1}".format(a, b)
print(frase)
```

O {0} indica que o primeiro item do parênteses deve ser incluído naquela posição, {1} o segundo item, e assim por diante.

Exercício 1.3. *Modifique o código 1.1 de maneira que cada instrução seja impressa na tela, precedida do número da instrução, seguindo o modelo: 'Passo 1: avançar'.*

Há diversas situações nas quais desejamos executar um conjunto de instruções *enquanto* uma dada condição for satisfeita. Isto é o que chamamos de *laço while* (ou while-loop). Um exemplo disso pode ser visto a seguir.

```
1 i = 0
2 while(i > 20):
3     i = i + 1
4     print(i)
5 print('Resultado final:',i)
```

Exercício 1.4. *Reescreva o código 1.1 utilizando um laço while. Dica: o que retornava a função livre()?*

1.3 Tateando paredes – algoritmos

Até o momento, trabalhamos olhando para o labirinto conhecido, mostrado na figura 1.1 mas como seria se não possuíssemos figura alguma? Qual conjunto de instruções poderia ser usado para se escapar de qualquer labirinto?

Um conjunto de instruções que podem ser listadas passo-a-passo sem ambiguidade e que quando seguidas resolvem um dado problema é que chamamos de *algoritmo*[7].

Exercício 1.5. *Encontre e escreva um algoritmo em pseudo-código (isto é, em palavras!) que permita a saída de um labirinto semelhante ao da figura 1.1, isto é um labirinto dentro de um retângulo no qual todas as paredes estão conectadas e no qual a saída está sobre borda*[8]*. Atenção:* não *é necessário ser eficiente, mas é necessário sempre sair do labirinto!*

Vamos agora nos armar de ferramentas que permitam *implementar* o algoritmo do exercício 1.5 em Python. Inevitavelmente, algoritmo descrito envolve tomadas de de-

[7] Esta é uma definição informal, mas suficiente para nossos propósitos.
[8] Se a saída for pelo centro ou houver paredes desconectadas a *topologia* do labirinto pode complicar as coisas...

cisão ou *condicionais*: "se a condicão X for verdadeira, execute a ação Y". A sintaxe para condicionais em Python é simplesmente:

```
if cond_X:   # Se cond_X for verdade
    Bloco_Y  # Então executa o Bloco_Y
```

Se houver múltiplas condições ou caso queiramos que um bloco seja executado quando a condição inicial não é satisfeita, podemos utilizar o seguinte:

```
1 if cond_A:   # Se cond_A for verdade
2     Bloco_1  # Então executa o Bloco_1
3 elif cond_B: # Se cond_A for false e cond_B for verdade
4     Bloco_2  # Então executa o Bloco_2
5 else:        # Se cond_A e cond_B forem falsas
6     Bloco_3  # Então executa o Bloco_3
```

Vamos testar nosso entendimento com um exemplo mais complicado:

```
1  # Pede que o usuário digite informações
2  dia_da_semana = input('Digite o dia da semana:')
3  periodo = input('Digite o período (manhã, tarde ou noite):')
4  curso_aos_sabados = True
5
6  # Dependendo da resposta, imprime uma mensagem na tela
7  if dia_da_semana == 'domingo':
8      print("Descanso.")
9  elif (dia_da_semana == 'sexta-feira') and (periodo == 'noite'):
10     print("Festa?")
11 elif dia_da_semana == 'sábado':
12     if (periodo == 'manhã') and curso_aos_sabados:
13         print("Curso.")
14     elif periodo=='noite':
15         print("Festa?")
16     else:
17         print("Descanso.")
18 else:
19     if periodo == 'noite':
20         print("Descanso.")
21     else:
22         print("Trabalho.")
```

Código 1.3: Exemplo de uso de condicionais.

Qual a resposta deste código quando `dia_da_semana` é um dia útil e `periodo`, noite? E durante o dia? Qual a resposta nos fins de semana? Verifique e você compreende bem o código 1.3 da linha 4 em diante e então execute-o.

Nas linhas 2 e 3 do código 1.3 há uma novidade: o comando `input`, que permite que *entrada* de informações no momento da execução. Note que `input` sempre retorna uma cadeia de caracteres.

Um último recurso possivelmente[9] útil é a possibilidade de sortear uma opção de uma lista. Para isto, utilizaremos o módulo `random`, que precisa ser importado no começo do script por usando:

```
import random
```

Um elemento aleatório de uma lista pode ser então sorteado usando

```
L = ['A','B','C']
random.choice(L)
```

Projeto "labirinto" Munidos de todas estas ferramentas, é chegada a hora de implementar o algoritmo do exercício 1.5.

Exercício 1.6. *Implemente o algoritmo do exercício 1.5 em Python e verifique se ele é capaz de resolver o labirinto. O labirinto pode agora ser visualizado em qualquer instante através da função* `Lab.mostra()`.

Exercício 1.7. *Um labirinto aleatório e maior pode ser produzido utilizando*

```
Lab = Labirinto(fixo=False)
```

verifique que seu algoritmo continua funcionando. Verifique também se os comentários que acompanham o código estão escritos com clareza.

[9] Mas não realmente necessário!

2 Organizar e representar

Objetivos de aprendizagem deste capítulo

- Manipular listas e utilizá-las como uma pilha,
- Utilizar *tuples* e saber a diferença entre eles e listas,
- Consultar informações sobre um módulo na documentação,
- Escrever funções, devidamente documentadas,
- Escrever um módulo, devidamente documentado e importá-lo.

2.1 Dando as cartas

Nosso objetivo inicial é achar uma boa maneira de representar as cartas de um baralho no computador. Na verdade, perceberemos que a resposta a esta pergunta dependerá do *uso* que faremos deste baralho, isto é, das características do problema que queremos resolver.

Em um primeiro momento, vamos nos concentrar em jogos simples nos quais as seguintes operações são possivelmente necessárias:

- produzir baralho completo de 52 cartas, em ordem,
- o baralho pode ser embaralhado,
- cartas podem ser retiradas do baralho e acrescentadas a um monte
- o estado do monte e do baralho original podem ser acessados a qualquer momento

É possível atender a primeira especificação de maneira simples, produzindo uma lista de *strings*, cada uma contendo um texto com o valor da carta (por exemplo `"Às de copas"`, `"2 de paus"`, etc). Mas, evidentemente, não vamos digitar as 52 cartas.

Para nos auxiliar nesta tarefa vamos começar aprendendo um pouco mais sobre *listas* em Python. Examine e execute (no modo interativo ou como um script) as seguintes linhas de código

```
A = []
A.append(12)
print(A)
A.append(0)
print(A)
A = A + [1,2,3,4]
print(A)
A.pop()
print(A)
b = A.pop()
print("A = {0}, b={1}".format(A,b))
```

Na primeira linha, criamos uma lista *vazia*. O método `append` acrescenta um elemento ao fim da lista. O método `pop` remove o último elemento e o retorna. O operador +, neste contexto, concatena duas listas.

O uso dos métodos `pop` e `append` permite que tratemos as listas como *pilhas*. Trata-se de uma estrutura de dados que agrupa elementos de maneira que o último elemento adicionado é o primeiro a ser removido (tal como uma pilha no sentido usual da palavra).

Um outro recurso extremamente útil é a habilidade de contar até um dado número. Para isto vamos introduzir o `range`:

```
L = []
for i in range(10):
    print("i = ", i)
    L.append(i)
print("L = ", L)
```

Como pode ser visto no exemplo acima, o construtor de sequências `range` permite que iteremos por uma sequência de inteiros menores que o valor do argumento. Note que isso nos fornece uma maneira de construir listas *empilhando* elementos a cada iteração.

Há 3 maneiras de usar o range: se houver apenas 1 argumento, este é tomado como limite superior da sequência e esta é gerada começando de 0. Se houver 2 argumentos, o primeiro corresponde ao início da sequência e o segundo ao fim da sequência. Finalmente, se houver um terceiro argumento, este é marca o passo. Isso tudo fica mais claro analisando o código 2.1.

Exercício 2.1. *Escreva um código que capaz de produzir um baralho completo. Utilize para isso laços, condicionais, listas e o* `range`*.*

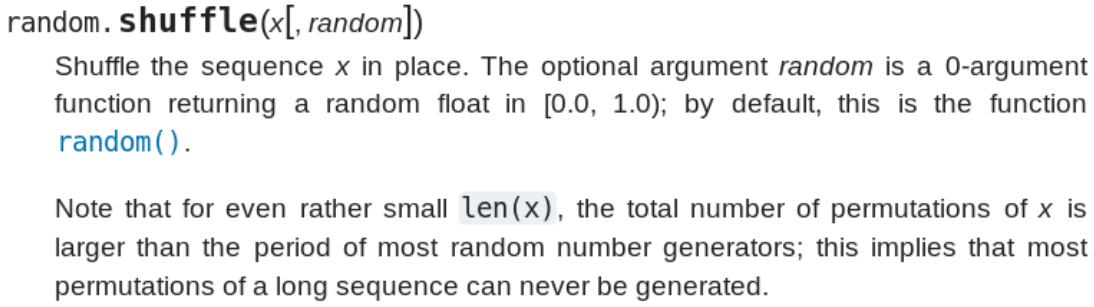
Uma vez possuindo uma lista contendo um baralho gostaríamos ser capazes de manipulá-la de maneira análoga a um baralho físico para sermos capazes de construir alguns jogos simples. Já vimos que é possível acrescentar e remover cartas utilizando os métodos append e pop, respectivamente. A próxima operação importante é ser capaz de embaralhar

```
print("Um argumento (somente o limite superior):")
limite_superior = 10
for i in range(limite_superior):
    print(i, end='\t')

print("\nDois argumentos (inclui o inicio):")
inicio = 3
for i in range(inicio, limite_superior):
    print(i, end='\t')

print("\nTrês argumentos (inclui o passo):")
passo = 2
for i in range(inicio, limite_superior,passo):
    print(i, end='\t')
```

Código 2.1: Exemplificando os diferentes usos da função `range`.

as cartas. Com isto, já seria possível planejar um grande número de jogos e truques.

Vamos usar esta oportunidade para conhecer a documentação da biblioteca padrão do Python, que pode ser encontrada no endereço https://docs.python.org/3/. Dado que o que procuramos envolve aleatoriedade, um bom palpite é pesquisar pelo termo random. De fato, nos deparamos com módulo `random` anteriormente, do qual (possivelmente) utilizamos a função `random.choice`. Ao analisarmos a documentação deste módulo, encontramos, logo após `choice`, a função `random.shuffle` que embaralha um lista.


random.**shuffle**(*x*[, *random*])

Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function `random()`.

Note that for even rather small `len(x)`, the total number of permutations of *x* is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

Figura 2.1: Trecho da documentação que descreve a função `random.shuffle` .

Alguns pontos a destacar desta entrada da documentação: quando aparece escrito "shuffle the sequence x in place", isto está indicando que a lista x será modificada ao se utilizar a função. O uso de colchetes no argumento da função não reflete a sintaxe do Python, mas ao invés disso trata-se de uma convenção: os colchetes indicam argumentos opcionais,

isto é, que podem ser omitidos. Neste caso, o argumento opcional permite que se forneça uma função personalizada para a geração números aleatórios (que de alguma maneira está envolvida, internamente, no embaralhamento). Não precisamos disto, sendo assim, podemos embaralhar nosso baralho simplesmente importando o módulo no começo do script (através de `import random`) e usando `random.shuffle(Bar)` onde `Bar` é uma variável contendo o baralho produzido no exercício 2.1.

2.2 Organizando ações

O script produzido no exercício 2.1 é capaz de gerar um baralho completo e armazená-lo em uma lista. Mas e se estivéssemos programando um jogo que requer vários baralhos? E se a geração do baralho tivesse que ser feita em diferentes partes do código? E se quiséssemos baralhos ligeiramente diferentes (por exemplo, com ou sem curinga)? Além disso, a geração do baralho pode envolver variáveis temporárias (por exemplo uma lista contendo os naipes) que são irrelevantes no restante de nosso programa. Como lidar com tudo isso?

2.2.1 Definindo e utilizando funções

Para nos ajudar com a tarefa de "empacotar" um trecho de código associado a uma dada operação que pode ser executada em vários momentos vamos construir uma *função*. Isso pode ser feito através da seguinte sintaxe:

```
def funcao_boas_vindas():
    print('Bem-vindo à função de boas vindas!')
    print('Saindo da função de boas vindas..')
```

Note que após o aparecimento do bloco acima a execução de `funcao_boas_vindas()` em qualquer parte do código produzirá aquelas mensagens na tela. Esta função não retorna ou recebe qualquer valor. Vamos ver um caso mais interessante:

```
1 def o_maior_numero(A, B):
2     """ Recebe dois números, A e B e retorna o maior dos dois."""
3     if (A>B):
4         return A
5     return B
```

Código 2.2: Uma função que retorna o maior número.

O comando `return` indica qual o *valor de retorno* da função e interrompe sua execução. Por conta disso não foi necessário utilizar um `else`, dado que, no caso `A>B`, o interpretador pula imediatamente da linha 4 para a próxima linha fora da função, sem executar

a linha 5.

Uma vez definida a função, ela pode ser usada em qualquer parte do código, por exemplo se fizermos

```
x = o_maior_numero(17, 42)
```

a variável `x` receberá 42[1]. Note que a função `o_maior_numero` precisa receber exatamente 2 argumentos, qualquer variação resultará em uma mensagem de erro do interpretador.

Há, na verdade, mais de uma maneira de chamar/invocar uma função em Python. No exemplo anterior utilizamos *argumentos posicionais* para chamar a função `o_maior_numero`, isto é, os números `17` e `42` foram associados respectivamente às variáveis A e B na definição da função devido à ordem com que os escrevemos. Há uma outra maneira de especificar os argumentos de uma função no Python que é através do uso de *argumentos palavra-chave* (ou *keyword arguments*). Veja um exemplo a seguir:

```
x = o_maior_numero(B=42, A=17)
```

a variável `x` continua recebendo 42. A ordem dos argumentos da função não é mais importante, mas os nomes das variáveis utilizadas no argumento passam a ser usados para identificá-las.

Voltando ao código 2.2, a linha 2 contém um tipo especial de comentário chamado de *docstring*. Nas docstrings costuma-se[2] escrever a *especificação* da função. A especificação é uma uma descrição sumária do que faz a função incluindo o que ela deve *receber* e *retornar*. É possível ler a docstring de qualquer função através do comando `help`. Experimente, por exemplo, `help(o_maior_numero)`. De fato, a maior parte da documentação do Python que encontramos online na verdade foi extraída de das docstrings escritas no código-fonte da biblioteca padrão!

[1] A função `o_maior_numero()` é apenas um exemplo simples. Se precisássemos realmente comparar dois números, seria mais eficiente e claro utilizar a função `max()` que, assim como `min()`, já vem embutida no Python.

[2] Leia: *deve-se*!

```
1  def nome_completo(nome, sobrenome, invertido=False):
2      """
3      Retorna uma string com um nome completo.
4      Recebe o nome e o sobrenome de uma pessoa, além do argumento
5      opcional 'invertido' que permite que o sobrenome venha escrito
6      antes.
7      """
8      if not invertido:
9          n = nome + " " + sobrenome
10     else:
11         n = sobrenome + ", " + nome
12     return n
```

Código 2.3: Um exemplo de função: nome e sobrenome.

No código 2.3 vemos um outro exemplo exemplo. Note que na declaração da função, na linha 1, há um valor associado ao argumento `invertido`. Esta é a sintaxe para definirmos um *argumento opcional*, sendo que o *valor padrão* do argumento `invertido` é `False`.

Caso invoquemos a função `nome_completo` com apenas dois argumentos, o interpretador assumirá que a variável `invertido` terá o valor padrão. Teste a função `nome_completo` com os seguintes exemplos

```
print(nome_completo("Maria", "da Silva", invertido=False))
print(nome_completo("Maria", "da Silva"))
print(nome_completo(nome="Maria", sobrenome="da Silva", invertido=True))
```

Exercício 2.2. *Adapte o exercício 2.1 escrevendo uma função nomeada* `gera_baralho` *que retorne a lista contendo o baralho. Argumentos opcionais devem controlar as características do baralho produzido (e.g.* `tem_curinga=True` *indicaria se o baralho será gerado com curinga ou não – sendo que, no caso padrão, ele teria um curinga). A função deve ser corretamente documentada, incluindo uma descrição dos argumentos opcionais.*

2.2.2 Escopo: variáveis locais e globais

É possível que nos exercícios anteriores tenham ocorrido algumas surpresas: valores de variáveis definidas dentro de funções simplesmente não eram acessíveis do lado de fora. Vamos executar um exemplo para entender melhor este problema.

```
def f(x):
    y = 1
    x = x + y
    print('x=',x)
    return x
x = 3
y = 2
z = f(x)
print('z=', z)
print('x=', x)
print('y=', y)
```

Código 2.4: Escopo de funções.

A saída deste código é

```
x= 4
z= 4
x= 3
y= 2
```

o que está acontecendo?

Quando definimos uma função, é definido um novo *espaço de nomes* ou escopo , associado exclusivamente aquela função. Isto significa que as variáveis definidas no interior da função, isto é, naquele *escopo*, são independentes daquelas definidas fora da função, elas são definidas *localmente*.

O código 2.4 pode ser entendido melhor se mentalmente trocarmos por outra coisa qualquer coincidência entre nomes contidos em escopos diferentes. Por exemplo, o código 2.5 é absolutamente equivalente ao código 2.4.

```
def f(xf):
    yf = 1
    xf = xf + yf
    print('x=',xf)
    return xf
x = 3
y = 2
z = f(x)
print('z=', z)
print('x=', x)
print('y=', y)
```

Código 2.5: Entendendo escopo: código equivalente ao código 2.4.

Percebemos então que o primeiro `x` impresso na tela pelo código 2.4 é uma *variável local* da função `f` que no código 2.5 recebe o nome de `xf`.

À primeira vista, a existência espaços de nome diferentes para as funções pode parecer confusa mas é na verdade extremamente benéfica: ela permite que funções possam ser "transplantadas" de um arquivo/programa a outro sem perda de funcionalidade.

Há casos, no entanto em que o *precisamos* que a função acesse uma variável de fora de um dado escopo. Para isso utiliza-se o comando `global` como no exemplo no código 2.6 a seguir (experimente também executar também sem a linha 2 para ver o que acontece).

```
1 def acrescenta_nome(n):
2     global nomes
3     nomes = nomes + ' e '+ n
4     return
5 nomes = "João"
6 acrescenta_nomes("Maria")
7 print(nomes)
```

Código 2.6: Variáveis globais.

A circunstância em que realmente precisamos de variáveis globais é bastante rara. Para manter o código mais estável e versátil recomenda-se evitá-las sempre que possível.

2.2.3 Módulos

Até o momento nós trabalhamos com dois módulos externos: o `random` e `labirinto`. *Módulos* são coleções de funções e objetos com funcionalidade ou domínio de aplicação semelhante que são agrupados em um único arquivo.

Vamos ilustrar o uso de módulos com a função `gera_baralho()`, que foi escrita para o exercício 2.2. Grave a função (somente ela) em um arquivo nomeado `baralho.py`. Crie então, no mesmo diretório, um novo arquivo nomeado `usa_baralho.py` que contenha o seguinte código:

```
import baralho
B = baralho.gera_baralho()
print("Produzi o seguinte baralho:", B)
```

O comando `import` executa o arquivo arquivo `baralho.py`, tornando disponível a função `gera_baralho()`. Uma outra maneira de fazer a importação é a que vimos no caso do labirinto, na qual apenas algumas componentes são importadas e incorporadas ao escopo atual, veja o exemplo a seguir.

```
from baralho import gera_baralho
B = gera_baralho()
print('Produzi o seguinte baralho:',B)
```

Não são apenas as funções que podem ser importadas de um dado módulo, mas tudo que está contido nele. Vamos ver um exemplo no código 2.7.

```
1  """ Este é um módulo inútil"""
2  var_A = "Uma variável do módulo"
3  var_B = "Outra variável do módulo"
4
5  def uma_funcao(coisa):
6      """ Recebe uma coisa, imprime a coisa e retorna a mesma coisa"""
7      print(coisa)
8      return coisa
9
10 if __name__ == "__main__":
11     print("Sou o arquivo principal!")
12     print(var_A)
13     print(var_B)
```

Código 2.7: Módulo exemplo, a ser gravado em `inutil.py`.

```
1  import inutil
2  # É possível acessar váriaveis do módulo
3  print('var_A = ', inutil.var_A)
4  print('var_B = ', inutil.var_B)
5  # Assim como funções
6  inutil.uma_funcao('Usando uma_funcao')
```

Código 2.8: Utilizando o módulo `inutil`.

Se gravarmos o código 2.8 na mesma pasta que `inutil.py`, conferimos que podemos acessar tanto funções como variáveis do módulo `inutil`, que através da sintaxe: `nome_do_modulo.variavel` que pode ser vista nas linhas 3 e 4.

As linhas 11, 12 e 13 são executadas *apenas* quando o arquivo `inutil.py` é executado isoladamente (i.e. como um script). Neste último caso, a variável especial `__main__` recebe o valor `"__main__"` e o bloco das linhas 11, 12 e 13 pode ser alcançado.

Isso fornece uma maneira simples de produzir arquivos híbridos: eles podem ser tanto ter conteúdo importado por outros arquivos (funcionando realmente como um módulo), como ser executados como um script (provavelmente interagindo neste caso com o usuário, lendo e imprimindo na tela).

2.3 Jogando: manipulação de listas e tuples

Incrementando nossas ferramentas de baralho, vamos ver um exemplo de como distribuir as cartas entre diferentes jogadores. O script do código 2.9 exemplifica como é possível alcançar isso utilizando o que já vimos.

```
import baralho

def distribui_cartas(numero_de_cartas, Baralho):
    """
    Distribui até numero_de_cartas cartas do baralho Baralho
    entre 2 jogadores. Retorna lista contendo as mãos dos
    2 jogadores (cada uma representada por uma lista).
    Note que Baralho é modificado no processo.
    """
    jogador1 = [] # Lista correspondente ao jogador 1
    jogador2 = [] # Lista correspondente ao jogador 2
    jogadores = [jogador1, jogador2] # Lista de listas

    for nCarta in range(numero_de_cartas):
        for jogador in jogadores:
            if len(Baralho)==0:
                # Evita continuar a distribuição caso o
                # baralho tenha acabado
                break
            # Retira uma carta de um monte e acrescenta
            # à mão de um jogador
            jogador.append(Baralho.pop())
    return jogadores

# Produz o baralho
bar = baralho.gera_baralho()
# Aplica a função, distribuindo 4 cartas do baralho bar
jogadores = distribui_cartas(4, bar)
# Podemos acessar a mão dos jogadores usando
# interagindo diretamente com a lista de listas
print('O Jogador 1 possui as cartas: ', jogadores[0])
print('O Jogador 2 possui as cartas: ', jogadores[1])
# Se quisermos apenas uma carta, basta usar dois índices
print('A primeira carta do jogador 2 é: ', jogadores[1][0])
```

Código 2.9: Script que distribui as cartas de um baralho entre 2 jogadores.

Há alguns recursos novos no código 2.9. O primeiro deles é que estamos utilizando listas que elas mesmas contém listas. O acesso aos elementos é exemplificado nas linhas 31–34.

O segundo recurso novo é a função `len`, que dá comprimento de uma lista (o nome vem do inglês, length). Esta função também pode ser utilizada para medir o comprimento de outros tipos sequenciais, como cadeias de caracteres. Por exemplo:

```
a = [1,2,'a','b']
print(len(a)) # Imprime o comprimento da lista a
b = 'teste'
print(len(b)) # Imprime o comprimento da string b
```

O terceiro recurso novo é um tanto mais sutil. Trata-se da pergunta: como é possível que o laço que se inicia da linha 15 do código 2.9 realmente altere a lista de listas `jogadores`? A resposta não é tão óbvia quanto parece. Vamos iniciar uma pequena digressão sobre este assunto, norteando-nos por alguns exemplos.

```
1  a = 17
2  b = a
3  a += 42 # Isto é uma outra maneira de escrever a = a + 42
4  print("a=",a)
5  print("b=",b)
6
7  La = [1,3,17]
8  Lb = La
9  La.append(42)
10 Lb += [0] # Isto é uma outra maneira de escrever Lb = Lb + [0]
11 La[0] = 13
12 print("La=",La)
13 print("Lb=",Lb)
14
15 Sa = "abcd"
16 Sb = Sa
17 Sa += "efg" # Isto é uma outra maneira de escrever Sa = Sa + "efg"
18 print("Sa=",Sa)
19 print("Sb=",Sb)
```

Código 2.10: Tipos mutáveis e tipos imutáveis.

No caso de tipos numéricos, a atribuição (i.e. o sinal de igual) corresponde a uma cópia do valor. Vejamos isso no código 2.10: na linha 2 copia-se o valor de `a` para `b`, na linha 3 acrescenta-se 42 a `a`, sem alterar o valor de `b`.

O mesmo acontece com as cadeias de caracteres: na linha 13, `Sa` recebe `"abcd"`; na

linha 14, `Sb` recebe uma cópia do valor de `Sa`; na linha 15, acrescenta-se `"efg"` ao valor original de `Sa`, sem interferir no valor de `Sb`.

No caso de listas, porém, o comportamento é diferente. Isto porque, as variáveis `La` e `Lb` na verdade são referências (ou links) que apontam para um mesmo objeto. Sendo assim, no código os nomes `La` e `Lb` são sinônimos: se alterarmos `La` o mesmo será feito a `Lb`.

Esta ideia – de que variáveis associadas a listas não contém a lista em si, mas uma referência para ela – é o que permite que a linha 15 do código 2.9 tenha a funcionalidade esperada: a cada iteração a variável jogador *não* recebe uma cópia das listas contidas em jogadores, mas sim uma referência que aponta para aquele mesmo objeto. Isto significa que, na primeira iteração a linha 21 do código 2.9 está modificando a lista `jogador` que equivale a `jogadores[0]` ou a `jogador1`.

Este mesmo raciocínio é o que permite que listas sejam modificadas no interior de funções: quando passamos uma lista como argumento, a função recebe na verdade uma referência para a lista[3], que pode ser utilizado para alterar o conteúdo da lista. No caso da função `distribui_cartas`, a lista `Baralho` é modificada no interior da função, pelo método `pop()` (como a docstring alerta). Um exemplo mais explícito da alteração de listas no interior de funções pode ser visto a seguir.

```
def testa_mutaveis_imutaveis(lista, numero):
    numero += 17
    lista[0]=2
    return lista, numero

l = [10,20]
x = 42

lNovo, xNovo = testa_mutaveis_imutaveis(l,x)
print(l, lNovo)
print(x, xNovo)
```

Em Python, os tipos de dados que se comportam como as listas são chamados de *mutáveis*, enquanto os que se comportam como os números ou cadeias de caracteres nos exemplos acima são chamados de *imutáveis*.

Há casos, porém, em que realmente queremos fazer uma cópia de uma lista e associá-la a uma nova variável. Para isto podemos utilizar o método `copy`:

[3] Indicando sua posição na memória do computador.

```
La = ["a","b","c","d"]
Lb = La
L_copia = La.copy()
Lb[0] = 42
print("La = ", La)
print("Lb = ", Lb)
print("L_copia = ", L_copia)
```

Exercício 2.3. *Generalize a função do código 2.9 de maneira a permitir que as cartas sejam distribuídas entre qualquer número de jogadores (i.e. a função deve receber um argumento extra dizendo o número de jogadores). Inclua esta função no módulo* `baralho`.

Exercício 2.4. *Acrescente uma função* `corta` *ao módulo* `baralho`*, que corta o baralho (isto é, divide o baralho em dois montes e troca os dois de lugar). A função* `corta` *deve receber uma lista* `Baralho` *como argumento e modificá-la em seu interior. A função deve retornar* `True` *caso o corte seja bem sucedido e retornar* `False`*, caso o baralho consista de uma lista vazia ou contenha apenas uma carta.*

Uma vez que um jogador receba uma mão de cartas (por exemplo, através da função `distribui_cartas`), é interessante que ele seja capaz de ordená-las. Há duas maneiras de ordenar uma lista, que são a função `sorted` e o método associado a listas `sort`, ambos exemplificados no exemplo a seguir.

```
a = ['a','z','g','c','d']
b = sorted(a)
print('a ', a)
print('b ', b)
a.sort()
print('a')
```

Código 2.11: Ordenamento simples de listas.

Como pode ser visto no código 2.11, a função `sorted` retorna uma lista ordenada, enquanto o método `sort` modifica a lista.

Exercício 2.5. *Acrescente a função* `mostra` *ao módulo* `baralho` *que imprime todas as cartas de um dado baralho na tela, mostrando uma carta por linha. A função deve receber uma lista* `B` *como argumento (que contém um baralho ou mão de um jogador). A função deve também possuir um argumento opcional* `em_ordem` *que controla se as cartas devem ser ordenadas antes da impressão ou não. Escreva a função* `mostra` *de maneira que ela* não *modifique* `B`.

Nosso kit de ferramentas para lidar com baralhos, o módulo `baralho`, está ganhando corpo. Há um pequeno problema que alguns já devem ter notado: para a maioria dos jogos, é importante acessar separadamente o naipe e o valor da carta.

Uma maneira de tornar as coisas mais práticas seria modificar a representação que usamos para as cartas. Cada carta precisaria ser representada por um par: (valor, naipe). Seria possível fazer isto utilizando listas, mas vamos introduzir um novo tipo do Python: os *tuples*.

Tuples podem ser definidos e acessados de maneira muito semelhante a listas, utilizando a sintaxe

```
um_tuple = (2, 3, 4, 5) # usando parênteses, e não colchetes!
print("O segundo elemento é", um_tuple[1], " o último é", um_tuple[-1])
```

isto é, trocando-se os colchetes por parênteses para ser definir os tuples e acessa-se os elemento de um tuple da mesma maneira que o fazíamos para listas: colocando o índice do elemento (que começa de 0) entre colchetes ao lado do nome do tuple.

A principal diferença dos tuples para as listas é que tuples são imutáveis, assim como as cadeias de caracteres. Uma vez criados os tuples, não conseguimos modificá-los, consequentemente eles não possuem métodos como `append` ou `pop`.

Esta rigidez é vantajosa para o nosso propósito. Veja o exemplo a seguir

```
1 Baralho_listas = [[2, "paus"], [3, "espadas"]]
2 Baralho_listas[0][0] = 13 # Aqui podemos mudar o valor da carta
3 print("Baralho_listas = ", Baralho_listas)
4
5 Baralho_tuples = [(2, "paus"), (3, "espadas")]
6 Baralho_tuples[0][0] = 13 # Esta linha produzirá um erro!
7 print("Baralho_tuples = ", Baralho_tuples)
```

Código 2.12: Tuples versus listas. N.B. este código produzirá um erro!

Exercício 2.6. *Faça um backup do módulo* `baralho` *em algum lugar seguro (para referência futura). Modifique então a função* `gera_baralho` *de maneira que esta produza uma lista de tuples seguindo o modelo da linha 5 do código 2.12. Desta vez, utilize o números 1, 11, 12, 13 para representar, respectivamente os valores Às, Valete, Dama e Rei.*

Exercício 2.7. *Atualize a função* `mostra` *do módulo* `baralho` *de maneira a torná-la compatível com a nova representação das cartas, que utiliza tuples. A saída da função* `mostra` *deve ser idêntica à da versão antiga (por exemplo, o tuple* `(3, "copas")` *deve ser impresso na tela como:* 3 de copas*).*

Demos um passo importante nos exercícios 2.6 e 2.7: tornamos independente a maneira como representamos as cartas internamente e a maneira como as mostramos, sendo que delegamos à função `mostra` a tarefa de fazer esta tradução.

Durante a resolução do exercício 2.7 deve ter aparecido a questão de como se ordenar

as cartas, dado que `em_ordem` era um argumento opcional na função original construída no exercício 2.5.

Quando aplicamos a função `sorted` sobre uma lista de tuples a ordenação é feita utilizando-se o primeiro elemento do tuple. É possível solicitar que o ordenamento[4] seja feito de maneira diferente utilizando a função auxiliar `itemgetter` do módulo `operator`. Isto está exemplificado no código 2.13, onde pode-se ver como utilizar o argumento opcional `key` da função `sorted` para fazer a ordenação utilizando diferentes itens de nossos tuples. Na linha 13, vemos que há mais um argumento opcional que permite inverter a ordem da lista ordenada que é retornada.

Exercício 2.8. *Acrescente à função `mostra` do módulo `baralho` mais um argumento opcional: `por_naipe`. Caso a função seja chamada com `em_ordem=True` e `por_naipe=True`, as cartas devem ser mostradas na tela ordenadas primeiro por naipe depois por valor. Caso a função seja chamada com `em_ordem=True` e `por_naipe=False`, as cartas devem ser apresentadas na tela ordenadas primeiro por valor depois por naipe.*

```
1  from operator import itemgetter
2  dados = [('Cecília','Meireles', 'Poesia Completa'),
3           ('Iain','Banks', 'The Player of Games'),
4           ('Franz','Kafka', 'Der Process'),
5           ('Fernando','Pessoa', 'Fausto, Tragédia Subjetiva')]
6  print("Ordenando por sobrenome:")
7  for item in sorted(dados, key=itemgetter(1)):
8      print(item)
9  print("\nOrdenando por obra e então nome e então sobrenome:")
10 for item in sorted(dados, key=itemgetter(2,0,1)):
11     print(item)
12 print("\nOrdenando por nome e sobrenome, de trás para frente:")
13 for item in sorted(dados, key=itemgetter(0,1), reverse=True):
14     print(item)
```

Código 2.13: Exemplos de ordenação.

Projeto "oito maluco" Podemos agora nos dedicar a um projeto mais ambicioso: escrever um programa que permita jogar "oito maluco" na tela. Neste jogo, cada jogador recebe nove cartas de um baralho sem curinga. O objetivo do jogo é ficar sem cartas na mão. A cada turno o jogador pode descartar uma carta de sua mão se o valor *ou* o naipe for igual ao da carta da mesa (por "mesa" me refiro à última carta descartada, no primeiro turno, vira-se uma carta do monte). Caso nenhuma carta da mão seja compatível com esta situação, deve-se comprar mais uma carta do monte (ou pular a vez, no caso

[4] Para uma discussão mais completa sobre ordenação em Python, recomenda-se a leitura de https://docs.python.org/3/howto/sorting.html

do monte ter acabado). Algumas cartas possuem valores especiais quando colocadas na mesa: o 8 funciona como um curinga, e pode sempre ser descartado; o Rei faz o jogador seguinte pegar uma carta do monte e descartar outra na mesa; a Rainha de copas faz o jogador anterior pegar 2 cartas; o Valete faz o jogador seguinte pegar duas cartas do monte e perder a vez, e o Às pula a vez do jogador seguinte.

Exercício 2.9. *Escreva um programa que permita n pessoas jogarem "oito maluco". O programa deve solicitar o número de jogadores (utilizando a função* `input`*) e seus nomes. O programa deve então preparar um baralho (por exemplo, armazenando em uma variável* `monte`*), embaralhá-lo, distribuir as cartas entre os jogadores e virar uma carta do monte na mesa. Enquanto nenhum jogador ficar sem cartas, o programa deve seguir mostrando o nome do jogador da vez, a carta da mesa e as cartas na mão do jogador. O jogador deve conseguir, então, interagir com o programa escolhendo qual carta descartar ou comprar uma carta do monte. O programa deve impedir qualquer tipo de jogada ilegal.*

Exercício 2.10. *Acrescente a possibilidade do jogo envolver jogadores robôs – isto é, o programa deve perguntar quantos jogadores e depois quantos humanos. Os jogadores robôs podem se comportar de maneira aleatória, desde que seguindo as regras do jogo.*

3 Automatizando tarefas

Objetivos de aprendizagem deste capítulo

- Leitura e gravação de arquivos,
- Interação com o sistema de arquivos,
- Busca por padrões utilizando expressões regulares,
- Utilização de dicionários,
- Escrever funções recursivas.

Uma das grandes vantagens de se saber programar é poder automatizar tarefas que seriam excessivamente maçantes no nosso dia-a-dia. Neste capítulo exemplificaremos como é possível utilizar pequenos scripts para extrair dados dispersos em um grande número de arquivos.

3.1 Lendo e gravando arquivos: em busca do texto perdido

Vamos nos motivar com o seguinte problema: um dado texto está escondido, com suas linhas dispersas entre cerca de 500 arquivos. Em cada um destes arquivos há apenas 1 linha válida, que contém a sequência de caracteres `xyi`, enquanto o restante do conteúdo destes arquivos é basicamente ruído. Vamos escrever um programa que encontra o texto oculto e o grava em um arquivo.

O primeiro passo é aprender a abrir e ler um arquivo de texto. Isto pode ser feito através da função `open`,

```
1 file_obj_texto = open('arquivo1.txt')
2 texto = file_obj_texto.read()
3 file_obj_texto.close()
4 print(text)
```

neste exemplo, a variável `arquivo_texto` recebe um *objeto arquivo* que trata da leitura do arquivo. Na linha 2 o arquivo é lido por completo e texto é associado à variável `text`. Na linha 3 o arquivo é fechado e o texto é impresso na tela na última linha.

Há uma outra maneira de interagir com um arquivo na qual ele é lido linha por linha, veja o seguinte exemplo, cujo resultado é equivalente ao anterior:

```
file_obj_texto = open('arquivo1.txt')
for linha in file_obj_texto:
    print(linha)
file_obj_texto.close()
```

quando iteramos sobre um objeto arquivo, a cada iteração uma linha é lida.

Um outro recurso do qual precisaremos é, evidentemente, gravar arquivos. A função `open` pode ser acrescida do argumento opcional indicando o *modo* da abertura do arquivo.

```
windows = True
if windows:
    # No windows, uma nova linha é marcada por dois caracteres:
    # \n (muda linha) e \r (volta para o começo)
    nova_linha = '\r\n'
else:
    # Em linux/Mac/Unix/etc usa-se apenas \n
    nova_linha = '\n'

f = open("teste.txt","w")
f.write("Teste" + nova_linha)
f.write("=====" + nova_linha)
f.write("Escrevo ")
f.write("assim em um arquivo ")
f.write("novo." + nova_linha)
f.close()
```

Quando usamos o modo de abertura `"w"` do exemplo acima, o arquivo é criado, caso não exista, ou apagado e recriado, caso exista. Há outros modos de abertura que podem ser vistos na tabela 3.1 abaixo.

modo	significado
`'r'`	abre para leitura (padrão)
`'w'`	abre para escrita, sobrescreve arquivo anterior (se houver)
`'x'`	abre para escrita, produz erro se já existir um arquivo
`'a'`	abre para escrita, escrevendo no fim de um arquivo existente
`'r+w'`	abre para leitura ou gravação

Tabela 3.1: Modos de abertura de arquivos

Há uma última ferramenta que será necessária para lidarmos com nosso problema que é a capacidade de encontrar uma cadeia de caracteres dentro de outra. Strings são tipos sequenciais, então é possível utilizar o comando `in` (que aliás, também funcionam para listas e tuples):

```
texto = "abracadabra, funcione!"

for palavra_magica in ("abracadabra", "alakazam"):
    if palavra_magica in texto:
        print("A palavra mágica "+palavra_magica+" está no texto.")
    else:
        print("A palavra mágica "+palavra_magica+" não está no texto.")
```

Exercício 3.1. *Cada um dos arquivos contidos na pasta 'secreto' contém uma única linha de texto válida, marcada pela sequência de caracteres 'xyi'. Escreva um programa capaz de reconstruir texto completo e gravá-lo em um arquivo nomeado 'idleness.txt'.*

O texto oculto já está bem perto de ser lido. Seria interessante, porém, remover a sequência 'xyi' toda vez que ela aparece. Para conseguir isto, basta lembrar mais uma vez que strings são tipos sequenciais. Portanto, assim como listas, é possível iterar sobre os caracteres (elementos) de uma string utilizando um laço `for`; é possível acessar partes de uma string utilizando índices e fatias, e é possível obter o comprimento de uma string através da função `len`. Veja o seguinte exemplo:

```
string = 'Demonstrando operações com strings.'
print(string[:12]+':')
for caracter in string[27:]:
    print(caracter, end='.')
print()
```

Exercício 3.2. *Escreva uma função que recebe duas strings, `pat` e `text`, e retorna o conteúdo de `text` subtraído do conteúdo de `pat` (i.e. uma função que remove `pat` de `text`), sendo que `pat` contém 3 caracteres.*

Exercício 3.3. *Utilize a função do exercício anterior para modificar a resposta do exercício 3.1 de maneira que o texto secreto seja mostrado sem os marcadores 'xyi'.*

O tipo string contém uma série de métodos úteis para a manipulação de pequenos textos. Uma lista completa deles pode ser obtida de https://docs.python.org/3/library/stdtypes.html#string-methods . Uma operação comum é, a partir de uma string contendo diversas linhas, produzir uma lista contendo uma string com cada linha. Isto pode ser feito através do método `splitlines`

```
teste = 'linha1\nlinha2\nlinha3'
linhas = teste.splitlines()
print('O texto todo:')
print(teste)
print('A segunda linha:')
print(linhas[1])
```

No exemplo a seguir há outros métodos bastante usados:

```
texto = 'podemos separar palavras maiúsculas MINÚSCULAS'
lista = texto.split(' ')
print(lista[0:3])
print(lista[3].upper())
print(lista[4].lower())
texto = 'podemos não encontrar coisas e substituí-las'
texto = texto.replace('não','')
texto = texto.replace('coisas','palavras')
print(texto)
```

Exercício 3.4. *Escreva uma função que recebe 3 listas de palavras:* sujeito, verbo *e* predicado, *e que retorna uma lista de frases com todas as combinações possíveis destas palavras. Cada deve terminar com um ponto final e iniciar com letra maiúscula.*

Exercício 3.5. *Escreva um programa que solicite ao usuário que este digite as 3 listas separadas por vírgulas e utilize a função anterior para imprimir todas as frases possíveis na tela.*

Vamos agora introduzir mais uma estrutura de dados básica do Python: os *dicionários*. Ao contrário das listas ou tuples, em que os índices tinham que ser necessariamente números, dicionários permitem que armazenemos uma coleção de objetos, indexados por (quase) qualquer coisa[1]. Vejamos um exemplo no código 3.1.

```
1 meu_dic = { "chave 1": "Valor associado à chave 1", 17: 'dezessete',
2             'quarenta_e_dois' : 42.0, 'lista': [0,1,2,3] }
3 print(meu_dic['chave 1'])
4 print(meu_dic['quarenta_e_dois']+10)
5 meu_dic[7] = 'aqui defini a nova chave: 7'
6 del meu_dic['chave 1'] # Aqui removi a chave: "chave 1"
7 print('Vou imprimir agora todas as chaves e valores:')
8 for key in meu_dic:
9     print("chave", key,"\tvalor",meu_dic[key])
```

Código 3.1: Exemplo de dicionário.

Na linha 1 há a definição do dicionário `meu_dic`, que é feita colocando entre chaves uma lista separada por vírgulas de todos os pares `chave:valor` . A sintaxe para acessar os elementos do dicionário é a mesma utilizada para acessar uma lista: coloca-se a chave/índice entre colchetes ao lado do nome do dicionário.

[1] Dicionários podem usar como chaves qualquer tipo imutável (como strings, números e tuples). Os dicionários em si são tipos mutáveis, como as listas.

Novos elementos podem ser incluídos fazendo uma atribuição para uma chave nova (como pode ser visto na linha 5). Chaves e valores podem ser removidos utilizando o comando `del` indicado na linha 5.

Finalmente, é possível também iterar sobre um dicionário. Neste caso, que pode ser visto na linha 8, a variável no laço `for` receberá o conteúdo de cada uma das chaves do dicionário.

É importante notar que, ao contrário das listas, a *ordem* não é memorizada ou preservada em dicionários. Isto significa que não há garantias de que a saída associada à linha 9 do código 3.1 seja sempre a mesma: a ordem pode mudar.

É possível produzir listas contendo todos as chaves ou todos os valores de um dicionário através dos métodos `key()` e `items()`, respectivamente. Isto nos permite lidar com o problema da ordem quando esta for relevante. Veja o exemplo no código 3.2.

```
1  import locale # Módulo usado para configuração de língua
2  # Escolhe o português (será usado para a ordenação).
3  locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')
4
5  # Define o dicionário
6  notas_dic = {"João": [6.5,7.0,3.0],
7               "Maria": [9.5, 8.75, 10.0],
8               "Ana": [6.5, 10.0, 7.25],
9               "José": [8.5, 9.0, 8.0]}
10
11 # A partir das chaves, gera uma lista de alunos
12 alunos = list(notas_dic.keys())
13 # Ordena a lista de alunos alfabéticamente
14 # (o argumento é necessário para se lidar com acentos)
15 alunos.sort(key=locale.strxfrm)
16
17 print('Médias finais:')
18 # Itera sobre a lista _ordenada_ de alunos
19 for aluno in alunos:
20     # Calcula a média
21     media = sum(notas_dic[aluno])/len(notas_dic[aluno])
22     print('  {0}:  {1:8.2f}'.format(aluno, media))
```

Código 3.2: Exemplo do uso dos métodos `keys()` e `items()`.

Para testar se uma chave está presente ou não em um dicionário utiliza-se os comandos `in` e `not in`. Isto é particularmente importante quando se quer modificar um elemento de um dicionário: não se pode modificar algo que não existe. No código 3.3 é possível ver um exemplo disto. O teste da linha 14 é necessário para inicializar (na linha 15) a

lista associada àquela chave.

Note que a maneira como o código 3.3 foi escrito permite considerável flexibilidade em relação ao número de notas dos alunos: diferentes alunos podem possuir diferentes números de notas e podem entregar ou não o trabalho extra (e, no caso peculiar de Pedro, podem até mesmo entregar apenas o trabalho extra).

Exercício 3.6. *Escreva uma função que recebe uma cadeia de caracteres e retorna um dicionário contendo palavras como chaves e o número de ocorrências de cada palavra como valor. Aplique, então, esta função ao texto secreto da atividade anterior, e imprima na tela as 5 palavras mais usadas no texto.*

```
import locale
locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')

# Define o dicionário
notas_dic = { "João": [6.5,7.0,3.0],
              "Maria": [9.5, 8.75, 10.0],
              "Ana": [6.5, 10.0, 7.25],
              "José": [8.5, 9.0, 8.0]}

trabalho_extra = {"João": 9.5,
                  "Maria": 10.0,
                  "José": 10.0,
                  "Pedro": 4.5}

for aluno in trabalho_extra:
    if aluno not in notas_dic:
        notas_dic[aluno] = [0.0,0.0,0.0]
    notas_dic[aluno].append(trabalho_extra[aluno])

print('Médias finais (incluindo trabalho extra):')
for aluno in sorted(notas_dic.keys(), key=locale.strxfrm):
    # Calcula a média
    media = sum(notas_dic[aluno])/len(notas_dic[aluno])
    print("  {0:}:  {1:8.2f}".format(aluno, media))
```

Código 3.3: Testando se elementos fazem parte de um dicionário.

3.2 Interagindo com o sistema de arquivos

Nesta seção vamos aprender algumas funções úteis dos módulos `os`, `os.path` e `shutil`. Estes módulos servem para interagir com o sistema de arquivos listando arquivos e diretórios em um dado caminho, criando ou copiando, movendo ou removendo arquivos, etc. Como sempre, a recomendação é que se verifique a documentação online destes módulos, onde outras funções úteis podem ser encontradas.

A habilidade de listar o conteúdo de um diretório (ou pasta, na nomenclatura do Windows) teria sido útil no exercício 3.1 dado que tivemos que apontar explicitamente o nome de cada arquivo. Podemos, no entanto, produzir uma lista contendo todo conteúdo do diretório utilizando função `os.listdir()`. Esta lista conterá tanto arquivos como (sub)diretórios. É possível checar se um dado item é um arquivo ou diretório utilizando as funções `os.path.isfile(caminho)` e `os.path.isdir(caminho)` que retornam `True` se `caminho` for um arquivo ou diretório,respectivamente. Veja o exemplo abaixo.

```
import os
import os.path
diretorios = []
arquivos = []
for item in os.listdir():
    if os.path.isdir(item):
        diretorios.append(item)
    elif os.path.isfile(item):
        arquivos.append(item)

print("Estamos no diretório: ",os.getcwd())
print("  Ele contém os subdiretórios:\n", end="\t")
for d in diretorios:
    print(d,end=" ")
print("\n  E contém os arquivos:\n", end="\t")
for d in arquivos:
    print(d,end=' ')
print()
```

Exercício 3.7. *Modifique o exercício 3.3 de maneira a usar as funções `os.listdir()` e `os.isfile()` ao invés de construir explicitamente os nomes dos arquivos.*

Exercício 3.8. *A função `os.path.getsize(path)` retorna o tamanho, em bytes, do caminho `path` dado no argumento. Utilizando-a, juntamente com `os.listdir` e `os.isfile`, escreva uma função que recebe um caminho para um diretório como argumento e retorna o nome do maior arquivo no diretório e seu tamanho.*

Suponha que queiramos encontrar o maior arquivo em toda árvore de diretórios de um dado caminho (isto é, olhar não apenas o diretório associado ao caminho do argumento,

mas também os subdiretórios, subsubdiretórios, etc). É possível fazer uma pequena modificação ao exercício 3.8 capaz de resolver este problema.

A solução, apresentada no código 3.4 a seguir, ilustra o uso de um recurso importante: a *recursão*. O problema é solucionado para um único diretório, como no exercício 3.8 porém a mesma função é então chamada, na linha 18, para cada subdiretório. Este truque, de chamar uma função de dentro dela mesma, permite muitos problemas difíceis sejam solucionados de maneira compacta e elegante.

```
1  import os, os.path
2  def maior_arquivo(caminho='.'):
3      """
4      Recebe um caminho para um diretório. Procura qual o maior
5      arquivo no diretório especificado e em todos os seus
6      subdiretórios. Retorna o caminho para o maior arquivo
7      encontrado e seu tamanho.
8      """
9      # Inicializa o tamanho do maior arquivo
10     t_max = -1
11     for item in os.listdir(caminho):
12         # Constrói o caminho para um dado
13         caminho_item = os.path.join(caminho, item)
14         if os.path.isfile(caminho_item):
15             # Obtém o tamanho do arquivo
16             tamanho = os.path.getsize(caminho_item)
17         elif os.path.isdir(caminho_item):
18             # Caso caminho_item apote para um diretório, chama a função
19             # novamente para ele.
20             caminho_item, tamanho = maior_arquivo(caminho_item)
21         # Caso necessário, atualize os dados do maior arquivo encontrado
22         if tamanho > t_max:
23             t_max = tamanho
24             c_max = caminho_item
25     # Se não encontrar nenhum carquivo, t_max será negativo.
26     if t_max<0:
27         return "", 0
28     return c_max, t_max
```

Código 3.4: Uma função recursiva que encontra o maior arquivo em uma árvore de diretórios.

Saber listar diretórios e encontrar arquivos é bastante útil mas só nos traz poder real se formos capazes de também de criar diretórios e mover, copiar ou apagar arquivos e diretórios. Estas ações são parte do módulo `shutil` e estão exemplificadas no próximo

trecho de código.

```
import shutil
import os
import os.path
print("Gerando o arquivo teste.txt.")
f = open("teste.txt","w")
f.write("Isto é um teste!")
f.close()
print(os.listdir())
print("Criando o diretório/pasta teste_dir.")
os.mkdir("teste_dir")
print(os.listdir())
print("Copiando o arquivo para dentro de teste_dir.")
shutil.copy("teste.txt", "teste_dir")
print(os.listdir())
print("Copiando o diretório teste_dir (e conteúdo) para teste_dir_2.")
shutil.copytree("teste_dir", "teste_dir_2")
print(os.listdir())
print("Removendo diretório teste_dir (e seu conteúdo).")
shutil.rmtree("teste_dir")
print(os.listdir())
print("Removendo o arquivo teste.txt")
os.remove("teste.txt")
print(os.listdir())
```

Código 3.5: Exemplo de uso do módulo `shutil`.

Exercício 3.9. *Separe os arquivos que contém o texto secreto do exercício 3.1 em diretórios chamados 'paragrafo1','paragrafo2', etc, agrupando os arquivos que contém, escondidas dentro deles, linhas de um mesmo parágrafo. (N.B. lembre que no texto secreto os parágrafos são separados por linhas vazias.)*

3.3 Reconhecendo padrões com regex

Vamos agora nos debruçar sobre um assunto de grande utilidade prática mas, na verdade, independente da linguagem Python em si. Em um mundo em a quantidade de dados disponíveis aumenta a cada minuto, frequentemente precisamos instruir o computador a percorrer um grande número de arquivos em busca de um padrão.

Existe uma linguagem para especificar buscas por padrões em textos chamada de "*expressões regulares*" ou *regex*. Ela pode ser utilizada em diversos editores de texto, bancos de dados, sistemas de busca e linguagens de programação como Perl, Awk e — é claro —

Python. No Python as regex estão associadas ao módulo `re`, cuja documentação pode ser obtida de https://docs.python.org/3.5/howto/regex.html .

Para codificarmos um padrão de busca com uma regex, alguns caracteres serão tratados de maneira especial, e são chamados de meta-caracteres. São eles:

```
. ^ $ * + ? { } [ ] \  | ( )
```

caso em algum momento queiramos utilizar estes caracteres com seu significado original, basta precedê-los de uma barra (e.g. o significado de `\.` é um ponto mesmo).

Para um primeiro exemplo, vamos tentar escrever uma regex que identifique um número de telefone. O primeiro recurso do qual precisaremos é a ideia de classe de caracteres, que é especificada por colchetes. Sendo assim, um algarismo arábico qualquer `[0123456789]`, que pode ser abreviado por: `[0-9]`, que significa: "a classe composta dos algarismos de 0 a 9". O metacaractere `+` significa 1 ou mais ocorrência, sendo assim um no caso mais simples, um número de telefone poderia ser identificado por

```
[-0-9]+
```

A regex acima, que identifica "uma ou mais ocorrências de algarismos arábicos ou hífen" funcionaria para identificar números como 992312312, 7723-3212, etc, mas não é muito específico: combinações como 1234, -33, 17-28-38 também seriam identificadas. Utilizando as chaves é possível especificar o número de ocorrências de um dado caractere, podemos melhorar nosso reconhecimento de números usando

```
[0-9]{4}[- ][0-9]{4}
```

a regex acima especifica o padrão composto por 4 algarismos, seguidos de um (único) hífen ou espaço, seguido de mais 4 algarismos. Há, no entanto, alguns problemas com esta regex: algumas pessoas simplesmente não utilizariam o espaço ou o hífen, além disso, números de celulares podem conter 5 dígitos ao invés de 4. Estes dois requisitos podem ser satisfeitos pela seguinte expressão

```
\d{4,5}[- ]?\d{4}
```

A interrogação, `?`, significa uma ou nenhuma vez. O `\d` significa digito numérico, ou seja, é um sinônimo de `[0-9]`. Vamos preparar agora uma regex que selecione um DDD. Ela deverá ser capaz de abranger os casos (usando São Paulo como exemplo):

```
(11) (011) 11, 011, (02111), (0xx11)
```

Com o que já vimos acima, isto pode ser conseguido usando

```
\(?0?[x\d]{2,4}\)?
```

Combinando esta expressão com a anterior, temos um bom candidato a regex para detecção de números de telefone brasileiros com DDD.

```
\(?0?[x\d]{2,4}\)? ?\d{4,5}[- ]?\d{4}
```

Código	Significado
`[caracteres]`	Define uma classe de caracteres. E.g. `[ab]` identifica a letra `a` ou `b`. Pode ser usado um hífen para designar um intervalo (e.g. `[0-3]` é o mesmo que `[0123]`).
`[^caracteres]`	Define o complemento de uma classe de caracteres. Exemplo: `[ab]` identifica qualquer caractere diferente de `a` ou `b`
`{n}`	O item anterior é repetido n vezes.
`{min,max}`	O item anterior é repetido entre min e max vezes
`?`	O item anterior aparece 0 ou 1 vez. Sinônimo de `{0,1}`.
`*`	O item anterior aparece 0 ou mais vezes.
`+`	O item anterior aparece 1 ou mais vezes.
`\d`	Digito numérico. Sinônimo de `[0-9]`
`\D`	Não-digito. Sinônimo de `[^0-9]`
`\w`	Parte de uma palavra (word). Sinônimo de `[A-Za-z0-9_]`
`\W`	Não-palavra. Sinônimo de `[^A-Za-z0-9_]`
`\s`	Qualquer tipo de espaço em branco. Sinônimo de `[\t\r\n\v\f]`.
`.`	Qualquer caractere.
`|`	Operador alternativa. Exemplo: `abc|efg` encontra `abc` ou `efg`.
`$`	Fim de uma string. Exemplo: `\w+sa$` identificará (apenas) a palavra `coisa` na string `"Esta casa é uma coisa"`.
`^`	Início de uma string. Exemplo: `^\w+sta` identificará (apenas) a palavra `esta` na string `"Esta casa é uma festa"`.

Tabela 3.2: Principais códigos para expressões regulares.

Exercício 3.10. *Escreva uma expressão regular capaz de identificar placas de carros.*

A primeira função do módulo `re` que veremos é a função `search` que busca um padrão, descrito por uma expressão regular, dentro de uma string.

Se função `re.search` não encontrar o padrão, ela retorna `None` (que é tratado como `False` na condicional). Caso haja uma correspondência, ela retorna um objeto SRE_Match e o resultado encontrado pode ser acessado da maneira indicada na linha 5 do código acima.

A próxima tarefa seria identificar pedaços de padrões. Por exemplo: qual o prefixo do número de telefone encontrado? Não é necessário identificar o número para depois escrever outra expressão regular para o prefixo. É neste momento que entra o uso dos parênteses: eles nos permitem selecionar grupos dentro de um dado padrão. Vejamos um exemplo no código 3.7 a seguir.

```
import re
teste = "Telefone ao menos uma vez para (21) 3334-4333"
matchObj = re.search(r"\(?0?[x\d]{2,4}\)? ?\d{4,5}[- ]?\d{4}", teste)
if matchObj:
    telefone = matchObj.group(0)
    print("Encontrei o telefone: ", telefone)
else:
    print("Não encontrei um telefone!")
```

Código 3.6: Identificando um telefone usando expressões regulares.

```
import re
# Define a frase onde o padrão será busca
frase = "Telefone ao menos uma vez para (21) 3334-4333"
# Define a regex
padrao = r"\(?0?[x\d]{0,2}(\d{2})\)? ?(\d{4,5})[- ]?(\d{4})"
matchObj = re.search(padrao, frase)
if matchObj: # Caso um número seja encontrado
    telefone = matchObj.group(0)
    DDD = matchObj.group(1)
    numero = matchObj.group(2) + "-" + matchObj.group(3)
    print("O telefone é: ", telefone)
    print("O DDD é: ", DDD)
    print("O restante do número: ", numero)
else:
    print("Não encontrei um telefone!")
```

Código 3.7: Exemplo de uso de grupos em expressões regulares.

Como pode ser visto nas linhas 8−10, os grupos demarcados pelos parênteses podem ser acessados utilizando o método `matchObj.group`, utilizando a numeração do grupo como argumento – o primeiro grupo da esquerda para a direita é acessado com `group(1)`, etc.

O método `search` apenas permite que se encontre uma única ocorrência de um dado padrão. Para buscar a mesma expressão regular diversas vezes em uma mesma string pode-se usar o método `re.finditer`, que permite iterar sobre as ocorrências. Veja o código 3.8 para um exemplo. Note que a variável `texto` poderia conter um texto arbitrariamente grande.

Exercício 3.11. *Escreva uma função que recebe o caminho para um arquivo texto contendo um poema (utilize o navegador para encontrar o seu favorito ou algum exemplo) e retorna um dicionário de rimas, isto é, um dicionário que possui a última sílaba de cada verso como chave e, associada a cada chave, uma lista contendo palavras com aquela*

```python
import re
# String de exemplo
texto = "Seu Osório: (021) 3334-4333 (do guarda-chuva do Noel)\n"
texto += "Ana Maria: 0xx11 1234 5678 (amiga do Gabriel)\n"
texto += "Belas Artes: (11)5576-7300\n"
# Define a regex
padrao = r"(\w+ \w+): \(?0?[x\d]{0,2}(\d{2})\)? ?(\d{4,5})[- ](\d{4})"
# Inicializa dicionário que conterá a agenda
agenda = {}
for matchObj in re.finditer(padrao, texto):
    # Analisa a ocorrência
    nome = matchObj.group(1)
    prefixo = "(" + matchObj.group(2) + ") "
    numero = matchObj.group(3) + "-" + matchObj.group(4)
    # Inclui a entrada no dicionário
    agenda[nome] = prefixo + numero
print("Texto original:")
print(texto)
print("Dicionário gerado:", agenda)
```

Código 3.8: Produzindo um dicionário com uma agenda telefônica a partir de um arquivo texto com anotações.

rima. Aplique então esta função aos Lusíadas de Camões[2].

```python
import re
# Data no curioso formato estadunidense
data_eua = "01/10/1984"
# Converte para o padrão ISO
data_iso = re.sub(r'(\d\d)/(\d\d)/(\d{0,2}\d\d)',
                  r'\3-\1-\2', data_eua)
# Converte para o padrão brasileiro
data_br = re.sub(r'(\d\d)/(\d\d)/(\d{0,2}\d\d)',
                 r'\2/\1/\3', data_eua)
print("ISO:", data_iso, "\nBR:",data_br, "\nEUA:",data_eua)
```

Código 3.9: Realizando substituições com o método `re.sub`.

Frequentemente utilizamos expressões regulares para efetuar substituições. Para isto utilizamos o método `re.sub` exemplificado no código 3.10. O método `re.sub` recebe 3 argumentos: o padrão a ser encontrado, um padrão de substituição e a string onde as

[2]Uma versão somente texto dos Lusíadas pode ser obtida, por exemplo, no site do Projeto Gutenberg.

substituições serão efetuadas. O conteúdo de cada grupo é então referenciado utilizando uma barra seguida do número do grupo ao qual nos referimos (por exemplo, o conteúdo do primeiro grupo é marcado por `\1`, o do segundo por `\2`, etc).

Um último recurso útil é o método `re.split`. Ele permite que quebrar uma string a cada ocorrência de um dado padrão. Veja o exemplo abaixo.

```
import re
frase = "separe...as palavras, se; for capaz"
palavras = re.split(r'\W+',frase)
print(palavras)
```

Código 3.10: Utilizando o método `re.split`.

Projeto "detetive" A habilidade recém adquirida de percorrer um sistema os diretórios de um computador, abrindo diversos arquivos e procurando padrões em seu interior permite diversos tipos de investigações. Isto facilita o trabalho de um "detetive" contemporâneo, que será o tema desta atividade. Sugere-se a formação de dois grupos: um deles terá que esconder uma informação em um conjunto de arquivos texto e pastas (exercício 3.12), enquanto o outro grupo terá que encontrá-la (exercício 3.13). Depois, os papeis podem ser trocados.

Exercício 3.12. *Escreva um programa que crie uma estrutura de diretórios e subdiretórios com até 6 (i.e. 5 níveis de pastas e subpastas). Espalhados pelos diretórios distribua arquivos de texto, que devem conter fragmentos escolhidos ao acaso de algum (ou mais de um) texto bem grande (baixado, por exemplo, do Projeto Gutenberg). Produza, no mínimo, algumas centenas de arquivos (aventureiros com espaço em disco podem optar por números maiores). Em pontos* aleatórios *destes arquivos texto, distribua nomes e placas de carro, seguindo os modelos: 'Fulano Beltrano: XYZ 9875', 'Sicrano, placa XYZ9875', 'Zetrano (XYZ-9875)' incluindo pequenas variações (com ou sem hífen, com ou sem espaço. Esconda cerca de 10 nomes e cerca de 15 placas de carro (alguns nomes devem estar associados a mais de uma placa) nos muitos arquivos gerados. Dica: para tornar as coisas mais imprevisíveis, talvez seja útil/interessante utilizar a função* `random.randint`*, que retorna um número inteiro aleatório (consulte a documentação para ver como usá-la). A escolha dos nomes dos arquivos e diretórios é livre (seja criativo).*

O resultado do exercício 3.12 deve ser utilizado para alimentar o programa do exercício 3.13, a seguir.

Exercício 3.13. *Sua tarefa escrever um programa capaz de encontrar um conjunto de placas de carro e os nomes a elas associados que estão dispersos entre muitos arquivos de texto, que por sua vez podem estar dentro de muitos (sub..)subdiretórios. O formato dos nomes e placas pode ser lido no exercício 3.12. O programa deve construir dois dicionários: um que contenha os nomes como chaves e listas de placas como valor (i.e. fica possível ao detetive saber que carros cada "suspeito" possui); outro que contenha a placas dos carros como chaves e uma lista com os nomes dos arquivos em que elas aparecem como valor (i.e. é possível dizer em quais documentos uma dada placa de carro aparece). Dica: inicie escrevendo a expressão regular adequada (adaptando o exercício 3.10) e depois adapte o código 3.4 para este propósito.*

4 Classes e objetos

Objetivos de aprendizagem deste capítulo

- Compreender os conceitos de classe e objeto,
- Escrever uma classe simples e utilizá-la,
- Compreender e utilizar a ideia de herança,
- Explorar propriedades de objetos,
- Ler e gravar objetos em disco.

Neste capítulo vamos abordar uma maneira diferente (frequentemente tida como mais avançada) de programar, na qual sempre abordamos um problema enfatizando propriedades comuns dos dados que estamos representando no computador e ações que são executadas exclusivamente sobre este conjunto de dados. Há um conjunto de técnicas e uma sintaxe especial para se trabalhar desta maneira, que recebe o nome de *Programação Orientada a Objetos*.

4.1 Definindo classes

Vamos introduzir o assunto examinando um exemplo concreto de como se define uma classe em Python e como é possível utilizá-la. Para isto, concentremo-nos no código 4.1 que define uma classe `Pessoa`. Antes de mais detalhes (analisaremos o código linha por linha logo), vejamos como é o *uso* da classe `Pessoa`. Para isto, execute o código 4.1 e continue no modo interativo (no IDLE, basta executar com Run Module e continuar na janela do shell). Execute então o comando:

```
>>> individuo_A = Pessoa("José Maria Santos")
```

Nesta linha foi criada uma *instância* da classe `Pessoa`, isto é, foi criado um *objeto* que cujas características são especificadas pela classe `Pessoa`. Note, apesar do nome estranho, a *instanciação* não é algo completamente novo: a primeira vez que executamos uma operação parecida foi na página 11, quando criamos uma instância do labirinto!

Há dois *atributos* associados a este objeto. Eles podem ser acessados através de

```
>>> individuo_A.ultimo_nome
>>> individuo_A.primeiros_nomes
```

```python
class Pessoa(object):
    """ Uma classe para representar pessoas """
    def __init__(self, nome):
        """ Inicializa o tipo pessoa, recebendo um nome """
        # Monta uma lista quebrando a string com o nome
        nomes = nome.split(" ")
        if len(nomes) > 1:
            # Junta os primeiros nomes e os armazena
            self.primeiros_nomes = " ".join(nomes[:-1])
            # Armazena o último nome
            self.ultimo_nome = nomes[-1]
        else:
            # Caso haja apenas 1 nome, trata como sobrenome
            self.primeiros_nomes = None
            self.ultimo_nome = nome

    def nome_completo(self):
        if self.primeiros_nomes:
            return self.primeiros_nomes + ' ' + self.ultimo_nome
        else:
            return self.ultimo_nome
```

Código 4.1: Uma classe simples que armazena o nome e sobrenome de uma pessoa.

Note que se trata da mesma sintaxe que utilizamos para acessar variáveis de um módulo (e.g. código 2.8).

Funções associadas a objetos costumam ser chamadas de *métodos*. É possível acessar o método `nome_completo` através de:

```
>>> individuo_A.nome_completo()
```

Mais uma vez, a sintaxe é muito semelhante àquela usada para acessar as funções de um módulo. Há porém uma diferença crucial: é possível criar mais de uma instância da classe `Pessoa`, cada uma com atributos completamente independentes. Por exemplo, se executarmos as linhas

```
>>> individuo_B = Pessoa('Maria José Silva')
>>> individuo_B.nome_completo()
>>> individuo_A.nome_completo()
```

conseguimos obter o nome completo de cada uma destas pessoas. Não é possível alcançar este tipo de efeito com variáveis de um módulo. Enquanto um módulo importado é único, os objetos armazenados nas variáveis `individuo_A` e `individuo_B` são independentes – muito embora compartilhem a mesma estrutura, que é ditada pela classe `Pessoa`.

Vemos, assim, que um *objeto* é algo que agrupa *dados*, na forma de atributos (neste caso

`primeiros_nomes` e `ultimo_nome`), e *métodos* para manipular estes dados (neste caso o humilde método `nome_completo`).

Execute agora a linha

```
>>> type(individuo_A)
```

neste momento, alguns talvez tenham se lembrado do resultado dos primeiros experimentos com a função `type` e tenham percebido a magnitude do passo que demos: quando criamos uma classe nova, estamos criando um novo *tipo de dados*. De fato, veremos que é possível utilizar funções especiais para definir o que significam para este tipo novo, os diversos operadores (+, <=, etc) que vimos no início do primeiro capítulo.

Agora que temos uma ideia de qual o uso da (ainda bem limitada) classe `Pessoa`, vamos agora nos debruçar sobre o código 4.1. A primeira linha informa ao interpretador que se iniciará a definição de uma classe. O método especial `__init__` inicializa o objeto, neste caso separando o nome completo em duas strings, uma contendo o último nome e outra contendo os primeiros nomes, que são gravadas em dois atributos (o caso em que um único nome é digitado é tratado em separado).

Note que todos os métodos devem receber a palavra chave `self` como primeiro argumento. Esta representa a instância que é produzida, que é transmitida automaticamente a todos os métodos quando eles são invocados. O armazenamento de valores em atributos também envolve a palavra `self` (por exemplo `self.ultimo_nome`).

No código 4.2 a classe `Pessoa` foi estendida de maneira a incluir a data de nascimento da pessoa. Quando o objeto é criado, o atributo `nascimento` recebe `None`, na linha 17. Este procedimento reflete o fato de que, embora todas as pessoas possuam uma data de nascimento, nem sempre a conhecemos. O atributo só recebe um valor quando é invocado o método `define_nascimento`, que recebe os argumentos dia, mês e ano. Este método, definido na linha 19, utiliza o módulo `datetime` para representar datas (na nomenclatura recém adquirida: estamos armazenando, no atributo `nascimento`, uma instância da classe `datetime.date` contendo a data de nascimento).

Na linha 23, há um novo método, nomeado `idade`, que calcula e retorna a idade da pessoa (note que isto serve como um exemplo do uso do módulo `datetime` – mais informações sobre este módulo podem ser obtidas, como sempre, da documentação).

Após executar o código 4.2, vamos mais uma vez examiná-lo no modo interativo.

```
>>> fulano = Pessoa('Fulano Sicrano Beltrano')
>>> fulano.define_nascimento(10,1,1984)
>>> anos = fulano.idade()
>>> print("O Sr. {0} tem {1} anos.".format(fulano.ultimo_nome, anos))
```

Conforme o esperado, estas linhas imprimem o sobrenome e a idade do objeto `Pessoa` gerado. Vamos experimentar imprimir o objeto como um todo:

```
>>> print(fulano)
```

```python
import datetime
class Pessoa(object):
    """ Uma classe para representar pessoas """
    def __init__(self, nome):
        """ Inicializa o tipo pessoa, recebendo um nome """
        # Monta uma lista quebrando a string com o nome
        nomes = nome.split(" ")
        if len(nomes) > 1:
            # Junta os primeiros nomes e os armazena
            self.primeiros_nomes = " ".join(nomes[:-1])
            # Armazena o último nome
            self.ultimo_nome = nomes[-1]
        else:
            # Caso haja apenas 1 nome, trata como sobrenome
            self.primeiros_nomes = None
            self.ultimo_nome = nome
        self.nascimento = None # Inicializa o atributo nascimento

    def define_nascimento(self, dia, mes, ano):
        """ Recebe inteiros dia, mês e ano e armazena a data de nascimento """
        self.nascimento = datetime.date(ano, mes, dia)

    def idade(self):
        """ Retorna a idade em anos """
        # Calcula o número de dias desde o nascimento
        dias_de_idade = (datetime.date.today() - self.nascimento).days
        return int(dias_de_idade/365.25) # Retorna o número de anos

    def nome_completo(self):
        if self.primeiros_nomes:
            return self.primeiros_nomes + ' ' + self.ultimo_nome
        else:
            return self.ultimo_nome

    def __lt__(self, other):
        """ Define a comparação entre pessoas utilizando nomes e sobrenomes."""
        if self.ultimo_nome != other.ultimo_nome:
            return self.ultimo_nome < other.ultimo_nome
        else:
            return self.primeiros_nomes < other.primeiros_nomes

    def __str__(self):
            """ Retorna o valor que a Pessoa terá se utilizada as funções
            str ou print """
            pessoa_str = "<Pessoa>\nNome: "+self.nome_completo()
            if self.nascimento:
                pessoa_str += "\nNascimento: "+str(self.nascimento)
            return pessoa_str
```

Código 4.2: Incluindo funcionalidades à classe do código 4.1.

Vejamos ainda mais um exemplo.

```
>>> x= str(fulano)
>>> print(x)
```

Fica claro assim, o papel do método especial `__str__`: ele instrui o Python sobre o que fazer caso uma string descrevendo este objeto seja solicitada, seja pela da função `str`, seja pela função `print` (experimente remover este método e re-executar as linhas anteriores para ver o que acontece).

Finalmente chegamos no misterioso método `__lt__`. Seu nome vem de *less than* (menor que, em inglês). Experimente as seguintes (estranhas) linhas:

```
>>> 17.0.__lt__(42.0)
>>> 42.0.__lt__(17.0)
```

A notação `42.0 < 17.0` em Python é na verdade um atalho para a linha acima, sendo que o método especial `__lt__` diz ao interpretador como ele deve proceder para fazer a comparação.

No caso da classe `Pessoa`, `__lt__` está definida de maneira que uma pessoa será "menor"que outra se seu último sobrenome for menor, como pode ser visto na linha 38. Definir esta comparação é útil porque ela é usada pela função `sorted`. Veja o seguinte exemplo.

```
>>> pessoas = [Pessoa("Fulano Sicrano"), Pessoa("João Almeida"),
               Pessoa("Maria Flores")]
>>> for p in sorted(pessoas):
        print(p)
```

Ou seja, definindo `__lt__` tornamos simples a tarefa de se ordenar alfabeticamente (por sobrenome) uma lista de pessoas[1].

Exercício 4.1. *Inclua, na classe `Pessoa`, os atributos `peso` e `altura`, juntamente com métodos para defini-los (análogos a `define_nascimento`). Acrescente também um método que retorna o índice de massa corpórea da pessoa (i.e. seu peso dividido pelo quadrado da altura) e outro que retorna o número aproximado de vezes que a pessoa respirou durante a vida (em média a taxa de respirações humana é de ≈ 20 respirações por minuto).*

Exercício 4.2. *Vamos voltar ao exemplo do baralho neste exercício. Escreva uma classe `Carta`, que contenha os atributos `naipe` e `valor`. Ajuste os método `__str__` de maneira que a conversão da carta em strings seja razoável. Ajuste também o método `__lt__` para permitir ordenação das cartas (escolha se será por naipe e depois por valor ou vice-versa).*

[1] Analogamente, é possível (re)definir o comportamento dos operadores `<=`, `>`, `>=`, `==`, `!=`, `+`, `-`, `*` e `/` através dos métodos `__le__`, `__gt__`, `__ge__`, `__eq__` e `__ne__`, `__add__`, `__sub__`, `__mul__` e `__truediv__`, respectivamente.

Exercício 4.3. *Escreva uma classe* `Baralho` *que contenha uma lista de objetos da classe* `Carta`*, definida no exercício 4.2. Inclua um método* `embaralha` *e um método* `ordena` *(que recebe o valor booleano* `por_naipe` *como argumento opcional). Inclua também os métodos* `retira` *e* `acrescenta`*, para retirada ou acréscimo de uma carta. Mais uma vez, ajuste o método* `__str__` *para que a função* `print` *gere uma saída razoável. Finalmente, experimente alterar o método* `__add__` *de maneira que adição de dois objetos* `Baralho` *(e.g.* `baralho_A + baralho_B`*) resulte na soma dos baralhos.*

4.2 Subclasses: herdando propriedades

Suponha que queiramos incluir uma classe que para representar um aluno. Este aluno possui um número de matrícula, está inscrito em uma série de disciplinas (e obtém diferentes notas nestas) e além disso[2], o aluno é uma pessoa (i.e. terá um nome, sobrenome, data de nascimento, etc.). Existe uma maneira de expressar este último ponto de maneira transparente, através da ideia de *herança*.

Na linha 2 do código 4.3 note que aparece o nome da classe `Pessoa`. Isto indica que a classe `Aluno` é uma *subclasse*, isto é, ela herda os atributos e métodos da classe `Pessoa`, que neste contexto é chamada de *superclasse*. Consequentemente, se executarmos o código 4.3, as seguintes operações também são válidas

```
>>> a = Aluno("Fulano Sicrano")
>>> a.define_nascimento(1,1,1995)
>>> print('O aluno {0} (matrícula {1}) tem {2} anos'.format(
                        a.nome_completo(), a.num_matricula, a.idade()))
```

isto é, os métodos `nome_completo`, `idade` (e também os atributos `ultimo_nome`, etc) estão presentes na subclasse `Aluno` assim como novos métodos e atributos, como atributo `num_matricula`. Isto permite que a complexidade dos objetos possa ser aumentada aos poucos, definindo classes mais e mais específicas.

Se definirmos, na subclasse, um método com o mesmo nome que um método presente na superclasse, o método recém-definido será utilizado e o método anterior será ignorado[3]. Há porém, casos em que queremos utilizar um método da superclasse e estendê-lo. É possível lidar com isso utilizando o procedimento demonstrado na linha 7, isto é, invocando diretamente o método da superclasse. Sendo assim, a linha 7 chama o método `__init__` como ele foi definido na classe `Pessoa`, i.e. ele é responsável por quebrar o nome da pessoa em `primeiros_nomes` e `ultimo_nome`, além de inicializar a variável `nascimento`. Nas linhas 9 a 13, procede-se com inicialização de atributos específicos de um `Aluno`.

[2] Ao menos enquanto a computação, a robótica ou o programa SETI não progredirem um pouco mais...
[3] Isto é o que chamamos de *sobrecarga* ou *overload*.

```
1  import statistics
2  class Aluno(Pessoa):
3      """ Uma classe para representar alunos """
4      # Variável de classe que armazenará o número instâncias gerado
5      contador_matriculas = 0
6      def __init__(self, nome):
7          Pessoa.__init__(self,nome)
8          # Incrementa o número de alunos
9          Aluno.contador_matriculas += 1
10         # Utiliza este número como número de matrícula deste aluno
11         self.num_matricula = Aluno.contador_matriculas
12         # Inicializa o atributo disciplinas
13         self.disciplinas = {}
14
15     def inclui_disciplina(self, nome_da_disciplina):
16         """ Recebe um nome de disciplina e a inclui """
17         if nome_da_disciplina not in self.disciplinas:
18             self.disciplinas[nome_da_disciplina] = []
19
20     def inclui_nota(self, nome_da_disciplina, nota):
21         """ Recebe um nome de disciplina e uma nota
22             e a inclui a nota (incluindo a disciplina se necessário)"""
23         if nome_da_disciplina not in self.disciplinas:
24             self.inclui_disciplina(nome_da_disciplina)
25         self.disciplinas[nome_da_disciplina].append(nota)
26
27     def lista_disciplinas(self):
28         """ Retorna uma lista contendo as disciplinas nas quais
29             o aluno está matriculado """
30         return sorted(self.disciplinas.keys())
31
32     def medias(self):
33         """ Retorna um dicionário contendo as médias """
34         medias_dic = {}
35         for d in self.disciplinas:
36             if len(self.disciplinas[d])>0:
37                 medias_dic[d] = statistics.mean(self.disciplinas[d])
38
39     def __lt__(self, other): # Ordena alunos por número de matrícula
40         return self.num_matricula < other.num_matricula
```

Código 4.3: A classe `Aluno`, uma subclasse da classe `Pessoa`.

Na linha 5 há a definição de uma *variável de classe* (note a ausência da palavra `self`). Como o próprio nome diz, uma variável de classe está associada a uma classe e não a uma instância. Isto significa que todos objetos compartilharão o valor desta variável. No nosso caso, este recurso é utilizado como um contador: a cada instância gerada, i.e. a cada vez que incluimos os dados de um aluno no sistema, a variável `contador_matriculas`

é incrementada. O valor desta variável no instante da instanciação é armazenado na variável (do objeto) `num_matricula`, na linha 11. Desta forma, cada aluno terá um número de matrícula diferente, associado à ordem com que os alunos foram inscritos no sistema.

Exercício 4.4. *Modifique a classe `Aluno` de maneira que a impressão do objeto aluno (e.g. `print(aluno_a)`) mostre o número de matrícula, as disciplinas e suas respectivas médias (e um aviso quando o aluno estiver cadastrado na disciplina mas a lista de notas estiver vazia). Também devem estar incluídas as informações que são impressas por padrão no caso da classe `Pessoa` (i.e. nome, sobrenome e, possivelmente, data de nascimento).*

Exercício 4.5. *Escreva uma classe `Disciplina` que contém como atributos o nome da disciplina, a descrição e uma lista contendo objetos `Aluno`. A classe deve incluir um método `notas_da_turma`, que retorna um dicionário contendo os números de matrícula dos alunos como chave e a lista de notas obtidas por cada aluno como valor (isto deve ser lido dos objetos `Aluno`).*

Exercício 4.6. *Inclua na classe `Disciplina` métodos que mostrem a distribuição de notas obtidas pela turma: um método `media`, que retorna a média média das notas de toda a turma; um método `mediana`, que calcula a mediana, e um método `desvio_padrao`, que calcula o desvio padrão. Dica: se necessário, use o módulo `statistics`.*

4.3 Explorando objetos

Existem algumas funções que nos permitem descobrir informações a respeito de um dado objeto. Por exemplo, a função `dir` permite que inspecionemos quais são os atributos e métodos associados um objeto. Vamos usar como exemplo a classe `Pessoa`, definida no código 4.2:

```
>>> pessoa = Pessoa("Fulano Sicrano")
>>> dir(pessoa)
```

a saída é uma lista contendo os nomes de todos os atributos e métodos associados a este objeto. Este recurso é útil quando estamos trabalhando no modo interativo com um objeto do qual sabemos pouco, ou quando tentamos nos lembrar do nome de um dado atributo ou método.

A função `dir` também funciona para módulos: em Python, módulos também são objetos (instâncias da classe `module`). Experimente, por exemplo, importar o módulo `random` e observar a saída do comando `dir(random)`.

É possível inspecionar e manipular atributos de um objeto através das funções `hasattr`, `getattr` e `setattr`. Veja o exemplo a seguir.

```python
import datetime

class Veiculo(object):
    def pode_circular_em_sp(self):
        """ Verifica se o veículo pode circular em São Paulo agora """

        # Há uma placa definida?
        if hasattr(self, "placa"):
            # Seleciona último caractere da placa e converte em string
            ultimo_digito = int(self.placa[-1])

            # Prepara um objeto com data e hora atuais
            hoje = datetime.datetime.today()
            hora = hoje.hour # Hora atual (inteiro)
            # Dia da semana = segunda->0, domingo->6
            dia_da_semana = hoje.weekday()

            # Pequeno ajuste para a próxima fórmula funcionar
            if ultimo_digito==0:
                ultimo_digito = 10

            # Se for horário e dia do rodízio do veículo, retorna True
            if ((7<hora<10 or 17<hora<20) and
                dia_da_semana<5 and
                (ultimo_digito==(dia_da_semana+1)*2 or
                 ultimo_digito==(dia_da_semana+1)*2-1)):
                 return False
        return True

if __name__ == "__main__":
    atributo_extra = "Categoria"

    carro = Veiculo()
    setattr(carro, atributo_extra, "Automovel")
    carro.placa = "ABC1233"

    veiculos = [carro]
    for t in ("Carroça", "Bicicleta", "Tico-Tico"):
        v = Veiculo()
        setattr(v, atributo_extra, t)
        veiculos.append(v)

    for i, v in enumerate(veiculos):
        print("\nVeiculo ", i, end="\t")
        print(atributo_extra+":", getattr(v, atributo_extra))
        if v.pode_circular_em_sp():
            print("\tPode circular em São Paulo agora!")
        else:
            print("\tNão pode circular em São Paulo agora!")
```

Código 4.4: Exemplo de uso das funções `hasattr`, `getattr` e `setattr`.

Na linha 8 do código 4.4 verificamos se o próprio objeto (representado pela palavra chave `self`) contém um atributo `placa`: a função `hasattr` retorna `True` se o atributo existir. Neste caso, o método `pode_circular_em_sp` verifica se o veículo pode circular em São Paulo.

Na linha 34, está exemplificado o uso da função `setattr`, que recebe 3 argumentos: o objeto, uma *string* contendo o nome do atributo e o valor a ser atribuído. Note que a linha

```
setattr(carro, "Categoria", "Automovel")
```

é equivalente a

```
carro.Categoria = "Automovel"
```

Finalmente, na linha 45, há um exemplo do uso da função `getattr`, que recebe o objeto, uma string com o nome do atributo e retorna o valor.

Exercício 4.7. *Escreva uma classe `Automovel` que seja subclasse de `Veiculo`. Um `Automovel` deve conter os atributos `placa` e um atributo `numero_de_portas` desde a inicialização. Escreva também uma classe `Motocicleta` que seja também subclasse de `Veiculo`, que deve incluir um atributo `placa` e um atributo `cilindradas`.*

Vamos utilizar o resultado do exercício anterior 4.7 para ilustrar a função `isinstance`. Em um shell onde estão definidas as classes `Veiculo`, `Automovel` e `Motocicleta`, execute as seguintes linhas

```
>>> carro = Automovel(placa = 'ABC1234', numero_de_portas = 4)
>>> isinstance(carro, Veiculo)
>>> isinstance(carro, Automovel)
>>> isinstance(carro, Motocicleta)
>>> moto = Motocicleta(placa = 'ZYX0987', cilindradas = 300)
>>> isinstance(moto, Motocicleta)
>>> isinstance(moto, Veiculo)
```

Assim, a função `isinstance` permite que testemos se um dado objeto é instância de uma dada (sub)classe.

4.4 Gravando objetos

Até agora vimos como é possível escrever novas classes, criar objetos e obter informações sobre objetos. Descobrimos que objetos permitem que organizemos os dados e a maneira como lidamos com eles em nossos programa. Vimos também que quase tudo em Python é um objeto: números, strings, listas, dicionários e até mesmo módulos ou funções – se

ainda estiver em dúvida sobre isto, experimente digitar (analogamente fizemos no primeiro capítulo) `type(print)` para descobrir de que classe a função `print` é uma instância ou `dir(print)` para descobrir seus atributos "secretos" (e.g. `print.__name__`).

No capítulo 3 vimos como criar e interagir com arquivos de texto. Quando os dados com os quais estamos trabalhando isto é *frequentemente uma boa ideia*: arquivos de texto puro podem ser lidos por praticamente qualquer aplicativo. É simples, por exemplo, escrever um arquivo texto contendo uma tabela separa por vírgulas (geralmente com a extensão `.csv`), que pode ser importado por um programa de edição de planilhas (como o LibreOffice Calc).

No entanto, seria interessante que houvesse uma maneira de gravar diretamente os objetos com os quais trabalhamos dentro de nossos programas, de maneira a aproveitar tanto o tempo investido nos dados em si, como também o de organizá-los de maneira estruturada. A conversão de objetos em algo que possa ser transmitido ou gravado em disco é o que costuma ser chamado de *serialização* no contexto de computação. No caso do Python, isto pode ser feito utilizando o módulo `pickle`.

```
1  import pickle
2
3  # Define um dicionário
4  um_dicionario = {'Chave': "Valor", 'Lista': [1,2,3,4]}
5  # Abre o arquivo dicionario.pickle para gravação
6  arquivo = open('dicionario.pickle', 'wb') # Note que o modo é 'wb'!
7  # Grava o dicionário exemplo no arquivo
8  pickle.dump(um_dicionario, arquivo)
9  # Fecha o arquivo!
10 arquivo.close()
11
12 # Define um objeto Automovel (assumindo presente a classe Automovel)
13 um_carro = Automovel(placa = 'ABC1234', numero_de_portas = 4)
14 # Esta é uma outra maneira de abrir arquivos para gravação
15 with open('carro.pickle', 'wb') as arquivo:
16     # Pickle the 'data' dictionary using the highest protocol available.
17     pickle.dump(um_carro, arquivo)
18     # Quando saímos do bloco with, arquivo é fechado automaticamente:
19     # um bom recurso para evitar esquecimentos.
```

Código 4.5: Exemplo do uso do módulo `pickle` para gravar (serializar) objetos.

No código 4.5 está exemplificado como armazenar[4] um dicionário e uma instância do objeto `Veiculo` em arquivos, utilizando a função `pickle.dump`. Note que o modo de

[4] A comunidade Python costuma utilizar o verbo *to pickle* (que significa algo como: "preparar uma conserva") para o ato de serializar um objeto e armazená-lo usando o módulo `pickle`. Para operação contrária utiliza-se o verbo *to unpickle*.

abertura dos arquivos é `'wb'`, onde a letra 'b' corresponde a bytes. Também é exemplificado a sintaxe do bloco `with` (que é uma maneira conveniente de não esquecer de fechar os arquivos). O código 4.6 mostra a operação inversa: como carregar um objeto previamente armazenado no disco.

```
1  import pickle
2  # Abre o arquivo dicionario.pickle para leitura
3  arquivo = open('carro.pickle', 'rb') # Note que o modo é 'rb'!
4  # Carrega o objeto do arquivo na variável carro
5  carro = pickle.load(arquivo)
6  # Fecha o arquivo
7  arquivo.close()
8
9  print('Carro carregado:', carro)
10 if carro.pode_circular_em_sp():
11     print('... e ele pode circular em São Paulo agora.')
12
13 # Esta é a maneira de ler arquivos usando um bloco with
14 with open('dicionario.pickle', 'rb') as arquivo:
15     # Pickle the 'data' dictionary using the highest protocol available.
16     dicionario = pickle.load(arquivo)
17     # Quando saímos do bloco with, arquivo é fechado automaticamente.
18 print('Dicionário carregado:',dicionario)
```

Código 4.6: Exemplo do uso do módulo `pickle` para carregar objetos gravados em disco.

Exercício 4.8. *Neste exercício faremos uso da classe* `Disciplina` *dos exercícios 4.5 e 4.6. Escreveremos que permita uso daquela classe na prática.*

- *Ao iniciar, programa deve solicita o nome (ou código) de uma disciplina.*
- *Caso não exista um arquivo correspondendo àquele nome (e.g. 'programação.pickle'), o programa deve criar um objeto* `Disciplina`*, solicitando ao usuário que digite a descrição.*
- *Caso o arquivo exista, o objeto* `Disciplina` *contido no arquivo deve ser carregado.*
- *O programa deve, então perguntar se o usuário gostaria de adicionar um novo aluno e solicitar os dados do aluno (criando um objeto* `Aluno`*), incluindo as notas na disciplina.*
- *Quando o usuário não quiser mais acrescentar alunos, o programa deve imprimir a média, a mediana e o desvio padrão das notas da turma.*
- *Antes de sair o programa deve gravar a disciplina em disco.*

(Note que a variável de classe `contador_matriculas` *não será automaticamente armazenada: a cada vez que executarmos o programa para acrescentar alunos, o contador será reinicializado. Procure uma alternativa que garanta a funcionalidade do contador de matrículas.)*

5 Esboços eletrônicos: o ambiente Processing

Objetivo de aprendizagem deste capítulo

- Fazer esboços eletrônicos estáticos utilizando formas básicas,
- Preparar esboços animados e interativos,
- Organizar o código dos esboços utilizando funções e classes.

O Processing é um ambiente de desenvolvimento que permite a produção de conteúdo visual e/ou interativo a partir de código relativamente simples. A maneira tradicional de especificar o código no Processing é utilizando a sintaxe da linguagem Java. Há, no entanto, um *modo* Python, que permite que usemos o que aprendemos de Python dentro do Processing.[1]

5.1 Esboços estáticos – a vila

Vamos começar examinando um exemplo: inicie um novo sketch do Processing e digite o código 5.1. Os esboços do Processing são feitos através de uma série de funções pré-definidas que desenham elementos na tela e alteram as propriedades de um "desenhista". O código digitado na tela branca do Processing pode ser executado clicando-se no botão triangular (de *play*) no canto superior esquerdo da janela. Qualquer saída não visual do código (gerada, por exemplo, pela função `print`) será impressa na tela preta abaixo.

Na linha 2 do código 5.1, a função `size` define o tamanho da janela de desenho. A linha 4, determina a cor do fundo de nosso desenho: as cores no Processing podem ser especificadas sempre como um trio de inteiros (entre 0 e 255) correspondendo aos valores RGB – e se fornecermos um quarto parâmetro, este controlará a transparência. Na linha 6, define-se a cor do contorno: após esta linha, qualquer desenho que for feito terá borda desta cor (o "desenhista" mergulhou o pincel nesta tinta). Também é possível instruir o processing a não desenhar bordas, através da função `noStroke()`. Na linha 8 é definida a espessura da linha.

[1] O modo Python pode ser instalado clicando-se no botão no canto superior direito (onde originalmente está escrito Java) e selecionando Add Mode...

```
1  # Define o tamanho da janela
2  size(200,200)
3  # Define a cor de fundo (vermelho, verde, azul)
4  background(250,230,210)
5  # Define a cor do contorno (vermelho, verde, azul)
6  stroke(139, 69, 19)
7  # Define a espessura do contorno
8  strokeWeight(3)
9  # Desenha linhas usando a sintaxe:
10 # line(x1, y1, x2, y2)
11 line(100, 120, 100, 85) # Tronco
12 line(100, 120, 85, 150) # Perna E
13 line(100, 120, 115, 150) # Perna D
14 line(100, 105, 80, 90) # Braço E
15 line(100, 105, 120, 90) # Braço D
16 # Especifica cor de preenchimento
17 fill(139, 69, 19)
18 # elllipse(x_centro, y_centro, largura, altura)
19 ellipse(100,75,21,25)
```

Código 5.1: Um primeiro esboço com o Processing.

Nas linhas 11 a 15 do código 5.1 pedimos ao Processing que desenhe algumas linhas. Isto é feito especificando-se as coordenadas de início e término das linhas. O sistema de coordenadas do Processing tem a origem no canto superior esquerdo da janela (isto é, no canto superior esquerdo, $x = 0$ e $y = 0$).

Na linha 17, definimos a cor de preenchimento, ou seja, a partir deste ponto, toda instrução que envolver um preenchimento utilizará esta cor. Se quisermos desenhar formas vazadas (sem preenchimento) basta utilizar a função `noFill()`.

Na última linha, desenhamos uma elípse. Os argumentos da função `ellipse` são as coordenadas do ponto central, a largura e altura da elípse que se quer desenhar. Esta elípse finaliza nosso desenho, acrescentando uma cabeça à nossa personagem.

Há uma outra maneira de desenhar uma elípse, através da função `ellipseMode`, podemos alterar o comportamento da função `ellipse`. Veja o exemplo abaixo.

```
1 ellipseMode(CORNER)
2 # ellipse(x_borda, y_borda, largura, altura)
3 ellipse(100-21/2.0,75-25/2.0,21,25)
```

Este código é equivalente à linha final do código 5.1. O modo `CORNER` permite que especifiquemos o canto superior esquerdo da elípse ao invés de seu centro. Há ainda o modo `CORNERS`, que permite que utilizemos os dois cantos para desenhar a elípse.

```
ellipseMode(CORNERS)
# ellipse(x_borda1, y_borda1, x_borda2, y_borda2)
ellipse(100-21/2.0,75-25/2.0,100+21/2.0,75+25/2.0)
```

E caso queiramos retornar ao modo original, basta utilizar:

```
ellipseMode(CENTER)
ellipse(100,75,21,25)
```

A sintaxe para desenharmos retângulos é idêntica à utilizada para desenhar elípses, utilizando-se as funções `rect` e `rectMode`. Se substituíssemos (experimente) a linha final do código 5.1 pelo trecho a seguir, tornaríamos a cabeça de nosso personagem quadrada.

```
# Desenhando um retângulo
rectMode(CENTER)
rect(100,75,21,25)
```

A função `rect` suporta ainda um argumento extra, que caracteriza quão arredondadas são as bordas do retângulo. Experimente as linhas abaixo.

```
# Desenhando um retângulo
rectMode(CENTER)
rect(100,75,21,25,20)
```

Exercício 5.1. *Desenhe um chapéu para a personagem do código 5.1, utilizando as funções* `fill`, `ellipse` *e* `rect`.

No código 5.2 há um outro exemplo de esboço, no qual desenhamos um cenário simples, composto de céu, gramado e casas. Note que para o desenho das casas definimos uma função `desenha_casa`. Assim como a definição de funções, toda sintaxe e estruturas do Python que vimos anteriormente (por exemplo: condicionais, laços, listas, dicionários) podem ser utilizadas no Processing.

A função `random` do Processing[2] é utilizada para introduzir aleatoriedade no desenho: cada vez que rodarmos o código, um percentual diferente da altura da casa será dedicado ao telhado.

Exercício 5.2. *Modifique o código 5.2 de maneira a incluir uma porta e uma janela em cada casa. Faça também qualquer ajuste estético que julgar necessário.*

[2] Também poderíamos ter importado uma função equivalente do módulo `random` da biblioteca padrão do Python.

```
import math
razao_aurea = (1.0 + math.sqrt(5))/2.0
size(480,380) # Especifica o tamanho de uma janela
# Especifica a cor do fundo utilizando o código RGB
background(0, 191, 255) # Azul celeste
noStroke() # Sem linha de contorno
fill('#4DBD33') # Cor de preenchimento (verde)
rect(0,200,480,380) # Desenha um retângulo (o chão)

def desenha_casa(x, y, altura_total, percentual_beiral=0.07,
                 percentual_de_telhado=0.4):
    """
    Desenha uma casa. Recebe as coordenadas x, y, a altura
    total e (opcionalmente) o percentual da altura
    correspondente ao telhado e o percentual da largura
    associado ao beiral.
    """
    # A casinha terá proporção aurea!
    largura = altura_total * razao_aurea
    # Percentual da altura que será telhado
    altura_telhado = percentual_de_telhado * altura_total
    # O telhado será ligeiramente mais largo, por conta do beiral
    extra = largura * percentual_beiral
    fill(255, 51, 0) # Preenchimento laranja-avermelhado (RGB)
    noStroke() # Sem contorno
    # Desenha o triângulo-telhado
    triangle(x-extra, y+altura_telhado,
             x+largura/2.0, y,
             x+extra+largura, y+altura_telhado)
    fill(255, 255, 255) # Preenchimento branco
    stroke(10,75) # Tênue contorno
    # Desenha um retângulo-casa
    rectMode(CORNERS)
    rect(x, y+altura_telhado,
         x+largura, y+altura_total)
# Desenha algumas casas
desenha_casa(40,150,70,
             percentual_de_telhado=0.33+random(0.3))
desenha_casa(300,190,140,
             percentual_de_telhado=0.33+random(0.3))
```

Código 5.2: Um outro exemplo de esboço estático: desenha o céu, o chão e algumas casas.

Exercício 5.3. *Inclua no código 5.2 uma função* `desenha_nuvens`*, que recebe um número como argumento e desenha este número de núvens, em posições aleatórias do céu. Cada núvem pode ser desenhada utilizando-se elipses brancas e/ou cinzas.*

5.2 Animações e objetos – a invasão

Os esboços que fizemos até agora são estáticos. Grande parte do poder do Processing, no entanto, está em produzir conteúdo dinâmico e interativo. Para isto, utiliza-se as funções especiais `setup` e `draw`. No contexto do Processing, a primeira função determina quais comandos são executados inicialmente, quando uma nova janela é criada; já a segunda desenha nesta janela. Na verdade, a função `draw` é executada 60 vezes por segundo,[3] o que permite utilizá-la para animações. Execute o exemplo abaixo.

```
1  y = 300
2
3  def setup():
4      size(600,600)
5
6  def draw():
7      global y
8      clear()
9      rectMode(CENTER)
10     rect(300,y,30,30)
11     y += 1
12     if y>600:
13         y=0
```

Código 5.3: Animação simples.

O código 5.3 produz um quadrado branco que fica constantemente caindo (quando atinge a base, a linha 13 o manda de volta ao topo). A taxa com a qual ele cai pode ser ajustada trocando-se o valor incrementado na linha 11 (experimente). A função `clear` na linha 8 apaga o conteúdo previamente desenhado (experimente remover esta linha e executar novamente o código).

Exercício 5.4. *Acrescente ao código 5.3 um circulo que se movimenta na diagonal.*

Exercício 5.5. *Faça com que o circulo e o quadrado tenham uma probabilidade de 50% de mudar de direção a cada 30 quadros. Dica: crie uma variável contador que é incrementada a cada chamada da função* `draw` *e use o operador aritmético* `%` *.*

Exercício 5.6. *Inclua na cena a personagem do código 5.1 (e exercício 5.1). Faça com que ele se movimente como um bêbado, isto é, com que cada passo possa ser dado em qualquer direção com igual probabilidade.*

Quem seguiu estritamente o modelo do código 5.3 para resolver o exercício 5.6, deve ter notado que o código começou a ficar desorganizado devido à multiplicação de variáveis

[3] Esta taxa, que é o número de quadros desenhados por segundo, pode ser alterada utilizando-se a função `frameRate`.

globais. Poderia-se lidar com isso definindo-se uma lista ou um dicionário para armazenar as coordenadas de cada componente do desenho (que é uma excelente solução em alguns contextos) mas também é possível aplicar nossos conhecimentos do capítulo 4 para encontrar uma solução um pouco mais natural: definir classes. Veja o exemplo no código 5.4.

Assim, utilizamos a função `setup` para inicializar os objetos e acrescentá-los a uma lista. Depois, basta iterar sobre esta lista e utilizar o método `desenha` de cada objeto para acrescentá-los à tela.

Nos próximos exercícios, vamos animar a paisagem dos exercícios 5.2 e 5.3. Vamos acrescentar movimentos suaves às núvens, um morador inquieto e invasores alienígenas.

Exercício 5.7. *Crie uma função `cenario` que desenhe o chão e as casas da paisagem. Adapte o código do exercício 5.3 de maneira a incluir núvens (distribuidas aleatoriamente que se movimentam lentamente pelo céu (tome cuidado para que as núvens não cruzem o horizonte ou encobram alguma coisa).*

Exercício 5.8. *Inclua também o personagem do exercício 5.6, tomando o cuidado de restringir os movimentos dele para que não passe por cima das casas.*

Exercício 5.9. *Finalmente, inclua um (ou alguns) invasor(es) alienígena(s) na cena.*

O código dos últimos exercícios pode ter ficado um tanto longo. O Processing também também pode trabalhar com módulos personalizados (que vimos no capítulo 2). De fato, é possível trabalhar com diversas abas, cada uma contendo um módulo (que será um arquivo `.py` simples.

Exercício 5.10. *Organize um pouco o código dos exercícios 5.7 a 5.9 gravando a classe `Alien` em um módulo `alien` e opcionalmente fazendo o mesmo para as núvens e a personagem.*

```
class Alien(object):
    def __init__(self, x, y, vx, vy):
        self.x = x; self.y = y
        self.vx = vx; self.vy = vy

    def atualiza_coords(self):
        self.x += self.vx
        self.y += self.vy
        if self.x > width+30:
            self.x = 0
        elif self.x < -30:
            self.x = width
        if self.y > height+30:
            self.y = 0
        elif self.y < -30:
            self.y = height

    def desenha(self):
        self.atualiza_coords()
        stroke(250,250,250)
        rectMode(CENTER)
        fill(0, 128, 0)
        rect(self.x,self.y,20,20)
        ellipse(self.x,self.y-30,60,60)
        fill(245,245,245)
        ellipse(self.x-16,self.y-30,15,32)
        ellipse(self.x+16,self.y-30,15,32)
        fill(100,100,100,100)
        noStroke()
        ellipse(self.x,self.y-17,120,120)
        fill(30,30,10)
        ellipse(self.x,self.y+30,170,40)

objetos = []
def setup():
    size(600,600)
    for i in range(3):
        x,y = random(600), random(600)
        vx,vy = random(-5,5), random(-5,5)
        objetos.append(Alien(x,y, vx, vy))

def draw():
    clear()
    for obj in objetos:
        obj.desenha()
```

Código 5.4: Uma classe que define um alien e seu uso.

5.3 Esboços interativos

Para tornar interativo um esboço, o Processing dispõe de uma série de variáveis globais especiais que contém o estado do sistema. Um exemplo simples são as variáveis `mouseX` e `mouseY` que contém as coordenadas do mouse naquele instante.

No código 5.5 estão exemplificados diversos efeitos que podem ser obtidos simplesmente manipulando estas variáveis. O código produz uma tela inicial contendo 2 círculos. Vamos ver como se dá a interação com cada um deles.

O primeiro círculo é controlado pelo teclado: na linha 21 utilizamos a variável `keyPressed`, que possui valor `True` no caso de alguma tecla estar pressionada e `False` do contrário. Caso uma tecla tenha sido pressionada, a variável `key` recebe uma string contendo o valor da tecla apertada, a menos que a tecla seja um caractere especial: neste caso váriável `key` recebe o valor `CODED` (outra variável especial do Processing). Quando `key` é igual a `CODED`, a variável especial, `keyCode`, pode ser usada para descobrirmos qual tecla foi pressionada. Seguindo este procedimento, nas linhas 22 a 29 modificamos as coordenadas do primeiro círculo de maneira que sua posição seja controlada pelas setas do teclado.

Há um outro círculo que aparece na inicialização (linha 17) que é programado para seguir o ponteiro do mouse em qualquer instante através das variáveis `mouseX` e `mouseY` (linha 43).

Nas linhas 38 a 42 define-se o comporamente no caso do usuário utilize os botões do *mouse*. Um clique com o botão esquerdo[4] inclui um novo círculo (que, seguindo as linha 9, irá perambular ao acaso pela tela, gerando um curioso desenho) e um clique com o botão direito removerá um dos círculos (tomando o cuidado de não remover os primeiros dois círculos).

As linhas 30 a 37, contém as instruções do que fazer se uma tecla não-especial (i.e. diferente de `CODED`) for pressionada. Caso a tecla seja R, limpa-se a tela. Caso seja qualquer outra tecla, seu valor é acrescentado à string `texto` que é impressa na tela na posição $(20, 75)$.

Exercício 5.11. *Acrescente sketch do código 5.5 um retângulo de cor clara contendo texto escuro mostrando o número de objetos ativos a cada instante.*

Exercício 5.12. *Torne interativa a cena dos exercícios 5.8 e 5.9. Para isto, faça com que seja possível controlar o movimento da personagem utilizando o teclado. Além disso, inicie a cena sem invasores alienígenas, e faça com que estes sejam acresentados à cena na posição do cursor do mouse quando o botão do mouse for pressionado.*

[4] Caso o esboço que se esteja preparando seja distribuido para diversas plataformas, deve-se levar em consideração que nem todo sistema operacional suporta vários botões para o *mouse*.

```
class circulo_colorido:
    def __init__(self, x, y, tamanho=15, move=True):
        self.R,self.G,self.B = random(255),random(255),random(255)
        self.x = x; self.y = y
        self.tamanho=tamanho; self.move = move
    def desenha(self):
        fill(self.R, self.G, self.B, 100)
        ellipse(self.x, self.y, self.tamanho, self.tamanho)
        if self.move:
            self.x += random(-2,2); self.y += random(-2,2)
objs = []; texto = ''

def setup():
    size(600,600) # Inicia com dois círculos (um deles sob o cursor)
    objs.append(circulo_colorido(300,300, tamanho=22, move=False))
    objs.append(circulo_colorido(mouseX, mouseX, move=False))
    clear()
def draw():
    global texto, objs
    if keyPressed: # Detecta se o teclado foi usado
        if key==CODED: # Detecta se é um não-caractere
            if keyCode==UP: # Para cima
                objs[0].y -= 5
            elif keyCode==DOWN: # Para baixo
                objs[0].y += 5
            elif keyCode==LEFT: # Para a esquerda
                objs[0].x -= 5
            elif keyCode==RIGHT: # Para a direita
                objs[0].x += 5
        else: # Se for um caractere normal
            if key=='r': # Se a tecla R foi pressionada
                texto = '' # Limpa o texto
                clear() # Limpa a tela
            else: # Se for outra tecla
                fill(240)
                texto += key # Acrescenta o caractere
                text(texto, 20, 75) # Escreve o texto
    if mousePressed: # Se o botão do mouse for pressionado
        if mouseButton == LEFT: # Se for o botao esquerdo
            objs.append(circulo_colorido(mouseX,mouseY)) # Acrescenta círculo
        elif len(objs)>2 and mouseButton==RIGHT:
            objs.pop() # Remove um círculo se for o botão direito
    objs[1].x = mouseX; objs[1].y = mouseY # Círculo que segue o mouse

    for obj in objs:
        obj.desenha()
```

Código 5.5: Demonstração de recursos interativos. Experimente, ao rodar o código, clicar e arrastar sobre a tela e depois esperar alguns minutos. Pressione a tecla R para limpar a tela.

5.4 Alterando referenciais e plantando árvores

Existe uma maneira pouco intuitiva mas poderosa de fazer os desenhos no Processing: é possível manipular o sistema de coordenadas. Isso é feito através das funções `rotate` e `translate`. Vejamos um exemplo.

```
1  def desenha_linhas():
2      for i in range(10,height,10):
3          line(0,i,width,i)
4  size(100,100)
5  pushMatrix() # Armazena o estado inicial do sist. de coord.
6  translate(width/2.0,height/2.0)
7  rotate(radians(45))
8  stroke(0,200,0) # Verde
9  desenha_linhas()
10 popMatrix() # Retorna ao estado inicial do sist. de coord.
11 stroke(200,0,0) # Vermelho
12 desenha_linhas()
```

Ao executar notamos que, embora isto não apareça explicitamente na função `desenha_linhas`, as linhas verdes estão deslocadas e rotacionadas em relação às linhas vermelhas. Vamos ver o que está acontecendo. Após a linha 7, a coordenada $(x, y) = (0, 0)$ passa a corresponder a um ponto no centro da janela. Da mesma forma, após a linha 8, os eixos foram rotacionados valores constantes de x passam a corresponder às diagonais verdes mostradas.

As funções `pushMatrix` e `popMatrix` servem para gravar e restaurar o estado do sistema de coordenadas (permitindo que desenhemos as linhas vermelhas seguindo nossa intuição inicial). Podemos utilizar um bloco `with` para evitar a situção de esquecermos de usar a função `popMatrix` (mais ou menos como o usamos antes para evitar esquecer de fechar arquivos). Veja o exemplo no código 5.6 a seguir.

Exercício 5.13. *Incline ligeiramente (cerca de* 15°*) o chapéu da personagem do exercício 5.1.*

O Processing vem com um grande número de exemplos que ilustram diversos usos e funcionalidades. Eles podem ser acessados do menu `File` → `Examples`. Vamos nos concentrar no exemplo *Tree*, que está dentro da pasta *Fractals and L-Systems*. Este exemplo demonstra como construir o desenho de uma árvore utilizando uma função recursiva (do tipo que foi discutido na seção 3.2). A função desenha um ramo, gira o sistema de coordenadas e invoca a si mesma para desenhar mais um ramo, gira o sistema de coordenadas novamente, no sentido oposto, e invoca a si mesma para desenhar outro ramo. Desta forma, com um código extremamente compacto, é construída uma "árvore". Esta figura possui uma série de propriedades interessantes, entre elas a *auto-similaridade*: se recortarmos um pedaço da árvore, rotacionarmos e aumentarmos de tamanho ele será

```
def desenha_linhas():
    for i in range(10,height,10):
        line(0,i,width,i)
def setup():
    size(300,300)
def draw():
    clear()
    with pushMatrix(): # Armazena o estado inicial do sist. de coord.
        translate(mouseX,height/width*float(mouseX))
        rotate(radians(float(mouseY)/height*360.0))
        stroke(0,200,0) # Verde
        desenha_linhas()
    stroke(200,0,0) # Vermelho
    desenha_linhas()
```

Código 5.6: Exemplo do uso de `rotate`, `translate` e `pushMatrix`.

idêntico à árvore inteira.

Exercício 5.14. *Faça pequenos experimentos com código do exemplo, certificando-se de compreendê-lo. Inclua então no último nível (i.e. quando* `h<=2`*) folhas e/ou frutas. Dê espessura à árvore, substituindo a função* `line` *por* `rect`*, e faça com que a espessura diminua a cada nível (i.e. faça a largura dos retângulos proporcional a* `h`*).*

6 Continuando daqui...

Interfaces gráficas Os programas em Python deste curso foram escritos para serem executados em um terminal e toda a interação com eles (a menos das aplicações utilizando o Processing) foi através de texto. É relativamente simples, no entanto, simples desenhar interfaces gráficas (ou GUIs, sigla em inglês para *Guided User Interfaces*) utilizando o módulo `tkinter`, que pertence à biblioteca padrão do Python[1]. Desta forma, é possível produzir aplicações com botões, janelas e caixas de texto com as quais estamos acostumados.

Produtividade Frequentemente, utilizamos planilhas do MS Excel ou LivreOffice Calc para organizar coisas no nosso dia-a-dia. Não é incomum a situação em que precisamos extrair ou reorganizar dados dispersos em diversas planilhas preparadas anteriormente.

É possivel ler e modificar arquivos do MS Excel utilizando a biblioteca `openpyxl`[2]. Além da documentação oficial, uma explicação passo-a-passo pode ser encontrada no capítulo 12[3] do livro "Automate the Boring Stuff with Python" (ABS), de Al Sweigart, que está disponível gratuitamente online.

No caso do LibreOffice Calc, é possível não só escrever programas em Python que modicam acessem os arquivos, mas também escrever macros diretamente em Python[4] que rodam dentro do programa.

Alternativamente, pode-se querer interagir com as planilhas do Google Sheets. Isso pode ser feito utilizando a API disponibilizada pelo Google.[5]

Documentos do processador de textos MS Word também podem ser manipulados, utilizando o módulo `python-docx`.[6] Além da documentação oficial, há uma discussão detalhada sobre isso no capítulo 13 do ABS.[7]

[1] http://www.tkdocs.com/tutorial/
[2] http://openpyxl.readthedocs.io
[3] https://automatetheboringstuff.com/chapter12/
[4] https://help.libreoffice.org/Common/Scripting
[5] https://developers.google.com/sheets/quickstart/python
[6] https://python-docx.readthedocs.io/en/latest/user/quickstart.html
[7] https://automatetheboringstuff.com/chapter13/

Extraindo dados da web Há um mundo de dados dispersos em páginas web. Pequenos scripts pode ser escritos para visitar um grande número de páginas e extrair o que nos interessa.[8,9,10] Para executar este tipo de tarefa é necessário empregar módulos como o `requests`[11], que permite o conteúdo de páginas seja baixado, e `BeautifulSoup`[12], que interpreta elementos html.

Informações geográficas Diversos orgãos públicos (incluindo a prefeitura de São Paulo[13]) disponibilizam hoje dados geográficos detalhados (e.g. topografia, população, edificação, etc) em *shapefiles* que podem ser baixados e analisados através de softwares GIS (sigla em inglês para 'sistemas de informação geográfica').

O QGIS é um popular software livre de GIS que suporta *scripting* em Python diretamente na sua janela principal.[14,15] O uso de programação facilita operações mais complexas de seleção e manipulação destes dados.

Rhinoceros e Grasshopper O aplicativo de CAD Rhinoceros 3D suporta Python como linguagem de scripting nativa[16]. Esta associação entre CAD e programação é bastante

O Rhinocceros 3D também suporta uma *linguagem de programação visual* chamada Grasshopper. Embora funcione bem para muitas tarefas, há casos que são difíceis de tratar "montando" o programa com os elementos visuais do Grasshopper. Convenientemente, porém é possível misturar as duas abordagens: incluindo elementos programados em Python dentro do Grasshopper.[17]

Cenas 3D / Jogos O software gráfico livre Blender (que permite a preparação de animações, cenas 3D e até mesmo impressão 3D e jogos) suporta scripting em Python.[18]

Também é possível escrever scripts em Python para o Autodesk Maya.[19]

A game engine Panda3D que permite o desenvolvimento de jogos 3D completos utiliza Python como linguagem básica[20].

[8] http://journalistsresource.org/tip-sheets/research/python-scrape-website-data-criminal-justice
[9] http://programminghistorian.org/lessons/intro-to-beautiful-soup
[10] https://automatetheboringstuff.com/chapter11/
[11] http://www.python-requests.org/en/master/
[12] http://programminghistorian.org/lessons/intro-to-beautiful-soup
[13] http://geosampa.prefeitura.sp.gov.br
[14] http://www.qgistutorials.com/en/docs/getting_started_with_pyqgis.html
[15] http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/
[16] http://developer.rhino3d.com/guides/rhinopython/
[17] https://www.youtube.com/playlist?list=PL5Up_u-XkWgP7nB7XIevMTyBCZ7pvLBGP
[18] https://www.blender.org/api/blender_python_api_current/
[19] http://cgkit.sourceforge.net/maya_tutorials/intro/
[20] http://www.panda3d.org/manual/

Gráficos e matemática Para tarefas matemáticas mais sofisticadas com o Python, o ponto de partida é o pacote `numpy`,[21] cuja classe `numpy.ndarray` permite que se trabalhe com vetores e matrizes de maneira simples e eficiente.

A biblioteca `scipy`,[22] fornece funções para calculos numéricos variados (desde achar raízes de funções até a solução sistemas de equações diferenciais).

Finalmente, há uma poderosa biblioteca para visualização de dados chamada `matplotlib`[23]. Com ela pode-se produzir desde histogramas ou gráficos pizza até complicados diagramas tridimensionais. A maneira mais simples de utilizar a `matplotlib` é: acessar a galeria de exemplos[24], procurar o gráfico que se aproxima do resultado que se quer obter e estudar o código fonte que o produziu.

Arduino Quem se interessa por automação inevitavelmente vai se deparar com placas Arduino, que são uma maneira simples de construir protótipos e implementar ideias. É relativamente simples escrever scripts em Python para interagir com sistemas Arduino através da porta serial do computador.[25]

Usar outra linguagem de programação Aprender uma linguagem de programação nova que é necessária ou conveniente para uma dada tarefa específica pode soar intimidador, mas não precisa ser. Os conceitos básicos de programação e o modo de raciocinar são os mesmos na maioria das linguagens. De fato, geralmente, basta uma olhada em bom manual ou tutorial para se conseguir completar tarefas simples em uma outra linguagem.

Uma boa maneira de se aproximar de uma linguagem nova é começar olhando códigos escritos na linguagem que conhecemos e ver como a mesma coisa é implementada na outra que queremos aprender. Existe um interessante website dedicado a isso chamado Rosetta Code.[26]

[21] http://numpy.org/
[22] http://scipy.org/
[23] http://matplotlib.org/
[24] http://matplotlib.org/gallery.html
[25] http://playground.arduino.cc/Interfacing/Python
[26] https://www.rosettacode.org/

Lista de códigos

Índice Remissivo