O'REILLY®

Segunda Edição

Data Science do Zero

Noções Fundamentais com Python

Joel Grus

ALTA BOOKS
EDITORA

networks 2

machine 2

neural 2

scikit-learn 2

r 2

Veremos formas mais sofisticadas de extrair tópicos dos dados no Capítulo 21.

Em Frente

Foi um ótimo primeiro dia! Exausto, você sai do prédio antes de ouvir outro pedido. Durma bem, porque amanhã você participará do curso para novos funcionários. (Sim, você trabalhou um dia inteiro sem nenhuma formação inicial. Culpa do RH.)

CAPÍTULO 2
Um Curso Intensivo de Python

Mesmo depois de vinte e cinco anos, as pessoas continuam fanáticas pelo Python, o que para mim é inacreditável.
—Michael Palin

Os novos funcionários da DataSciencester têm que participar de um ciclo de formação inicial, cuja parte mais interessante é o curso intensivo de Python.

Não se trata de uma especialização abrangente em Python, mas de um curso que prioriza os aspectos da linguagem que são mais importantes para nossos propósitos (alguns deles nem costumam ser priorizados nos tutoriais de Python). Se você nunca usou o Python, é melhor recorrer também a um tutorial para iniciantes.

O Zen do Python

Os princípios de design do Python são descritos de forma um tanto quanto zen (*http://legacy.python.org/dev/peps/pep-0020/*); para acessá-los no intérprete do Python, digite "import this."

Este é um dos mais populares:

> Deve haver uma — preferivelmente, apenas uma — forma evidente de fazer algo.

Em geral, quando escrito dessa forma "evidente" (que talvez não seja tão óbvia para um novato), o código é descrito como "Pythonic". Embora este livro não seja sobre Python, vamos comparar métodos Pythonic e não Pythonic para realizar os mesmos processos, priorizando o uso de soluções Pythonic para os nossos problemas.

Vários princípios são estéticos:

> É melhor algo belo do que feio. Algo expresso do que implícito. Algo simples do que complexo.

Essas diretrizes orientarão nosso código.

Iniciando no Python

É difícil colocar as instruções de instalação mais recentes nos livros físicos, mas você pode encontrá-las atualizadas no repositório deste livro no GitHub

(*https://github.com/joelgrus/data-science-from-scratch/blob/master/INSTALL.md*).

Se as instruções indicadas aqui não funcionarem, confira o repositório.

Você pode baixar o Python em Python.org (*https://www.python.org/*). Mas, para quem ainda não tem a linguagem, recomendo a distribuição Anaconda (*https://www.anaconda.com/download/*), que contém a maioria das bibliotecas necessárias para praticar o data science.

Quando escrevi o primeiro esboço deste livro, o Python 2.7 era a versão mais popular entre os cientistas de dados, portanto, a primeira edição foi baseada nela.

No entanto, nos últimos anos, praticamente todo mundo migrou para o Python 3. Como as versões recentes do Python facilitam o código limpo, utilizaremos bastante alguns recursos que só estão disponíveis na versão 3.6 ou em versões superiores. Por isso, você deve obter o Python 3.6 ou superior. (Além disso, agora muitas bibliotecas úteis são incompatíveis com o Python 2.7, o que é mais um motivo para migrar.)

Ambientes Virtuais

A partir do próximo capítulo, usaremos a biblioteca matplotlib para gerar gráficos. Como essa biblioteca não está integrada ao Python, você terá que instalá-la. Todo projeto de ciência de dados precisa de uma combinação de bibliotecas externas e, às vezes, versões específicas diferentes das usadas anteriormente. Se você só tiver uma instalação do
Python, essas bibliotecas entrarão em conflito e causarão todo tipo de problema.

Para isso, a solução padrão são os ambientes virtuais, ambientes Python de área restrita com versões específicas de bibliotecas (ou, dependendo da configuração, até mesmo do Python).

Recomendei a instalação da distribuição do Anaconda; então, nesta seção, explicarei como os ambientes do Anaconda funcionam. Se você não quiser o Anaconda, use o módulo interno venv (*https://docs.python.org/3/library/venv.html*) ou instale o virtualenv (*https://virtualenv.pypa.io/en/latest/*). Nesse caso, siga as instruções aplicáveis.

Para criar um ambiente virtual (Anaconda), siga as seguintes etapas:

```
# crie um ambiente Python 3.6 chamado "dsfs" conda create -n dsfs python=3.6
```

Siga os avisos para criar um ambiente virtual chamado "dsfs" com as seguintes instruções:

```
#
# Para ativar esse ambiente, use:
# > source activate dsfs
#
# Para desativar um ambiente ativo, use:
# > source deactivate
#
```

Como indicado, para ativar o ambiente, use:

```
source activate dsfs
```

Neste ponto, o prompt de comando deve indicar o ambiente ativo. No MacBook, o prompt fica da

seguinte forma:

```
(dsfs) ip-10-0-0-198:~ joelg$
```

Enquanto esse ambiente estiver ativo, todas as bibliotecas serão instaladas apenas no ambiente dsfs. Depois de estudar este livro, quando desenvolver seus projetos, crie ambientes para eles.

Agora que você criou um ambiente, é uma boa ideia instalar o IPython (*http://ipython.org/*), um shell Python completo:

```
python -m pip install ipython
```

O Anaconda dispõe de um gerenciador de pacotes, o conda, mas você também pode usar o pip padrão do gerenciador de pacotes do Python, como faremos a partir de agora.

De agora em diante, prosseguiremos como se você tivesse criado e ativado um ambiente virtual no Python 3.6 (utilize o nome que quiser), e os próximos capítulos podem citar as bibliotecas cuja instalação foi recomendada nos capítulos anteriores.

Por uma questão de disciplina, sempre trabalhe usando um ambiente virtual e nunca a instalação "básica" do Python.

Formatação de Espaço em Branco

Muitas linguagens usam chaves para delimitar blocos de código, mas o Python adota o recuo:

```
# A cerquilha indica o início de um comentário. O Python
# ignora esses comentários, mas eles orientam os leitores do código.
for i in [1, 2, 3, 4, 5]:
print(i) # primeira linha do bloco "for i"
for j in [1, 2, 3, 4, 5]:
print(j) # primeira linha do "for j"
print(i + j) # última linha do bloco "for j"
print(i) # última linha do bloco "for i"
print("done looping")
```

Por isso, o código Python é bem legível, mas você tem que ser muito cuidadoso com a formatação.

Os programadores costumam debater sobre o uso de tabulação ou espaço para o recuo. Em muitas linguagens, isso não é tão importante; no entanto, o Python lê

tabulações e espaços como recuos diferentes e não executará o código se você misturar os dois. No Python, use sempre espaço, nunca tabulação. (Se você utiliza um editor, configure a tecla Tab para inserir apenas espaços.)

O espaço em branco é ignorado quando aparece dentro de parênteses e colchetes, algo muito útil para lidar com computações intermináveis:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

Para facilitar a leitura do código:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
easier_to_read_list_of_lists = [[1, 2, 3],
[4, 5, 6],
[7, 8, 9]]
```

Você também pode usar uma barra invertida para indicar que a instrução continua na próxima linha, mas isso raramente ocorre:

```
two_plus_three = 2 + \
3
```

A formatação de espaço em branco pode dificultar as ações de copiar e colar o código no shell do Python. Por exemplo, se você tentasse colar o seguinte código no shell comum do Python:

```
for i in [1, 2, 3, 4, 5]:
# Observe a linha em branco
    print(i)
```

Apareceria a seguinte reclamação:

```
IndentationError: expected an indented block
```

Isso ocorre porque o interpretador acha que a linha em branco indica o final do bloco do loop for.

O IPython tem a função mágica %paste, que copia corretamente o conteúdo da área de transferência, com os espaços em branco e tudo mais. Só isso já é um excelente motivo para usar o IPython.

Módulos

Alguns recursos do Python não são carregados por padrão, como certos componentes integrados à linguagem e elementos externos, disponíveis para download. Para usar esses recursos, você terá que import (importar) seus respectivos módulos.

Uma opção é importar o módulo em questão:

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

Aqui, re é o módulo que contém as funções e constantes aplicáveis às expressões regulares. Após esse tipo de import, para acessar as respectivas funções, você deve usar o prefixo re..

Se já houver outro re no código, você pode usar um alias:

```
import re as regex
```

```
my_regex = regex.compile("[0-9]+", regex.I)
```

Você pode fazer isso se o módulo tiver um nome complicado ou se precisar digitar um trecho muito longo. Por exemplo, para visualizar dados com o matplotlib, existe um padrão:

```
import matplotlib.pyplot as plt

plt.plot(...)
```

Para obter valores específicos de um módulo, você pode importá-los expressamente e usá-los sem qualificação:

```
from collections import defaultdict, Counter

lookup = defaultdict(int)
my_counter = Counter()
```

Para fazer uma bagunça, você pode importar o conteúdo inteiro de um módulo para o namespace, substituindo (sem querer) as variáveis definidas anteriormente:

```
match = 10

from re import * # opa, o re tem uma função match print(match) # "<function match at 0x10281e6a8>"
```

Mas, como você não é bagunceiro, nunca faça isso.

Funções

Cada função é uma regra que recebe zero ou mais entradas e retorna a saída correspondente. No Python, definimos as funções usando def:

```
def double(x): """
Nesse ponto, você coloca um docstring opcional para descrever a
função. Por exemplo, esta função multiplica a entrada por 2.
"""

return x * 2
```

As funções de Python são de primeira classe, ou seja, podemos atribuí-las a variáveis e inseri-las nas funções como argumentos:

```
def apply_to_one(f):

"""Chama a função f usando 1 como argumento""" return f(1)

my_double = double # refere-se à função x já definida

x = apply_to_one(my_double) # igual a 2
```

Também é fácil criar pequenas funções anônimas, as lambdas:

```
y = apply_to_one(lambda x: x + 4) # igual a 5
```

Você pode atribuir lambdas a variáveis, embora quase todo mundo recomende o def:

```
another_double = lambda x: 2 * x # não faça isso

def another_double(x): """Faça isso"""
return 2 * x
```

Os parâmetros da função também podem receber argumentos padrão, que devem ser especificados se você quiser obter um valor diferente do padrão:

```
def my_print(message = "my default message"): print(message)

my_print("hello") # imprime 'hello'
```

```
my_print() # imprime 'my default message'
```

Às vezes, é útil especificar argumentos pelo nome:

```
def full_name(first = "What's-his-name", last = "Something"): return first + " " + last
full_name("Joel", "Grus") # "Joel Grus"
full_name("Joel") # "Joel Something"
full_name(last="Grus") # "What's-his-name Grus"
```

Criaremos muitas funções.

Strings (Cadeias de Caracteres)

As strings podem ser delimitadas por aspas simples ou duplas (mas sempre combinando):

```
single_quoted_string = 'data science' double_quoted_string = "data science"
```

No Python, a barra invertida serve para codificar caracteres especiais. Por exemplo:

```
tab_string = "\t" # representa o caractere tab len(tab_string) # é 1
```

Para usar o caractere da barra invertida (como vemos nos nomes dos diretórios e nas expressões regulares do Windows), você pode criar strings brutas com r"":

```
not_tab_string = r"\t" # representa os caracteres '\' e 't' len(not_tab_string) # é 2
```

Para criar strings de várias linhas, use três aspas duplas:

```
multi_line_string = """Esta é a primeira linha.
e esta é a segunda linha
e esta é a terceira linha"""
```

No Python 3.6, há um novo recurso: a f-string, uma forma simples de substituir os valores nas strings. Por exemplo, quando o nome e o sobrenome são indicados separadamente:

```
first_name = "Joel" last_name = "Grus"
```

Nesse caso, queremos formar o nome completo com eles. Há várias formas de construir uma string full_name:

```
full_name1 = first_name + " " + last_name # adição de strings
full_name2 = "{0} {1}".format(first_name, last_name) # string.format
```

Mas usar a f-string é bem menos complicado:

```
full_name3 = f"{first_name} {last_name}"
```

Utilizaremos esse método ao longo do livro.

Exceções

Quando algo dá errado, o Python gera uma exceção. Se não forem tratadas, as exceções causarão falhas no programa. Para tratá-las, você pode usar try e except:

```
try:
print(0 / 0)
except ZeroDivisionError: print("cannot divide by zero")
```

Apesar de terem uma má reputação em muitas linguagens, no Python, as exceções são utilizadas

sem grilo para deixar o código mais limpo; de vez em quando, faremos isso.

Listas

Provavelmente, a estrutura de dados mais fundamental do Python é a lista. Uma lista é apenas uma coleção ordenada (parecida com o array das outras linguagens, mas com funcionalidades adicionais):

```
integer_list = [1, 2, 3]
heterogeneous_list = [“string”, 0.1, True]

list_of_lists = [integer_list, heterogeneous_list, []]

list_length = len(integer_list) # igual a 3

list_sum = sum(integer_list) # igual a 6
```

Você pode obter e definir o elemento de número n de uma lista usando colchetes:

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

zero = x[0] # igual a 0, as listas são indexadas a partir de 0

one = x[1] # igual a 1

nine = x[-1] # igual a 9, ‘Pythonic’ para o último elemento

eight = x[-2] # igual a 8, ‘Pythonic’ para o penúltimo elemento

x[0] = -1 # agora x é [-1, 1, 2, 3, ..., 9]
```

Você também pode usar os colchetes para fatiar as listas. A fatia i:j contém todos os elementos de i (incluído) a j (não incluído). Se o início da fatia não for indicado, ela começará no início da lista; se o final da fatia não for indicado, ela terminará no final da lista:

```
first_three = x[:3] # [-1, 1, 2]

three_to_end = x[3:] # [3, 4, ..., 9]

one_to_four = x[1:5] # [1, 2, 3, 4]

last_three = x[-3:] # [7, 8, 9]

without_first_and_last = x[1:-1] # [1, 2, ..., 8]

copy_of_x = x[:] # [-1, 1, 2, ..., 9]
```

Do mesmo modo, você pode fatiar strings e outros tipos de “sequências”.

A fatia pode receber um terceiro argumento para indicar seu stride, que pode ser negativo:

```
every_third = x[::3] # [-1, 3, 6, 9]

five_to_three = x[5:2:-1] # [5, 4, 3]
```

O Python dispõe de um operador in para verificar a associação à lista:

```
1 in [1, 2, 3] # Verdadeiro

0 in [1, 2, 3] # Falso
```

Como essa verificação analisa todos os elementos da lista, você só deve executá-la se a lista for bem pequena (ou se o tempo da verificação não for importante).

É fácil concatenar listas. Para modificar a lista, você pode usar extend e adicionar itens de outra coleção:

```
x = [1, 2, 3]
x.extend([4, 5, 6]) # x agora é [1, 2, 3, 4, 5, 6]
```

Se não quiser modificar x, você pode usar a adição de listas:

```
x = [1, 2, 3]
y = x + [4, 5, 6] # y é [1, 2, 3, 4, 5, 6]; x não mudou
```

Na maioria das vezes, acrescentaremos item por item às listas:

```
x = [1, 2, 3]
x.append(0) # x agora é [1, 2, 3, 0]
y = x[-1] # igual a 0
z = len(x) # igual a 4
```

Muitas vezes, é conveniente descompactar as listas quando sabemos quantos elementos elas contêm:

```
x, y = [1, 2] # agora x é 1, y é 2
```

No entanto, aparecerá um ValueError se não houver o mesmo número de elementos nos dois lados.

Geralmente, usamos um sublinhado para indicar o valor que será descartado:

```
_, y = [1, 2] # agora y == 2, não considerou o primeiro elemento
```

Tuplas

As tuplas são muito parecidas com as listas, mas não podem ser modificadas. Com uma tupla, podemos fazer quase tudo que se pode fazer com uma lista, menos modificá-la. Para especificar uma tupla, use parênteses (ou nada) em vez de colchetes:

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3 # my_list agora é [1, 3]
try:
my_tuple[1] = 3 except TypeError:
print("cannot modify a tuple")
```

As tuplas são uma forma eficaz de usar funções para retornar múltiplos valores:

```
def sum_and_product(x, y): return (x + y), (x * y)
sp = sum_and_product(2, 3) # sp é (5, 6)
s, p = sum_and_product(5, 10) # s é 15, p é 50
```

As tuplas (e as listas) também podem ser usadas em atribuições múltiplas:

```
x, y = 1, 2 # agora x é 1, y é 2
x, y = y, x # forma Pythonic de trocar variáveis; agora x é 2, y é 1
```

Dicionários

Outra estrutura fundamental é o dicionário, que associa valores a chaves e permite a rápida recuperação do valor correspondente a uma determinada chave:

```
empty_dict = {} # Pythonic
empty_dict2 = dict() # menos Pythonic
grades = {"Joel": 80, "Tim": 95} # dicionário literal
```

Para pesquisar o valor de uma chave, você pode usar os colchetes:

```
joels_grade = grades["Joel"] # igual a 80
```

Mas aparecerá um KeyError se você procurar uma chave que não está no dicionário:

```
try:
kates_grade = grades["Kate"] except KeyError:
print("no grade for Kate!")
```

Para verificar a existência de uma chave, você pode usar o in:

```
joel_has_grade = "Joel" in grades # Verdadeiro
kate_has_grade = "Kate" in grades # Falso
```

Essa verificação de associação é rápida até em dicionários grandes.

Nos dicionários, o método get retorna um valor padrão (em vez de gerar uma exceção) quando você procura por uma chave que não está no dicionário:

```
joels_grade = grades.get("Joel", 0) # igual a 80
kates_grade = grades.get("Kate", 0) # igual a 0
no_ones_grade = grades.get("No One") # o padrão é None
```

Você também pode atribuir pares de valor-chave usando os colchetes:

```
grades["Tim"] = 99 # substitui o valor anterior
grades["Kate"] = 100 # adiciona uma terceira entrada
num_students = len(grades) # igual a 3
```

Como vimos no Capítulo 1, os dicionários podem representar dados estruturados:

```
tweet = {
"user" : "joelgrus",
"text" : "Data Science is Awesome", "retweet_count" : 100,
"hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

Mas logo aprenderemos uma abordagem melhor do que essa.

Além de procurar por chaves específicas, podemos conferir todas elas:

```
tweet_keys = tweet.keys() # iterável para as chaves
tweet_values = tweet.values() # iterável para os valores

tweet_items = tweet.items() # iterável para as tuplas (chave, valor)

"user" in tweet_keys # Verdadeiro, mas não é Pythonic

"user" in tweet # forma Pythonic de verificar as chaves
"joelgrus" in tweet_values # Verdadeiro (lento, mas é a única forma de verificar)
```

As chaves do dicionário devem ser “hashable” [com função hash, imutáveis]; logo, as listas não podem ser usadas como chaves. Se você precisar de uma chave com várias partes, use uma tupla ou transforme a chave em uma string.

defaultdict

Imagine que você deseja contar as palavras de um documento. Uma abordagem óbvia seria criar um dicionário com chaves para palavras e valores para a contagem. Ao verificar cada palavra, você pode incrementar a contagem (se ela já estiver no dicionário) ou adicioná-la ao dicionário (se ela ainda não estiver nele):

```
word_counts = {}
for word in document:
if word in word_counts: word_counts[word] += 1
else:
word_counts[word] = 1
```

Você também pode usar o método “é melhor pedir perdão do que permissão”, tratando a exceção gerada na pesquisa pela chave ausente:

```
word_counts = {}
for word in document: try:
word_counts[word] += 1 except KeyError:
word_counts[word] = 1
```

Uma terceira abordagem é usar o get, que lida de forma muito interessante com chaves ausentes:

```
word_counts = {}
for word in document:
previous_count = word_counts.get(word, 0) word_counts[word] = previous_count + 1
```

Mas, como esses métodos são um tanto quanto complicados, o defaultdict é um bom recurso. Embora pareça um dicionário comum, quando procuramos uma chave que não está contida nele, ele adiciona um valor para ela usando a função de argumento zero que indicamos ao criá-lo. Para usar os defaultdicts, você deve importá-los das collections:

```
from collections import defaultdict
word_counts = defaultdict(int) # int() produz 0 for word in document:
word_counts[word] += 1
```

Esses recursos também são úteis com list, dict e outras funções:

```
dd_list = defaultdict(list) # list() produz uma lista vazia dd_list[2].append(1) # agora dd_list contém {2: [1]}
dd_dict = defaultdict(dict) # dict() produz um dict vazio dd_dict[“Joel”][“City”] = “Seattle” # {“Joel” : {“City”: Seattle”}}
dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1 # agora dd_pair contém {2: [0, 1]}
```

Isso será útil quando usarmos dicionários para “coletar” os resultados de alguma chave sem verificar se ela existe a cada operação.

Contadores

O Counter (contador) converte uma sequência de valores em algo parecido com o objeto defaultdict(int) mapeando as chaves correspondentes às contagens:

```
from collections import Counter
c = Counter([0, 1, 2, 0]) # c é (basicamente) {0: 2, 1: 1, 2: 1}
```

Essa é uma forma bem simples de resolver o problema do word_counts:

```
# lembre-se, o documento é uma lista de palavras
word_counts = Counter(document)
```

Uma instância Counter contém um método most_common bastante útil:

```
# imprima as 10 palavras mais comuns e suas contagens
for word, count in word_counts.most_common(10):
print(word, count)
```

Conjuntos

Outra estrutura de dados útil é o set (conjunto), uma coleção de elementos distintos. Para definir um conjunto, você pode listar seus elementos entre chaves:

```
primes_below_10 = {2, 3, 5, 7}
```

No entanto, isso não funciona com conjuntos vazios, pois {} significa "dict vazio". Nesse caso, você deve usar o set():

```
s = set()
s.add(1) # s agora é {1}
s.add(2) # s agora é {1, 2}
s.add(2) # s ainda é {1, 2} x = len(s) # igual a 2
y = 2 in s # igual a Verdadeiro
z = 3 in s # igual a Falso
```

Usaremos os conjuntos por dois motivos importantes. Primeiro, o in é uma operação muito rápida em conjuntos. Para aplicar um teste de associação em uma grande coleção de itens, é melhor usar um conjunto do que uma lista:

```
stopwords_list = ["a", "an", "at"] + hundreds_of_other_words + ["yet", "you"]
"zip" in stopwords_list # Falso, mas verifica todos os elementos
stopwords_set = set(stopwords_list)
"zip" in stopwords_set # verificação muito rápida
```

Segundo, encontraremos itens distintos em uma coleção:

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list) # 6
item_set = set(item_list) # {1, 2, 3} num_distinct_items = len(item_set) # 3
distinct_item_list = list(item_set) # [1, 2, 3]
```

Usaremos os conjuntos com menos frequência do que os dicionários e listas.

Fluxo de Controle

Como na maioria das linguagens de programação, você pode executar uma ação condicional usando if:

```
if 1 > 2:
message = "if only 1 were greater than two..." elif 1 > 3:
message = "elif stands for 'else if'" else:
message = "when all else fails use else (if you want to)"
```

Você também pode escrever um ternário do tipo if-then-else [se-então-senão] em uma linha (de vez em quando, faremos isso):

```
parity = "even" if x % 2 == 0 else "odd"
```

O Python tem um loop while:

```
x = 0
while x < 10:
print(f"{x} is less than 10") x += 1
```

Mas usaremos mais for e in:

```
# range(10) corresponde aos números 0, 1, ..., 9
for x in range(10):
print(f"{x} is less than 10")
```

Para obter uma lógica mais complexa, você pode usar continue e break:

```
for x in range(10): if x == 3:
continue # vá imediatamente para a próxima iteração
if x == 5:
break # saia do loop totalmente
print(x)
```

Essa operação imprimirá 0, 1, 2 e 4.

Veracidade

No Python, os Booleanos funcionam como na maioria das linguagens, mas começam com letras maiúsculas:

```
one_is_less_than_two = 1 < 2 # igual a True
true_equals_false = True == False # igual a False
```

No Python, o valor None indica um valor não existente, como o null das outras linguagens:

```
x = None
assert x == None, "esta não é uma forma Pythonic de verificar o None" assert x is None, "esta é a forma Pythonic de verificar o None"
```

No Python, você pode inserir qualquer valor que indique um booleano. Todos os exemplos a seguir são "falsy" [falsos]:

- False
- None

- [] (uma list vazia)
- {} (um dict vazio)
- “”
- set()
- 0
- 0.0

Quase todos os outros valores são tratados como True. Por isso, você pode usar instruções if para procurar listas vazias, strings vazias, dicionários vazios e assim por diante. Entretanto esse recurso pode causar bugs complicados quando seu comportamento não é esperado:

```
s = some_function_that_returns_a_string() if s:
first_char = s[0] else:
first_char = “”
```

Esta é uma forma mais breve (e talvez mais complexa) de fazer a mesma coisa:

```
first_char = s and s[0]
```

O and retorna um segundo valor quando o primeiro é “truthy” [verdadeiro] e o primeiro quando ele não é. Por isso, x deve ser um número ou None:

```
safe_x = x or 0
```

Com certeza, é um número:

```
safe_x = x if x is not None else 0
```

Possivelmente, é mais legível.

O Python tem uma função all, que recebe um iterável e retorna True quando todos os elementos são verdadeiros, e uma função any, que retorna True quando há, pelo menos, um elemento verdadeiro:

```
all([True, 1, {3}]) # True, todos são verdadeiros
all([True, 1, {}]) # False, {} é falso
any([True, 1, {}]) # True, True é verdadeiro
all([]) # True, não há nenhum elemento falso na lista
any([]) # False, não há nenhum elemento verdadeiro na lista
```

Classificação

No Python, toda lista tem um método sort que a organiza e, para não bagunçá-la, você pode usar a função sorted, que retorna uma nova lista:

```
x = [4, 1, 2, 3]
y = sorted(x) # y é [1, 2, 3, 4], x não mudou
x.sort() # agora x é [1, 2, 3, 4]
```

Por padrão, o sort e a sorted organizam a lista do menor elemento para o maior com base em uma modesta comparação entre eles.

Para organizar os elementos do maior para o menor, você pode especificar o parâmetro reverse=True. E, para comparar os resultados de uma função especificada (em vez de avaliar os elementos), use key:

```
# classifique a lista por valor absoluto do maior para o menor
x = sorted([-4, 1, -2, 3], key=abs, reverse=True) # é [-4, 3, -2, 1]
# classifique as palavras e contagens do maior número para o menor
wc = sorted(word_counts.items(),
key=lambda word_and_count: word_and_count[1], reverse=True)
```

Compreensões de Listas

Muitas vezes, para transformar uma lista em outra, você deve selecionar alguns elementos, transformá-los ou fazer as duas coisas. As compreensões de listas são a forma Pythonic de fazer isso:

```
even_numbers = [x for x in range(5) if x % 2 == 0] # [0, 2, 4]
squares = [x * x for x in range(5)] # [0, 1, 4, 9, 16] even_squares = [x * x for x in even_numbers] # [0, 4, 16]
```

Da mesma forma, você pode transformar listas em dicionários ou conjuntos:

```
square_dict = {x: x * x for x in range(5)} # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16} square_set = {x * x for x in [1, -1]} # {1}
```

Quando não precisamos do valor da lista, geralmente usamos um sublinhado como variável:

```
zeros = [0 for _ in even_numbers] # tem o mesmo tamanho de even_numbers
```

Uma compreensão de lista pode conter múltiplos fors:

```
pairs = [(x, y)
for x in range(10)
for y in range(10)] # 100 pares (0,0) (0,1) ... (9,8), (9,9)
```

Os fors posteriores podem usar os resultados dos anteriores:

```
increasing_pairs = [(x, y) # só pares com x < y,
for x in range(10) # range(lo, hi) é igual a
for y in range(x + 1, 10)] # [lo, lo + 1, ..., hi - 1]
```

Usaremos muito essas compreensões de listas.

Testes Automatizados e asserção

Os cientistas de dados escrevem muito código, mas como podemos conferir se o código está correto? Uma opção são os tipos (que veremos logo mais); mas há também os testes automatizados.

Existem frameworks sofisticados para escrever e executar testes, mas aqui usaremos apenas as instruções assert [asserção] para que o código gere um AssertionError caso a condição especificada não seja verdadeira:

```
assert 1 + 1 == 2
assert 1 + 1 == 2, "1 + 1 should equal 2 but didn't"
```

Como vemos no segundo caso, você pode adicionar uma mensagem para ser impressa em caso de falha na asserção.

Não é muito interessante declarar que 1 + 1 = 2. É bem melhor declarar que as funções estão funcionando como esperado:

```
def smallest_item(xs): return min(xs)
assert smallest_item([10, 20, 5, 40]) == 5
assert smallest_item([1, 0, -1, 2]) == -1
```

Ao longo deste livro, usaremos o assert dessa forma. É uma boa prática; recomendo que você utilize bastante essa instrução no código. (Se você conferir o código do livro disponível no GitHub, verá que ele contém muitas instruções assert; bem mais do que na versão física do livro. Assim, posso garantir que o código que escrevi está correto.)

Outro uso, menos comum, são as asserções sobre as entradas das funções:

```
def smallest_item(xs):
assert xs, “empty list has no smallest item” return min(xs)
```

De vez em quando, faremos isso, mas geralmente usaremos o assert para verificar se o código está correto.

Programação Orientada a Objetos

Como em muitas linguagens, no Python, você pode definir classes para encapsular dados e suas respectivas funções. De vez em quando, usaremos esse recurso para deixar o código mais limpo e simples. Agora, talvez seja mais fácil explicar as classes com um exemplo cheio de comentários.

Criaremos uma classe para representar um “contador manual numérico”, como os que usamos para contar os participantes da nossa conferência sobre “tópicos avançados em ciência de dados”.

A classe contém um count, incrementa a contagem ao ser clicked (clicada), permite a read_count (leitura da contagem) e pode ser reset, voltando para zero. (Na vida real, essas máquinas vão de 9999 para 0000, mas não vamos mexer nisso agora.)

Para definir uma classe, use a palavra-chave class e um nome no padrão PascalCase:

```
class CountingClicker:
“””A classe pode/deve ter um docstring, como as funções”””
```

Uma classe contém zero ou mais funções de membro. Por convenção, cada função recebe um primeiro parâmetro, self, que se refere à instância da classe específica.

Em geral, a classe tem um construtor chamado init , que recebe todos os parâmetros necessários para construir uma instância da classe e implementa as configurações:

```
definit(self, count = 0):
self.count = count
```

Embora o construtor tenha um nome engraçado, criamos as instâncias do contador usando apenas o nome da classe:

```
clicker1 = CountingClicker() # inicializado em 0 clicker2 = CountingClicker(100) # começa em count=100
clicker3 = CountingClicker(count=100) # forma mais expressa de fazer a mesma coisa
```

Observe que o nome do método init inicia e termina com duplos sublinhados. Esses métodos “mágicos” às vezes são chamados de métodos “dunder” (de double-UNDERscore, entendeu?) e

representam comportamentos “especiais”.

Os métodos de classe cujos nomes começam com um sublinhado são considerados — por convenção — “privados”; os usuários da classe não devem chamá-los diretamente. No entanto, o Python não impede os usuários de chamarem esses métodos.

Outro método como esse é o repr , que produz a representação de string de uma instância de classe:

```
def

(self):

return f“CountingClicker(count={self.count})”
```

E, finalmente, precisamos implementar a API pública da classe:

```
def click(self, num_times = 1):

“““Clique no contador algumas vezes.”””
self.count += num_times

def read(self):
return self.count

def reset(self): self.count = 0
```

Depois de defini-lo, usaremos o assert para escrever casos de teste para o contador:

```
clicker = CountingClicker()

assert clicker.read() == 0, “clicker should start with count 0”
clicker.click()

clicker.click()

assert clicker.read() == 2, “after two clicks, clicker should have count 2”
clicker.reset()

assert clicker.read() == 0, “after reset, clicker should be back to 0”
```

Com esses testes, confirmamos que o código está funcionando da forma como foi projetado e que ele continua assim depois de eventuais alterações.

De vez em quando, também criaremos subclasses, que herdam algumas funcionalidades de uma classe pai. Por exemplo, podemos criar um contador sem a opção de redefinição usando o Counting Clicker como a classe base e definindo o método reset para que ele não faça nada:

```
# A subclasse herda todo o comportamento da classe pai. class NoResetClicker(CountingClicker):

# Esta classe tem os mesmos métodos da CountingClicker

# Mas seu método reset não faz nada.

def reset(self):

pass

clicker2 = NoResetClicker() assert clicker2.read() == 0 clicker2.click()
```

```
assert clicker2.read() == 1
clicker2.reset()

assert clicker2.read() == 1, “reset shouldn’t do anything”
```

Iteráveis e Geradores

Uma característica legal da lista é a possibilidade de recuperar elementos específicos pelos seus índices, no entanto isso nem sempre é necessário! Uma lista com um bilhão de números ocupa memória demais. Se você quer obter um elemento por vez, não precisa recorrer a esse recurso. Se você só quer os primeiros elementos, gerar um bilhão deles é uma grande perda de tempo.

Em geral, só precisamos iterar na coleção usando for e in. Nesse caso, podemos criar geradores; eles podem ser iterados como listas, mas são lentos e geram apenas os valores solicitados.

É possível criar geradores usando funções e o operador yield:

```
def generate_range(n): i = 0

while i < n:

yield i # cada chamada para yield produz um valor do gerador

i += 1
```

O loop a seguir consumirá cada um dos valores gerados pelo yield até que não sobre nenhum:

```
for i in generate_range(10): print(f“i: {i}”)
```

(Na verdade, como o range também é lento, não é necessário fazer isso.)

Com um gerador, você pode até criar uma sequência infinita:

```
def natural_numbers():
“““returns 1, 2, 3, ...”””

n = 1

while True:

yield n n += 1
```

Entretanto você não deve iterar isso sem algum tipo de lógica break.

> Por outro lado, nessa abordagem lenta, você só pode iterar uma vez no gerador. Se for necessário iterar várias vezes, você terá que recriar o gerador a cada vez ou usar uma lista. Se a geração dos valores custar muito caro, talvez seja uma boa ideia usar uma lista.

Também é possível criar geradores colocando compreensões de for entre parênteses:

```
evens_below_20 = (i for i in generate_range(20) if i % 2 == 0)
```

Essa “compreensão de gerador” não faz nada até você promover a iteração (usando for ou next).

Isso pode ser aplicado na construção de pipelines sofisticados de processamento de dados:

```
# Nenhuma dessas computações *faz* nada até a iteração

data = natural_numbers()

evens = (x for x in data if x % 2 == 0) even_squares = (x ** 2 for x in evens)

even_squares_ending_in_six = (x for x in even_squares if x % 10 == 6) # e assim por diante
```

Em geral, quando iteramos em uma lista ou um gerador, não queremos obter apenas os valores, mas também seus índices. Nesse caso, o Python tem a função enumerate, que transforma os valores em pares (index, value):

```
names = [“Alice”, “Bob”, “Charlie”, “Debbie”]

# não é Pythonic

for i in range(len(names)):
print(f”name {i} is {names[i]}”)

# também não é Pythonic

i = 0

for name in names:

print(f”name {i} is {names[i]}”) i += 1

# Pythonic

for i, name in enumerate(names): print(f”name {i} is {name}”)
```

Usaremos bastante esse recurso.

Aleatoriedade

Ao longo do nosso curso de data science, teremos que gerar números aleatórios de vez em quando; isso pode ser feito com o módulo random:

```
import random

random.seed(10) # assim, obteremos sempre os mesmos resultados

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.5714025946899135, # random.random() produz números

# 0.4288890546751146,

# uniformemente entre 0 e 1.

#0.5780913011344704,

# É a função random que usaremos

#0.20609823213950174]

# com mais frequência.
```

O módulo random produz números pseudoaleatórios (ou seja, determinísticos) com base em um estado interno que você pode definir com random.seed para obter resultados reproduzíveis:

```
random.seed(10) # define a semente em 10

print(random.random()) # 0.57140259469

random.seed(10) # redefine a semente em 10

print(random.random()) # 0.57140259469 novamente
```

Às vezes, usaremos o random.randrange, que recebe um ou dois argumentos e retorna um

elemento escolhido aleatoriamente no range correspondente:

```
random.randrange(10) # seleciona aleatoriamente no range(10) = [0, 1, ..., 9]
random.randrange(3, 6) # seleciona aleatoriamente no range(3, 6) = [3, 4, 5]
```

Outros métodos são mais convenientes em alguns casos. Por exemplo, o random.shuffle reordena aleatoriamente os elementos de uma lista:

```
up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(up_to_ten) print(up_to_ten)
# [7, 2, 6, 8, 9, 4, 10, 1, 3, 5] (seus resultados provavelmente serão diferentes)
```

Para escolher aleatoriamente um elemento de uma lista, você pode usar o random.choice:

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"]) # "Bob" para mim
```

E, para escolher aleatoriamente uma amostra de elementos sem substituição (especialmente, sem repetição), você pode usar o random.sample:

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6) # [16, 36, 10, 6, 25, 9]
```

Para escolher uma amostra de elementos com substituição (especialmente, com repetição), você pode fazer múltiplas chamadas para o random.choice:

```
four_with_replacement = [random.choice(range(10)) for _ in range(4)] print(four_with_replacement) # [9, 4, 4, 2]
```

Expressões Regulares

As expressões regulares são uma forma de procurar texto. Elas são incrivelmente úteis, mas um pouco complicadas, tanto que há muitos livros sobre esse tema. Explicaremos melhor essas expressões mais adiante; até lá, confira estes exemplos de como usá-las no Python:

```
import re
re_examples = [ # Todos são True, porque
not re.match("a", "cat"), # 'cat' não começa com 'a'
re.search("a", "cat"), # 'cat' contém um 'a'
not re.search("c", "dog"), # 'dog' não contém um 'c'.
3 == len(re.split("[ab]", "carbs")), # Divide em a ou b para ['c','r','s'].
"R-D-" == re.sub("[0-9]", "-", "R2D2") # Substitui dígitos por traços.
]
assert all(re_examples), "all the regex examples should be True"
```

É importante destacar que o re.match verifica se o início de uma string corresponde a uma expressão regular, enquanto o re.search verifica se alguma parte de uma string corresponde a uma expressão regular. Em algum momento, você confundirá esses dois recursos e terá dor de cabeça com isso.

Há mais detalhes na documentação oficial (*https://docs.python.org/3/library/re.html*).

Programação Funcional

Aqui, a primeira edição deste livro introduzia as funções par tial, map, reduce e filter. Porém, após um intenso momento de iluminação, concluí que é melhor evitar essas funções; neste livro, elas foram substituídas por compreensões de listas, loops de for e outras construções mais Pythonic.

zip e Descompactação de Argumento

Muitas vezes, teremos que zipar (compactar) duas ou mais listas. A função zip transforma vários iteráveis em um só iterável de tuplas da função correspondente:

```
list1 = ['a', 'b', 'c'] list2 = [1, 2, 3]

# como o zip é lento, você tem que fazer algo parecido com isto

[pair for pair in zip(list1, list2)] # é [('a', 1), ('b', 2), ('c', 3)]
```

Quando as listas têm tamanhos diferentes, o zip para ao final da primeira.

Você também pode "extrair" uma lista usando um truque meio incomum:

```
pairs = [('a', 1), ('b', 2), ('c', 3)]

letters, numbers = zip(*pairs)
```

O asterisco (*) executa a descompactação de argumento, que usa os elementos de pairs como argumentos individuais para o zip. É o mesmo que chamar:

```
letters, numbers = zip(('a', 1), ('b', 2), ('c', 3))
```

Você pode usar a descompactação de argumento com qualquer função:

```
def add(a, b): return a + b

add(1, 2) # retorna 3 try:

add([1, 2]) except TypeError:

print("add expects two inputs") add(*[1, 2]) # retorna 3
```

É raro usar isso, mas, quando ocorre, é um truque engenhoso.

args e kwargs

Imagine que queremos criar uma função de alta ordem que receba como entrada uma função f e retorne uma nova função, que retornará duas vezes o valor de f para qualquer entrada:

```
def doubler(f):

# Aqui, definimos uma nova função que mantém uma referência a f

def g(x):

return 2 * f(x)

# E retorna a nova função

return g
```

Isso funciona em alguns casos:

```
def f1(x):
return x + 1
g = doubler(f1)
assert g(3) == 8, "(3 + 1) * 2 should equal 8"
assert g(-1) == 0, "(-1 + 1) * 2 should equal 0"
```

No entanto, não funciona com funções que recebem mais de um argumento:

```
def f2(x, y):
return x + y
g = doubler(f2) try:
g(1, 2)
except TypeError:
print("as defined, g only takes one argument")
```

Precisamos especificar uma função que receba argumentos arbitrários. Podemos fazer isso usando a descompactação de argumento e um pouco de mágica:

```
def magic(*args, **kwargs):
print("unnamed args:", args) print("keyword args:", kwargs)
magic(1, 2, key="word", key2="word2")
# imprime
# sem nome args: (1, 2)
# palavra-chave args: {'key': 'word', 'key2': 'word2'}
```

Ou seja, quando definimos uma função como essa, args é uma tupla dos seus argumentos sem nome e kwargs é um dict dos seus argumentos nomeados. Isso também funciona no sentido contrário, quando você quer usar uma list (ou tuple) e um dict para fornecer argumentos a uma função:

```
def other_way_magic(x, y, z): return x + y + z
x_y_list = [1, 2] z_dict = {"z": 3}
assert other_way_magic(*x_y_list, **z_dict) == 6, "1 + 2 + 3 should be 6"
```

Esse recurso pode ser aplicado em um monte de truques esquisitos, mas só vamos utilizá-lo para produzir funções de alta ordem com entradas que aceitem argumentos arbitrários:

```
def doubler_correct(f):
"""funciona para qualquer entrada recebida por f""" def g(*args, **kwargs):
"""todo argumento fornecido para g deve ser transmitido para f"""
return 2 * f(*args, **kwargs)
return g
g = doubler_correct(f2)
assert g(1, 2) == 6, "doubler should work now"
```

Em regra, o código ficará mais correto e legível se você indicar expressamente os argumentos recebidos pelas funções; portanto, só usaremos args e kwargs quando não houver outra opção.

Anotações de Tipo

O Python é uma linguagem tipada dinamicamente. Ou seja, ele geralmente não se importa com os tipos dos objetos se eles forem utilizados de forma válida:

```
def add(a, b): return a + b

assert add(10, 5) == 15, "+ is valid for numbers" assert add([1, 2], [3]) == [1, 2, 3], "+ is valid for lists" assert add("hi ", "there") == "hi there", "+ is valid for strings"

try:

add(10, "five") except TypeError:
    print("cannot add an int to a string")
```

Já em uma linguagem tipada estaticamente, as funções e objetos têm tipos específicos:

```
def add(a: int, b: int) -> int: return a + b

add(10, 5) # isso deve estar OK

add("hi ", "there") # isso não deve estar OK
```

Na realidade, as versões recentes do Python têm essa funcionalidade (ou algo parecido com ela). A versão anterior do add com as anotações de tipo int é válida no Python 3.6!

Mas essas anotações de tipo não fazem nada. Quando você usa a função add anotada para adicionar strings, a chamada para add(10, "five") ainda gera o mesmo TypeError.

Porém, ainda há (pelo menos) quatro bons motivos para você usar anotações de tipo no código do Python:

- Os tipos são uma importante forma de documentação, o que se aplica essencialmente a um livro que usa código para ensinar conceitos teóricos e matemáticos. Compare estes dois stubs de função:

```
def dot_product(x, y): ...

# ainda não definimos o Vector, mas achamos que sim

def dot_product(x: Vector, y: Vector) -> float: ...
```

Para mim, o segundo está bem mais informativo; espero que você ache isso também. (Agora, já estou tão habituado a indicar tipos que acho difícil ler código não tipado em Python.)

- Existem ferramentas externas (a mais popular é o mypy) que leem o código, inspecionam as anotações de tipo e informam os erros de tipo antes mesmo da execução do código. Por exemplo, se você rodar o mypy em um arquivo contendo o add("hi ", "there"), ele emitirá o seguinte aviso:

```
error: Argument 1 to "add" has incompatible type "str"; expected "int"
```

Como os testes com asserções, essa é uma forma eficiente de encontrar erros no código antes de executá-lo. O texto não abordará esse verificador de tipos; no entanto, nos bastidores, executarei um para confirmar se o livro está correto.

- Quando pensa melhor nos tipos do código, você cria funções e interfaces mais limpas:

```
from typing import Union
```

```
def secretly_ugly_function(value, operation): ...
def ugly_function(value: int,
operation: Union[str, int, float, bool]) -> int:
...
```

Nessa função, o parâmetro operation pode ser uma string, um int, um float, ou um bool. Muito provavelmente, ela é frágil e difícil de usar, mas ficará bem mais consistente quando os tipos forem indicados expressamente, nos motivando a criar um design menos tosco, e os usuários agradecerão.

- Ao usar tipos, você tem acesso aos recursos do editor, como o preenchimento automático (Figura 2-1), e pode ficar irritado com os erros de tipo.

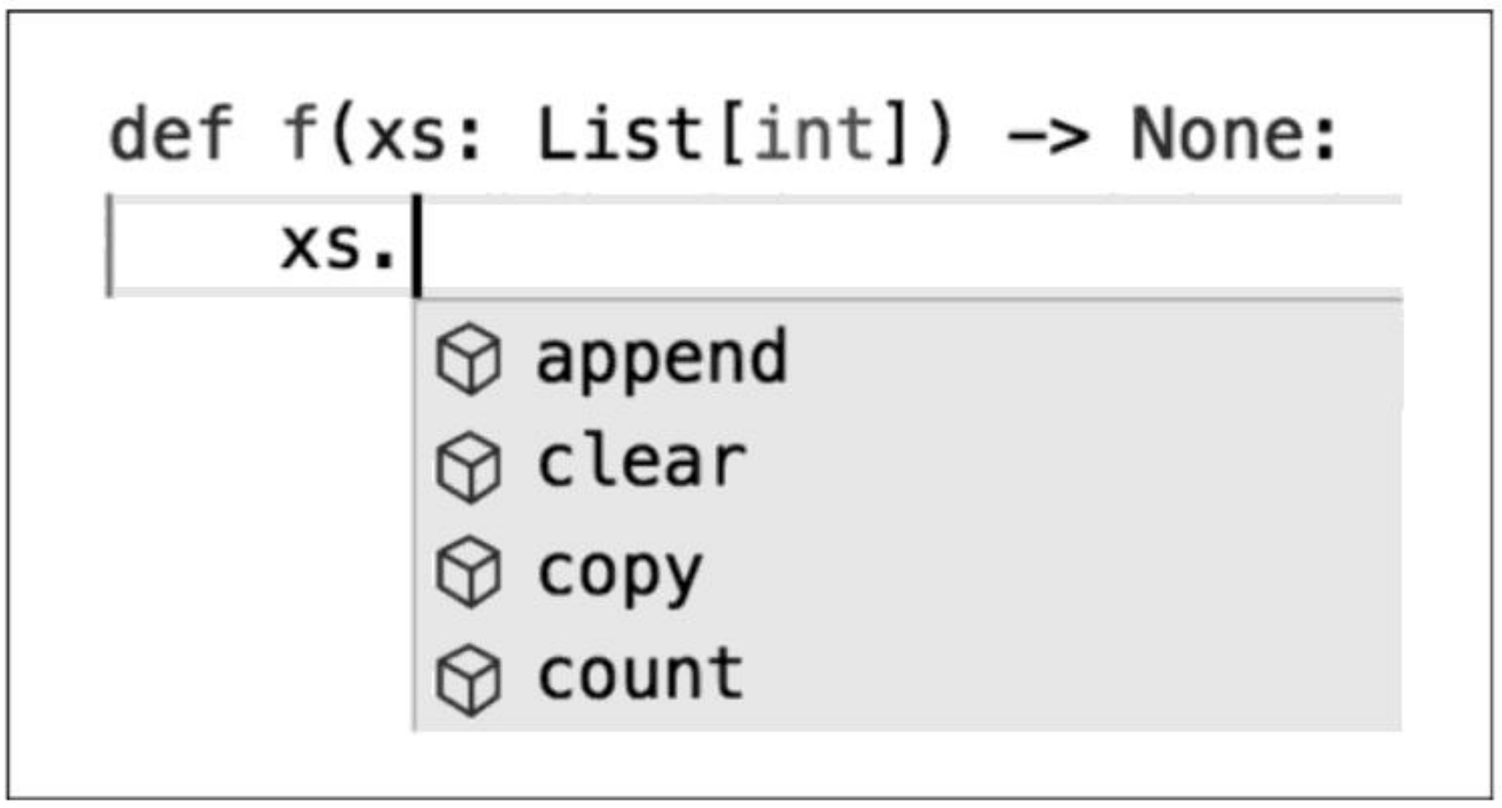

Figura 2-1. VSCode, mas seu editor provavelmente faz a mesma coisa

Há quem diga que as dicas de tipos podem ser importantes em grandes projetos, mas não valem a pena em trabalhos pequenos. Porém, como essas dicas quase não demoram para ser digitadas e o editor ainda economiza bastante tempo, acredito que elas de fato aceleram a escrita do código, mesmo em projetos pequenos.

Por todos esses motivos, o código utilizado ao longo do livro trará anotações de tipo. Alguns leitores talvez fiquem incomodados com isso, no entanto, acredito que, até o final do curso, eles mudarão de ideia.

Como Escrever Anotações de Tipo

Como já vimos, quando você usa tipos internos como int, bool e float, o tipo em si serve como anotação. Mas e se você tiver (digamos) uma list?

```
def total(xs: list) -> float: return sum(total)
```

Isso não está errado, mas o tipo não é suficientemente específico. Aqui, queremos que o xs seja uma

lista de floats, não (digamos) uma lista de strings.

O módulo typing dispõe de muitos tipos parametrizados que podemos usar para fazer isso:

```
from typing import List # note capital L
def total(xs: List[float]) -> float: return sum(total)
```

Até agora, apenas especificamos anotações para parâmetros de função e tipos de retorno. Para as variáveis, geralmente o tipo é mais óbvio:

```
# É assim que anotamos os tipos de variáveis quando as definimos.
# Mas isso é desnecessário; "obviamente", x é um int.
x: int = 5
```

Entretanto, às vezes, o tipo não é tão óbvio:

```
values = [] # qual é o meu tipo?
best_so_far = None # qual é o meu tipo?
```

Nesses casos, indicaremos dicas de tipo em linha:

```
from typing import Optional
values: List[int] = []
best_so_far: Optional[float] = None # pode ser um float ou None
```

O módulo typing contém muitos tipos, mas só usaremos alguns deles:

```
# as anotações de tipo neste trecho são desnecessárias from typing import Dict, Iterable, Tuple
# as chaves são strings, os valores são ints
counts: Dict[str, int] = {'data': 1, 'science': 2}
# as listas e geradores são iteráveis
if lazy:
evens: Iterable[int] = (x for x in range(10) if x % 2 == 0) else:
evens = [0, 2, 4, 6, 8]
# as tuplas especificam um tipo para cada elemento
triple: Tuple[int, float, int] = (10, 2.3, 5)
```

Finalmente, como o Python tem funções de primeira classe, também precisamos de um tipo para representá-las. Confira este exemplo bem artificial:

```
from typing import Callable
# A dica de tipo indica que o repetidor é uma função que recebe
# dois argumentos, uma string e um int, e retorna uma string.
def twice(repeater: Callable[[str, int], str], s: str) -> str: return repeater(s, 2)

def comma_repeater(s: str, n: int) -> str:
n_copies = [s for _ in range(n)]
return ', '.join(n_copies)

assert twice(comma_repeater, "type hints") == "type hints, type hints"
```

Como as anotações de tipo são objetos Python, podemos atribuí-las a variáveis para facilitar as referências a elas:

```
Number = int
Numbers = List[Number]
def total(xs: Numbers) -> Number: return sum(xs)
```

Ao final do livro, você saberá ler e escrever anotações de tipo e, assim espero, estará utilizando essas anotações no seu código.

Seja bem-vindo à DataSciencester!

Esse foi o curso de formação inicial. Ah, em tempo: não vá surrupiar nada!

Materiais Adicionais

- O mundo não sofre com a escassez de tutoriais do Python. O site oficial (*https://docs.python.org/3/tutorial/*) é um bom ponto de partida.
- Se você estiver disposto, o tutorial oficial do IPython (*http://ipython.readthedocs.io/en/stable/interactive/index.html*) também é uma boa introdução. Leia agora mesmo.
- A documentação do mypy (*https://mypy.readthedocs.io/en/stable/*) contém muitas informações sobre anotações e verificação de tipo no Python.

CAPÍTULO 3
Visualizando Dados

Acredito que a visualização seja uma das formas mais poderosas de atingir metas pessoais.
—Harvey Mackay

Um item essencial do kit de ferramentas do cientista de dados é a visualização de dados. Embora seja muito fácil criá-las, é bem difícil produzir boas visualizações.

A visualização de dados tem duas funções básicas:

- Explorar dados;
- Comunicar dados.

Neste capítulo, desenvolveremos as habilidades necessárias para começar a explorar dados e produzir as visualizações que utilizaremos ao longo do livro. Como a maioria dos tópicos abordados aqui, a visualização de dados é uma área de estudos muito ampla e merece um livro exclusivo. Entretanto, ainda assim, tentarei mostrar o que caracteriza ou não uma boa visualização.

matplotlib

Existem muitas ferramentas de visualização de dados. Aqui, usaremos a biblioteca matplotlib (*http://matplotlib.org/*), um recurso muito popular (embora já esteja envelhecendo). Se você estiver interessado em produzir visualizações elaboradas e interativas para a web, essa provavelmente não é a melhor opção, mas, para gráficos simples de barras, de linhas e de dispersão, ela funciona muito bem.

Como vimos antes, o matplotlib não integra a biblioteca principal do Python. Então, ative o ambiente virtual (para aprender a configurá-lo, volte para a seção "Ambientes Virtuais" e siga as instruções) e instale a biblioteca usando este comando:

```
python -m pip install matplotlib
```

Usaremos o módulo matplotlib.pyplot. Em essência, o pyplot mantém um estado interno no qual você pode construir uma visualização passo a passo. Ao terminar, você pode salvá-la com savefig ou exibi-la com show.

Por exemplo, é bem fácil fazer um gráfico simples (como o da Figura 3-1):

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]

gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# crie um gráfico de linhas, anos no eixo x, gdp no eixo y
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')
```

```
# adicione um título plt.title("Nominal GDP")
# adicione um rótulo ao eixo y
plt.ylabel("Billions of $") plt.show()
```

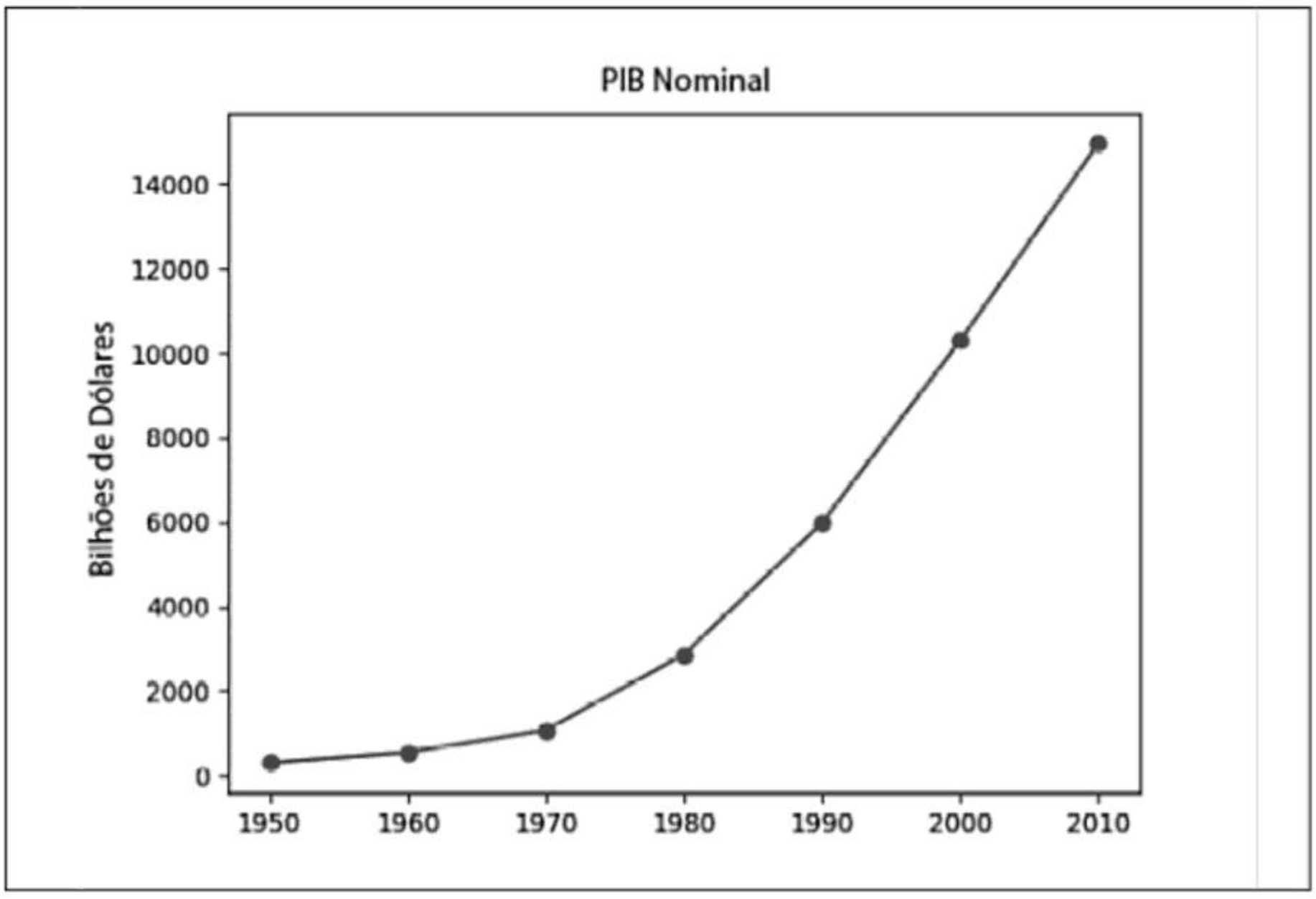

Figura 3-1. Um gráfico de linhas simples

Aprender a criar gráficos de excelente qualidade é mais complicado e não cabe a este capítulo. Há muitas formas de personalizar os gráficos com rótulos de eixos, estilos de linha e marcadores de ponto, por exemplo. Porém, em vez de fazer uma análise minuciosa dessas opções, usaremos (e destacaremos) apenas algumas delas nos exemplos.

Não aplicaremos muito essa funcionalidade, mas o matplotlib pode produzir gráficos complexos e sobrepostos, com formatação sofisticada e visualizações interativas. Confira a documentação (*https://matplotlib.org*) para se aprofundar mais nesse tema.

Gráficos de Barras

Um gráfico de barras é uma boa opção para mostrar como algumas quantidades variam em um conjunto discreto de itens. Por exemplo, a Figura 3-2 representa o número de
Oscars recebido pelos filmes indicados:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]
```

```
# plote as barras com coordenadas x à esquerda [0, 1, 2, 3, 4], alturas [num_oscars]

plt.bar(range(len(movies)), num_oscars)

plt.title("My Favorite Movies") # adicione um título
plt.ylabel("# of Academy Awards") # rotule o eixo y

# rotule o eixo x com os nomes dos filmes nos centros das barras
plt.xticks(range(len(movies)), movies)

plt.show()
```

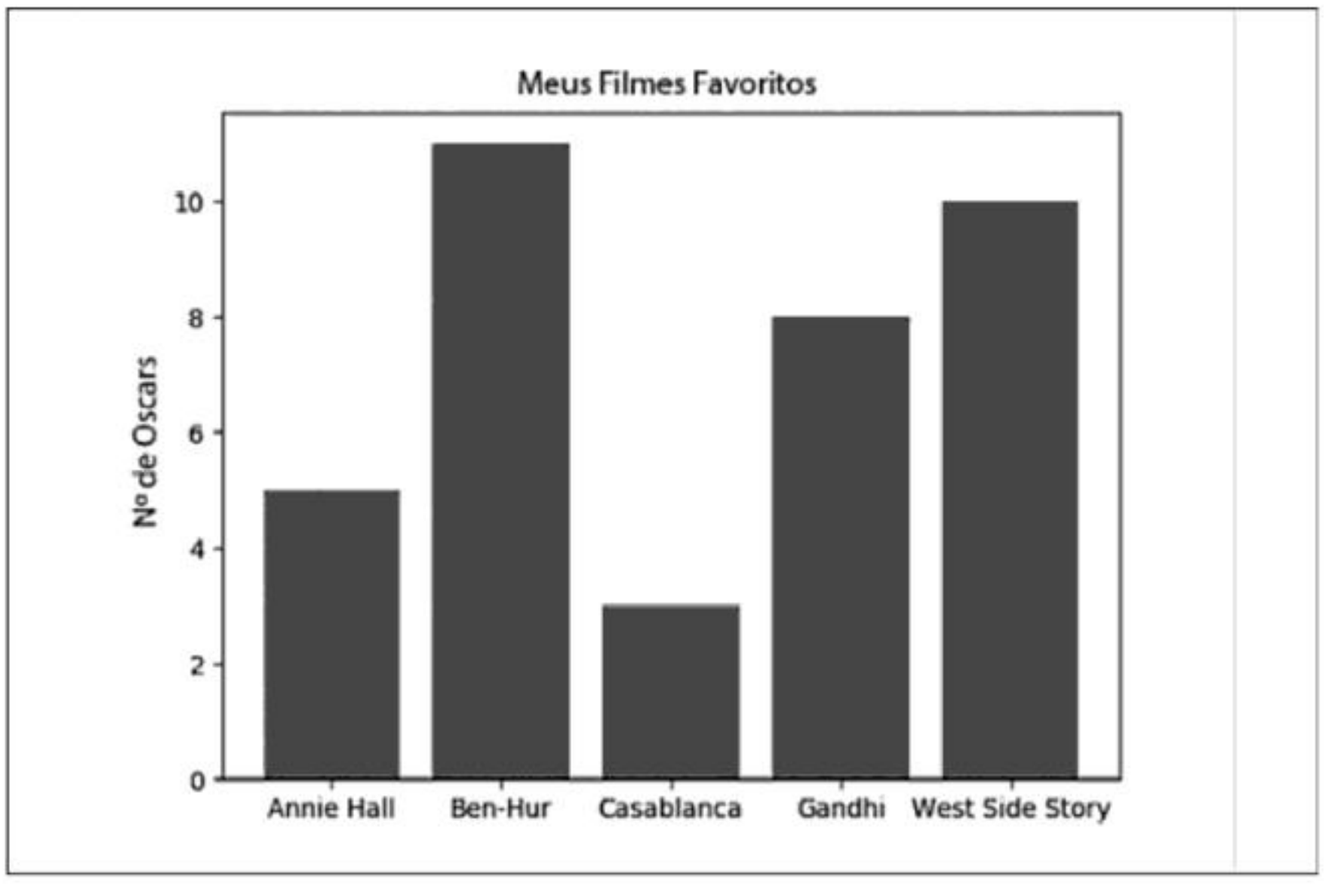

Figura 3-2. Um gráfico de barras simples

Um gráfico de barras também pode ser uma boa opção para plotar histogramas de valores numéricos agrupados e representar visualmente a distribuição dos valores, como na Figura 3-3:

```
from collections import Counter

grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]

# Agrupe as notas por decil, mas coloque o 100 com o 90

histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)

plt.bar([x + 5 for x in histogram.keys()], # Mova as barras para a direita em 5

histogram.values(), # Atribua a altura correta a cada barra

10, # Atribua a largura 10 a cada barra

edgecolor=(0, 0, 0)) # Escureça as bordas das barras

plt.axis([-5, 105, 0, 5]) # eixo x de -5 a 105,

# eixo y de 0 a 5

plt.xticks([10 * i for i in range(11)]) # rótulos do eixo x em 0, 10, ..., 100
plt.xlabel("Decile")

plt.ylabel("# of Students")

plt.title("Distribution of Exam 1 Grades") plt.show()
```

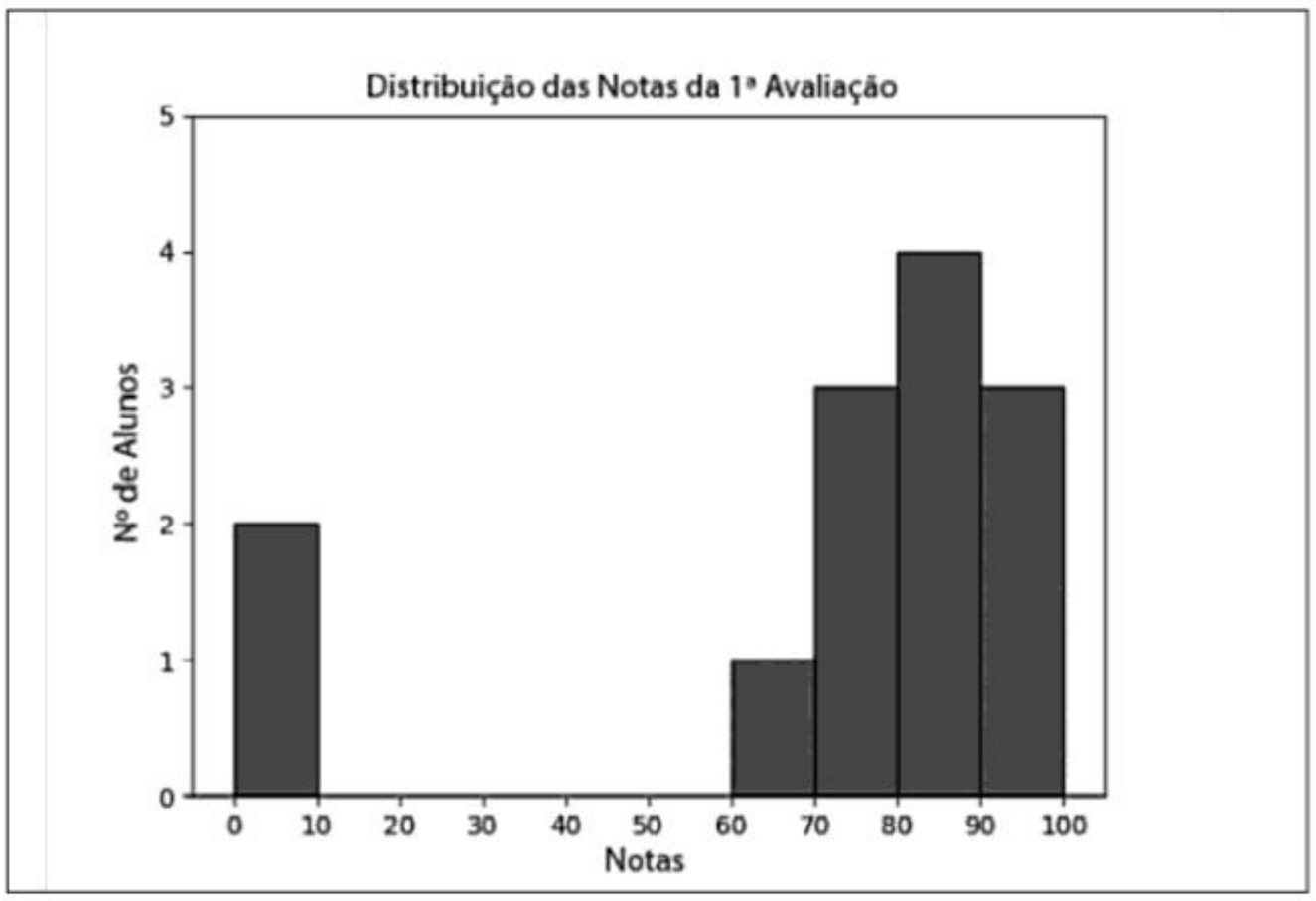

Figura 3-3. Criando um histograma com um gráfico de barras

O terceiro argumento para plt.bar especifica a largura da barra. Aqui, definimos a largura em 10 para preencher completamente o decil. Além disso, movemos as barras para a direita em 5 de modo que a barra "10" (que corresponde ao decil 10–20) ficasse com o centro em 15, ocupando, assim, o intervalo correto. Também adicionamos bordas escuras para otimizar a visualização.

A chamada para plt.axis indica que o eixo x deve variar entre –5 e 105 (deixando um pequeno espaço à esquerda e à direita) e que o eixo y deve variar entre 0 e 5. A chamada para plt.xticks coloca os rótulos do eixo x em 0, 10, 20, ..., 100.

Seja criterioso ao usar o plt.axis. Nos gráficos de barras, é um grande vacilo não iniciar o eixo y em 0, pois isso pode ser uma malandragem para desorientar os usuários (Figura 3-4):

```
mentions = [500, 505]

years = [2017, 2018]

plt.bar(years, mentions, 0.8) plt.xticks(years)

plt.ylabel("# of times I heard someone say 'data science'")

# se você não fizer isso, matplotlib rotulará o eixo x como 0, 1 # e adicionará um +2.013e3 off no canto (que feio,
matplotlib!) plt.ticklabel_format(useOffset=False)

# o eixo y malandro mostra apenas a parte acima de 500

plt.axis([2016.5, 2018.5, 499, 506])

plt.title("Look at the 'Huge' Increase!") plt.show()
```

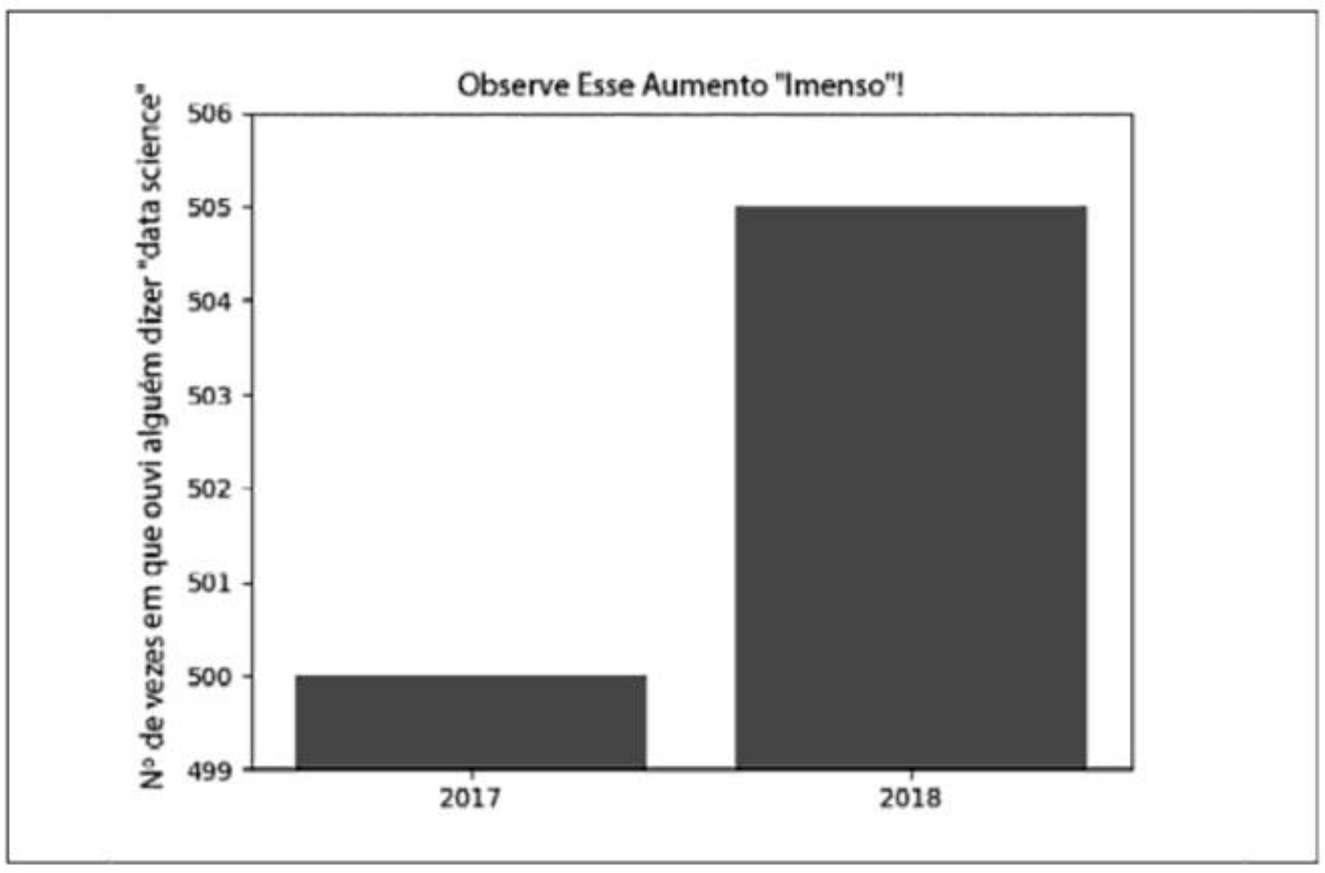

Figura 3-4. Um gráfico com um eixo y malandro

Na Figura 3-5, usamos eixos mais sensíveis, com um resultado bem menos expressivo:

```
plt.axis([2016.5, 2018.5, 0, 550])
plt.title("Not So Huge Anymore") plt.show()
```

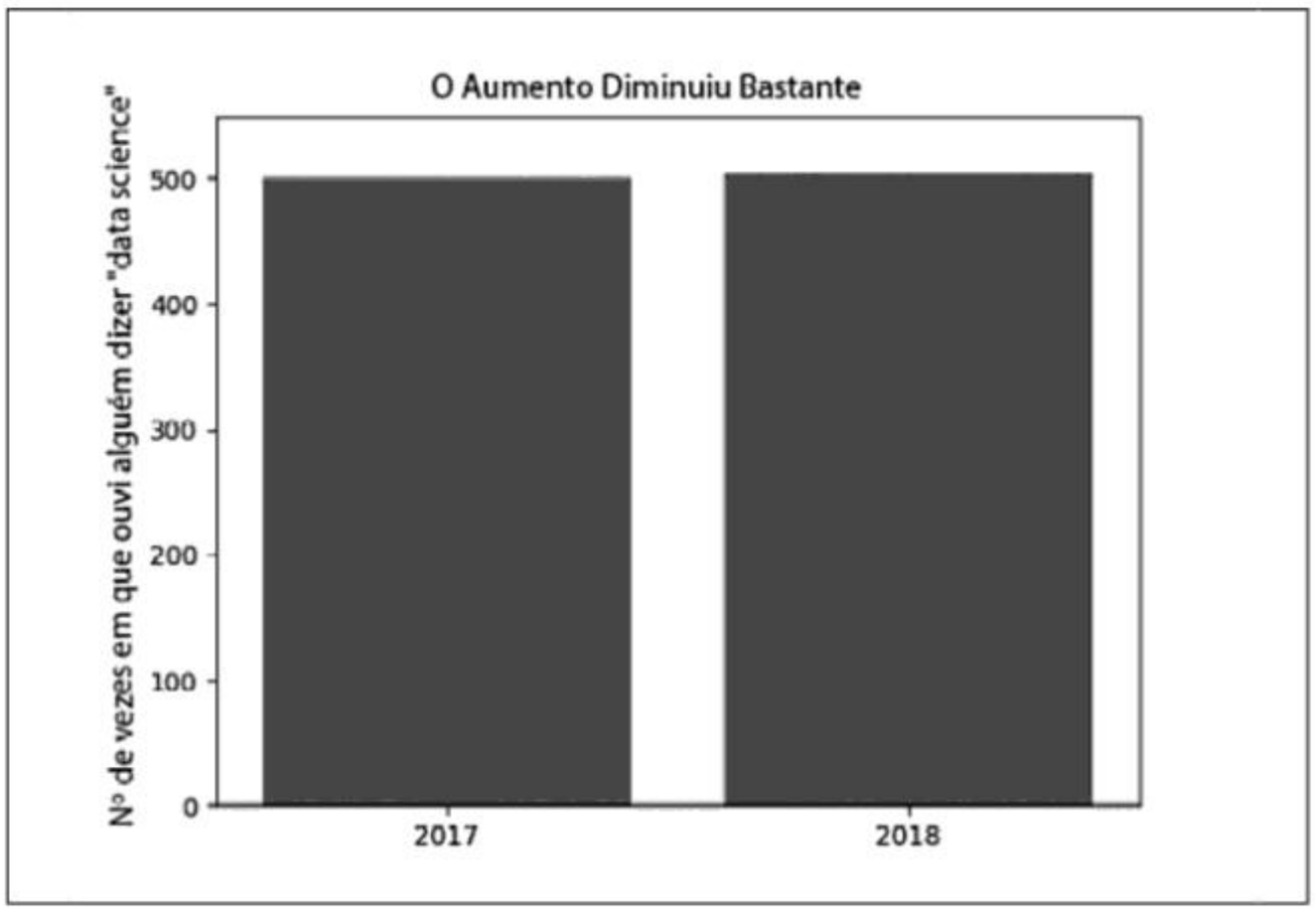

Figura 3-5. O mesmo gráfico com um eixo y boa-praça

Gráficos de Linhas

Como vimos antes, podemos criar gráficos de linha usando o plt.plot. Essa é uma boa opção para mostrar tendências, como na Figura 3-6:

```
variance = [1, 2, 4, 8, 16, 32, 64, 128, 256]

bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]

total_error = [x + y for x, y in zip(variance, bias_squared)] xs = [i for i, _ in enumerate(variance)]

# Podemos fazer múltiplas chamadas para plt.plot

# para mostrar múltiplas séries no mesmo gráfico

plt.plot(xs, variance, 'g-', label='variance') # linha verde sólida
plt.plot(xs, bias_squared, 'r-.', label='bias^2') # linha vermelha de ponto
tracejado

plt.plot(xs, total_error, 'b:', label='total error') # linha pontilhada azul
```

```
# Como atribuímos rótulos a cada série,
# podemos criar uma legenda de graça (loc=9 means "top center")
plt.legend(loc=9)
plt.xlabel("model complexity") plt.xticks([])
plt.title("The Bias-Variance Tradeoff") plt.show()
```

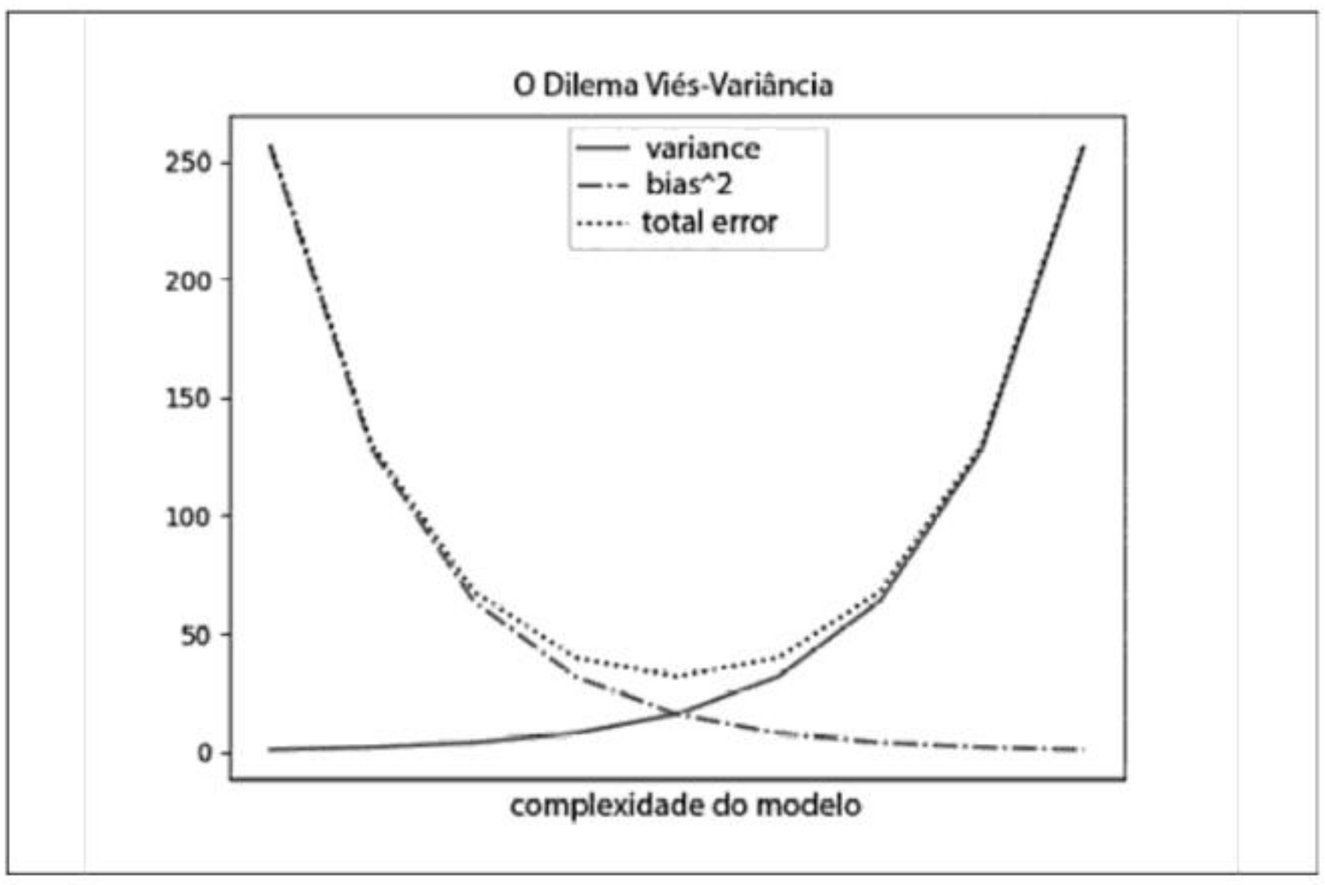

Figura 3-6. Vários gráficos de linhas com uma legenda

Gráficos de Dispersão

O gráfico de dispersão é a opção certa para representar as relações entre pares de conjuntos de dados. Por exemplo, a Figura 3-7 ilustra as relações entre o número de amigos dos usuários e o número de minutos que eles passam no site por dia:

```
friends = [ 70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
plt.scatter(friends, minutes)
# rotule cada ponto
for label, friend_count, minute_count in zip(labels, friends, minutes): plt.annotate(label,
xy=(friend_count, minute_count), # Coloque o rótulo no respectivo ponto
xytext=(5, -5), # mas levemente deslocado
textcoords='offset points')
plt.title("Daily Minutes vs. Number of Friends") plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site") plt.show()
```

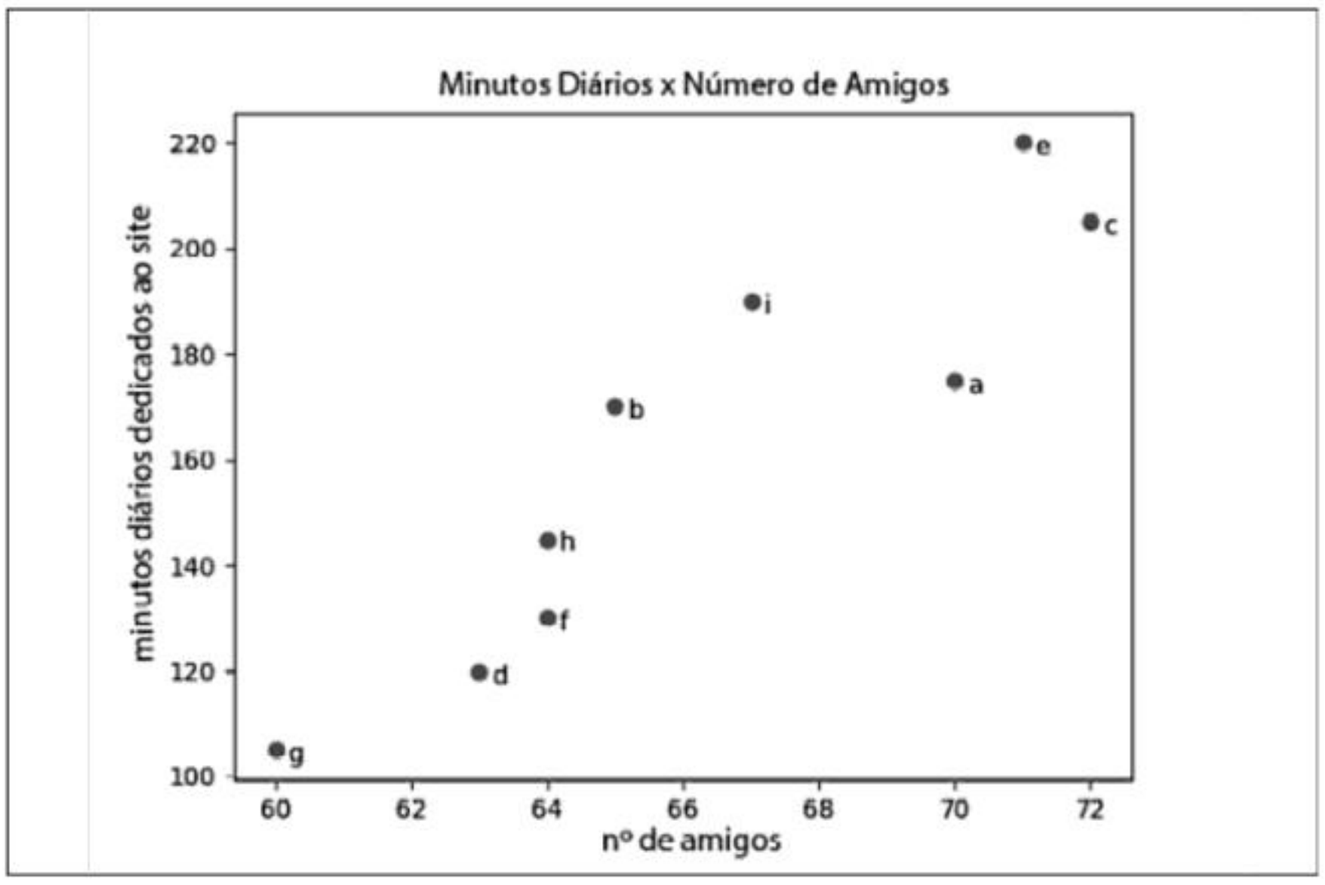

Figura 3-7. Um gráfico de dispersão entre o número de amigos e o tempo dedicado ao site

Se permitir que o matplotlib selecione a escala ao dispersar variáveis comparáveis, talvez você obtenha uma imagem equivocada, como na Figura 3-8.

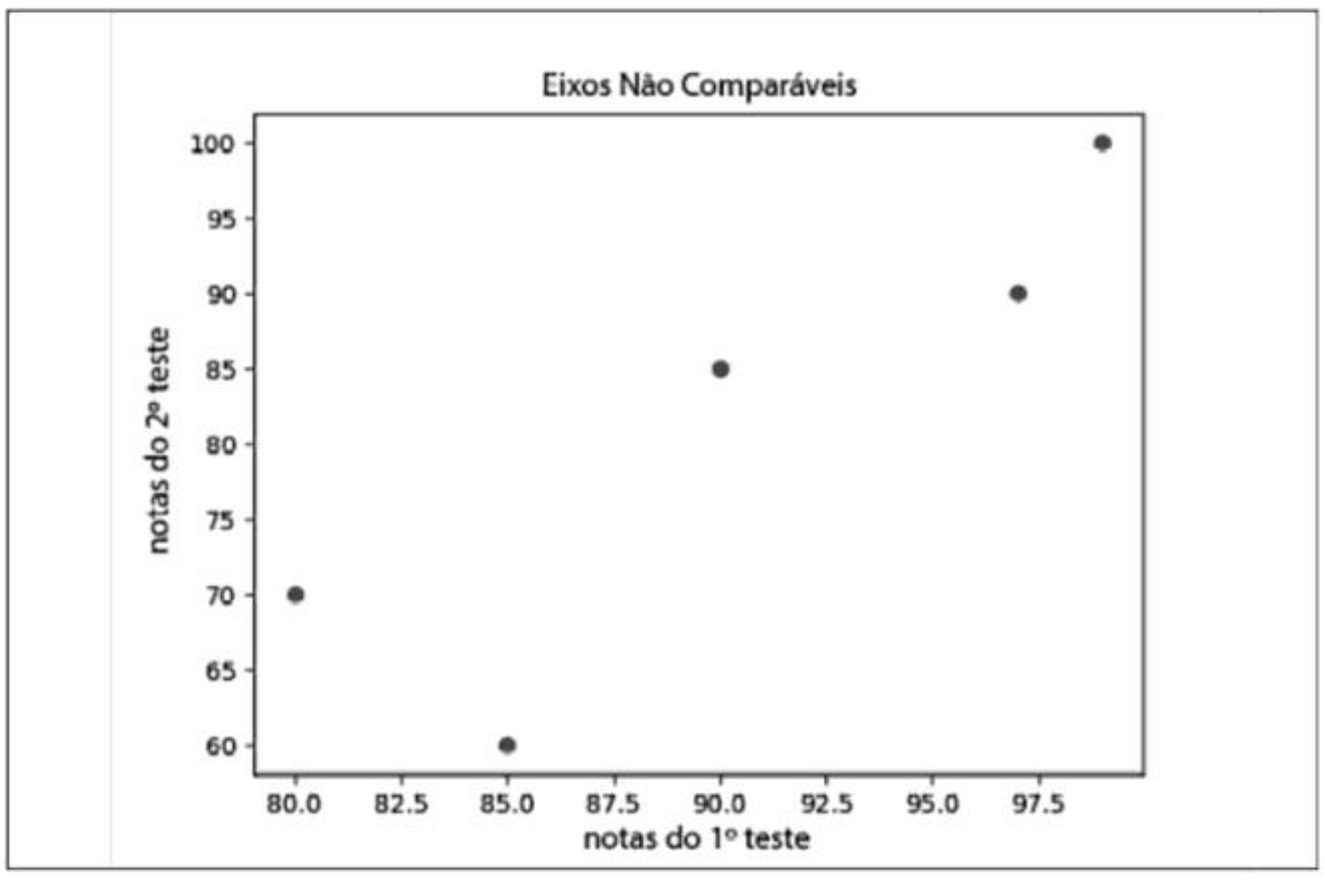

Figura 3-8. Um gráfico de dispersão com eixos incomparáveis

```
test_1_grades = [ 99, 90, 85, 97, 80]

test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades) plt.title("Axes Aren't Comparable")
plt.xlabel("test 1 grade")

plt.ylabel("test 2 grade") plt.show()
```

Quando incluímos uma chamada para plt.axis("equal"), o gráfico (Figura 3-9) mostra com mais precisão que a maior parte da variação acontece no teste 2.

Isso é tudo que você precisa saber para começar a criar visualizações. Aprenderemos muito mais sobre esse tema ao longo do livro.

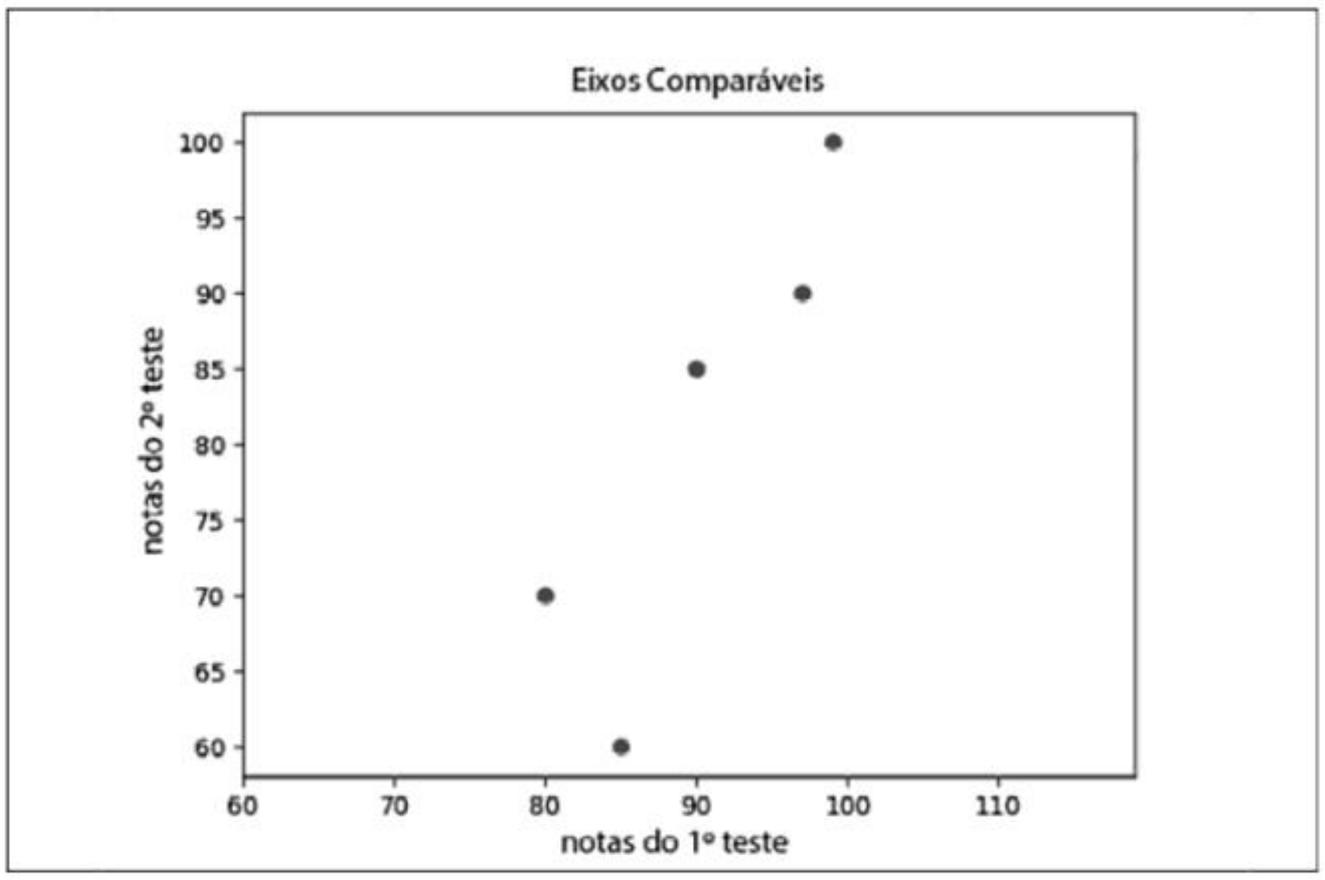

Figura 3-9. O mesmo gráfico de dispersão com eixos iguais

Materiais Adicionais

- A galeria do matplotlib (*https://matplotlib.org/gallery.html*) é uma boa introdução para quem quer aprender a usar (e desenvolver) as funções do matplotlib;
- O seaborn (*https://seaborn.pydata.org/*) é executado no matplotlib e facilita a criação de visualizações mais sofisticadas (e mais complexas);
- O Altair (*https://altair-viz.github.io/*) é uma nova biblioteca Python que cria visualizações declarativas;
- O D3.js (*http://d3js.org*) é uma biblioteca JavaScript que produz visualizações sofisticadas e interativas para a web. Embora não seja integrado ao Python, é muito popular; vale a pena conhecê-lo;
- O Bokeh (*http://bokeh.pydata.org*) é uma biblioteca que introduz o estilo de visualização D3 no Python.

CAPÍTULO 4
Álgebra Linear

Existe algo mais inútil ou menos útil do que Álgebra?
—Billy Connolly

A Álgebra Linear é o ramo da matemática que calcula espaços vetoriais. Embora seja difícil ensinar essa matéria em um só capítulo, ela é fundamental para um grande número de conceitos e técnicas de data science; ou seja, vamos, pelo menos, fazer uma tentativa. Os tópicos deste capítulo serão aplicados várias vezes ao longo do livro.

Vetores

Em teoria, os vetores são objetos que podem ser somados ou multiplicados por escalares (como, por exemplo, números) para formar outros vetores.

Na prática (para nós), os vetores são pontos em um espaço de dimensão finita. Embora você não pense nos dados como vetores, essa é uma ótima forma de representar dados numéricos.

Por exemplo, se tivermos as alturas, pesos e idades de muitas pessoas, podemos tratar esses dados como vetores tridimensionais [height, weight, age]. Se você passar quatro avaliações para uma turma, pode tratar as notas dos alunos como vetores quadridimensionais [exam1, exam2, exam3, exam4].

Saindo do zero, a abordagem mais simples é representar vetores como listas de números. Uma lista com três números corresponde a um vetor em um espaço tridimensional e vice-versa.

Para isso, usaremos um alias de tipo indicando que um Vector é só uma list de floats:

```
from typing import List

Vector = List[float]

height_weight_age = [70, # polegadas,
170, # libras,
40 ] # anos

grades = [95, # teste1

80, # teste2

75, # teste3

62 ] # teste4
```

Também faremos cálculos aritméticos com os vetores. Como as lists do Python não são vetores (o que não facilita a aritmética vetorial), temos que construir essas ferramentas aritméticas. Então, vamos começar com isso.

Primeiro, geralmente é necessário somar dois vetores. A soma ou adição de vetores ocorre por

componente. Ou seja, se os vetores v e w tiverem o mesmo tamanho, a adição produzirá um vetor cujo primeiro elemento será v[0] + w[0], cujo segundo elemento será v[1] + w[1] e assim por diante. (Não podemos adicionar vetores com tamanhos diferentes.)

Por exemplo, a soma dos vetores [1, 2] e [2, 1] produz [1 + 2, 2 + 1] ou [3,3], com vemos na Figura 4-1.

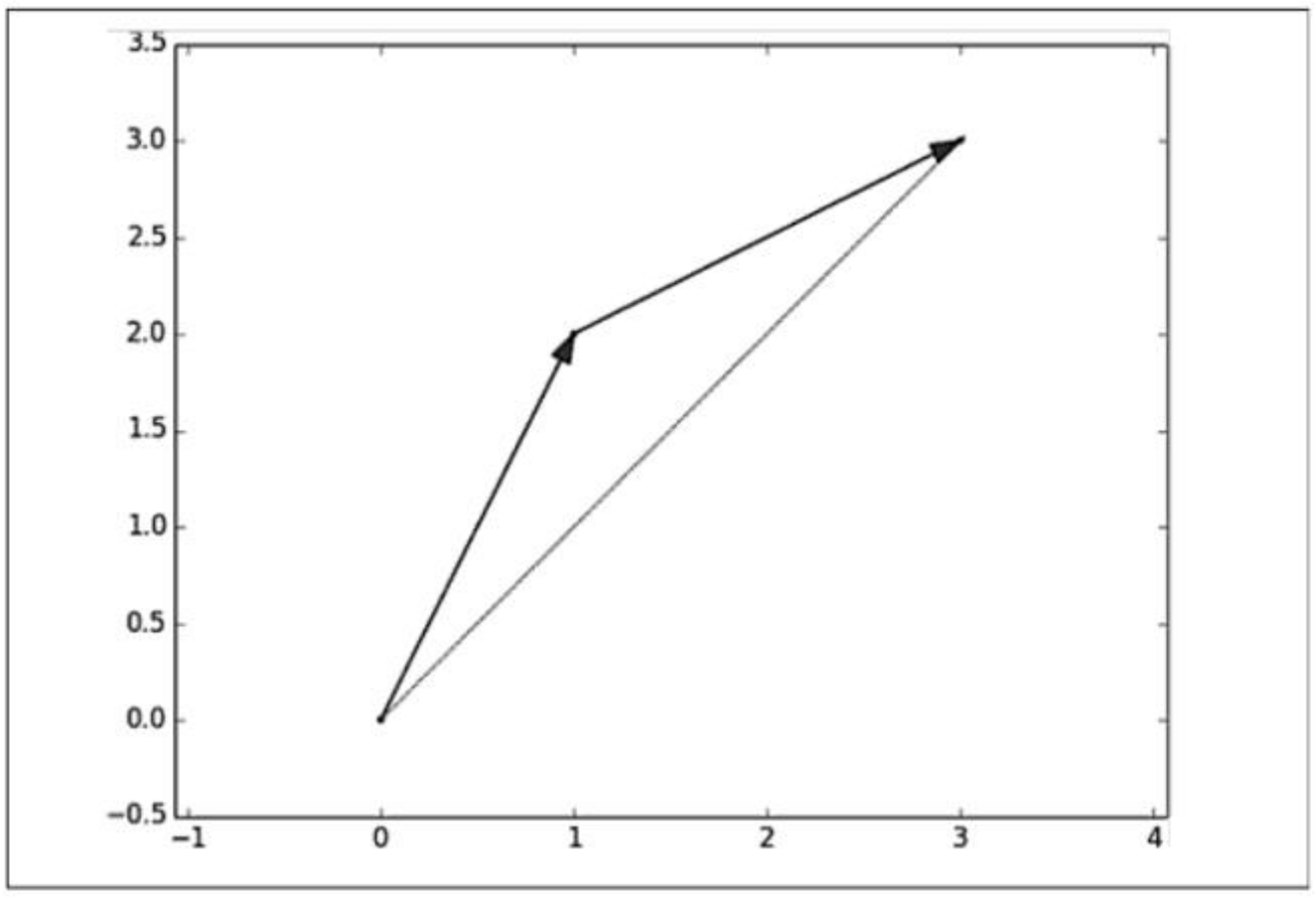

Figura 4-1. Somando dois vetores

Isso pode ser implementado facilmente ao compactar os vetores com zip e aplicar uma compreensão de lista para adicionar os elementos correspondentes:

```
def add(v: Vector, w: Vector) -> Vector:
"""Soma os elementos correspondentes"""
assert len(v) == len(w), "vectors must be the same length"
return [v_i + w_i for v_i, w_i in zip(v, w)]
assert add([1, 2, 3], [4, 5, 6]) == [5, 7, 9]
```

Da mesma forma, para subtrair dois vetores, basta subtrair os elementos correspondentes:

```
def subtract(v: Vector, w: Vector) -> Vector:
"""Subtrai os elementos correspondentes"""
assert len(v) == len(w), "vectors must be the same length"
return [v_i - w_i for v_i, w_i in zip(v, w)]
assert subtract([5, 7, 9], [4, 5, 6]) == [1, 2, 3]
```

Às vezes, é preciso somar uma lista de vetores por componente. Para isso, crie um vetor cujo primeiro elemento seja a soma de todos os primeiros elementos, cujo segundo elemento seja a soma de todos os segundos elementos e assim por diante:

```
def vector_sum(vectors: List[Vector]) ->
Vector: """Soma todos os elementos correspondentes"""
# Verifique se os vetores não estão vazios
assert vectors, "no vectors provided!"
# Verifique se os vetores são do mesmo tamanho
num_elements = len(vectors[0])
```

```
assert all(len(v) == num_elements for v in vectors), “different sizes!”
# o elemento de nº i do resultado é a soma de todo vector[i] return [sum(vector[i] for vector in vectors)
for i in range(num_elements)]
assert vector_sum([[1, 2], [3, 4], [5, 6], [7, 8]]) == [16, 20]
```

Também multiplicaremos um vetor por um escalar; para isso, basta multiplicar cada elemento do vetor pelo número em questão:

```
def scalar_multiply(c: float, v: Vector) -> Vector: “””Multiplica cada elemento por c”””
return [c * v_i for v_i in v]
assert scalar_multiply(2, [1, 2, 3]) == [2, 4, 6]
```

Assim, podemos computar a média dos componentes de uma lista de vetores (do mesmo tamanho):

```
def vector_mean(vectors: List[Vector]) -> Vector: “””Computa a média dos elementos”””
n = len(vectors)
return scalar_multiply(1/n, vector_sum(vectors))
assert vector_mean([[1, 2], [3, 4], [5, 6]]) == [3, 4]
```

Uma ferramenta menos conhecida é o produto escalar (ou dot product). O produto escalar de dois vetores é a soma dos produtos por componente:

```
def dot(v: Vector, w: Vector) -> float:
“””Computa v_1 * w_1 + ... + v_n * w_n”””
assert len(v) == len(w), “vectors must be same length”
return sum(v_i * w_i for v_i, w_i in zip(v, w))
assert dot([1, 2, 3], [4, 5, 6]) == 32 # 1 * 4 + 2 * 5 + 3 * 6
```

Quando w tem magnitude 1, o produto escalar mede a extensão do vetor v na direção w. Por exemplo, se w = [1, 0], então dot(v, w) é apenas o primeiro componente de v.Em outras palavras, esse é o comprimento do vetor quando você projeta v em w (Figura 4-2).

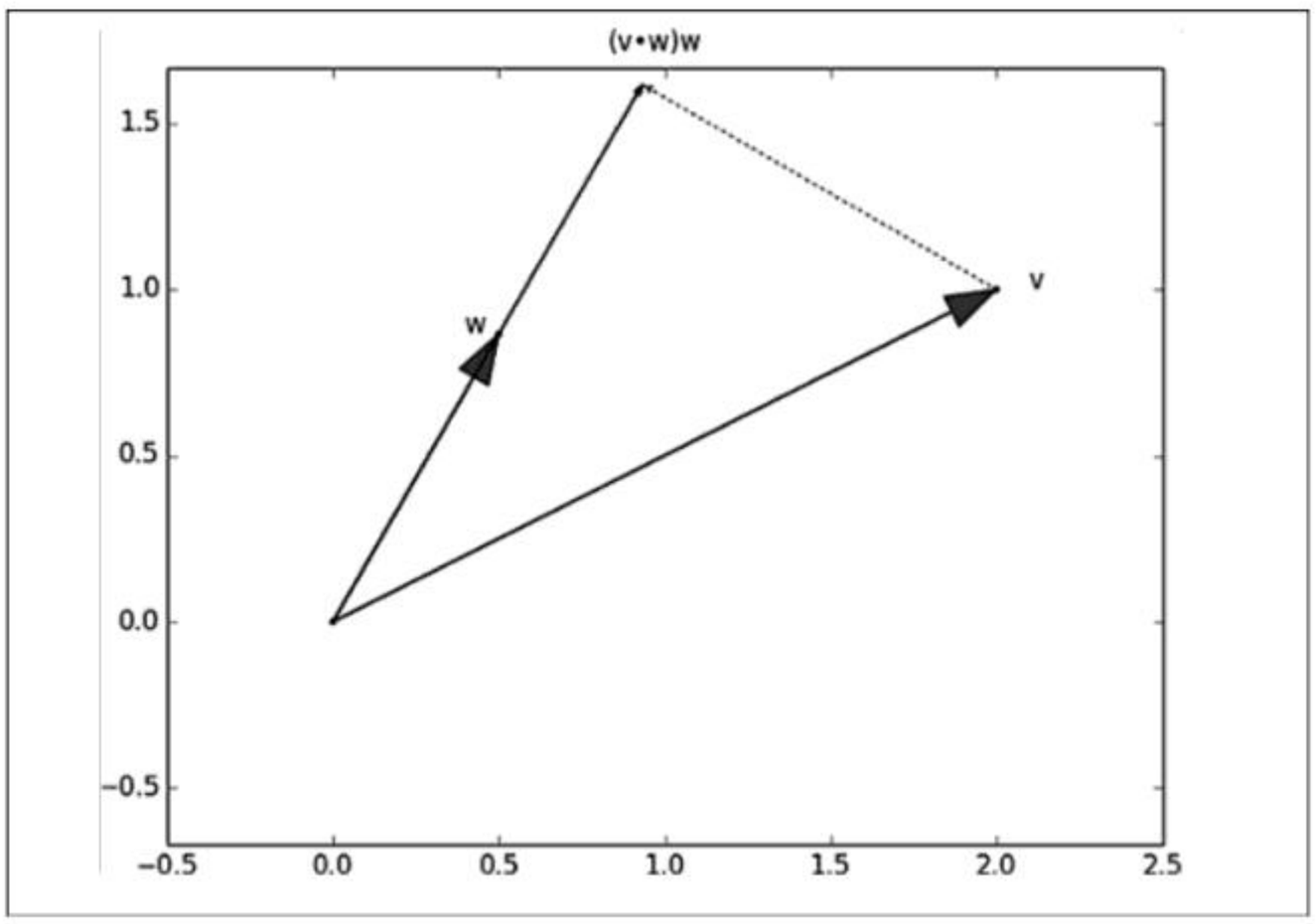

Figura 4-2. O produto escalar como projeção de vetor

Dessa forma, é fácil computar a soma dos quadrados de um vetor:

```
def sum_of_squares(v: Vector) -> float:
```

```
"""Retorna v_1 * v_1 + ... + v_n * v_n""" return dot(v, v)
assert sum_of_squares([1, 2, 3]) == 14 # 1 * 1 + 2 * 2 + 3 * 3
```

Podemos usar esse valor para computar a magnitude (ou comprimento) do vetor:

```
import math
def magnitude(v: Vector) -> float:
"""Retorna a magnitude (ou comprimento) de v""""
return math.sqrt(sum_of_squares(v)) # math.sqrt é a função de raiz quadrada
assert magnitude([3, 4]) == 5
```

Agora temos tudo que precisamos para computar a distância entre os dois vetores, definida da seguinte forma:

$$\sqrt{(v_1 - w_1)^2 + ... + (v_n - w_n)^2}$$

Em código:

```
def squared_distance(v: Vector, w: Vector) -> float:
"""Computa (v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2""" return sum_of_squares(subtract(v, w))
def distance(v: Vector, w: Vector) -> float: """Computa a distância entre v e w"""
return math.sqrt(squared_distance(v, w))
```

Talvez fique mais claro da seguinte forma (equivalente):

```
def distance(v: Vector, w: Vector) -> float: return magnitude(subtract(v, w))
```

Isso é tudo que precisamos para começar; usaremos essas funções várias vezes ao longo do livro.

> Usar listas como vetores é bom como apresentação, mas terrível para o desempenho.
>
> No código de produção, use a biblioteca NumPy, que contém uma classe array de alto desempenho com diversas operações aritméticas.

Matrizes

Uma matriz é uma coleção bidimensional de números. Representaremos as matrizes como listas de listas; as listas internas terão o mesmo tamanho e representarão as linhas da matriz. Se A é uma matriz, então A[i][j] é o elemento que está na linha i e na coluna j. Por convenção matemática, usaremos letras maiúsculas para representar matrizes na maioria das vezes. Por exemplo:

```
# Outro alias de tipo
Matrix = List[List[float]]

A = [[1, 2, 3], # A tem 2 linhas e 3 colunas

[4, 5, 6]]
```

```
B = [[1, 2], # B tem 3 linhas e 2 colunas
[3, 4],
[5, 6]]
```

Na matemática, costumamos dizer que a primeira linha da matriz é a "linha 1" e a primeira coluna é a "coluna 1". Mas, como estamos representando matrizes com as lists do Python (que são indexadas em zero), a primeira linha da matriz será a "linha 0" e a primeira coluna será a "coluna 0".

Como representa uma lista de listas, a matriz A contém as linhas len(A) e colunas len(A[0]) que consideramos seu shape [formato]:

```
from typing import Tuple
def shape(A: Matrix) -> Tuple[int, int]:
"""Retorna (nº de linhas de A, nº de colunas de A)"""
num_rows = len(A)
num_cols = len(A[0]) if A else 0 # número de elementos na primeira linha
return num_rows, num_cols
assert shape([[1, 2, 3], [4, 5, 6]]) == (2, 3) # 2 linhas, 3 colunas
```

Se a matriz tem n linhas e k colunas, dizemos que ela é uma matriz n x k. Podemos (e, às vezes, iremos) considerar cada linha da matriz n x k como um vetor de comprimento k e cada coluna como um vetor de comprimento n:

```
def get_row(A: Matrix, i: int) -> Vector:
"""Retorna a linha i de A (como um Vector)"""
return A[i] # A[i] já está na linha i
def get_column(A: Matrix, j: int) -> Vector:
"""Retorna a coluna j de A (como um Vector)"""
return [A_i[j] # elemento j da linha A_i
for A_i in A] # para cada linha A_i
```

Também criaremos matrizes, produzindo seus elementos a partir da sua forma e de uma função. Para isso, usamos uma compreensão de lista aninhada:

```
from typing import Callable
def make_matrix(num_rows: int,
num_cols: int,
entry_fn: Callable[[int, int], float]) -> Matrix:
"""
Retorna uma matriz num_rows x num_cols cuja entrada (i,j) é entry_fn(i, j)
"""
return [[entry_fn(i, j) # com i, crie uma lista for j in range(num_cols)] # [entry_fn(i, 0), ... ]
```

```
for i in range(num_rows)] # crie uma lista para cada i
```

Com esta função, você pode criar uma matriz de identidade 5 × 5 (com 1s na diagonal e 0s nos outros pontos):

```
def identity_matrix(n: int) -> Matrix:
"""Retorna a matriz de identidade n x n"""

return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

assert identity_matrix(5) == [[1, 0, 0, 0, 0],

[0, 1, 0, 0, 0],

[0, 0, 1, 0, 0],

[0, 0, 0, 1, 0],

[0, 0, 0, 0, 1]]
```

Utilizaremos as matrizes de várias formas.

Primeiro, podemos usar uma matriz para representar um conjunto de dados com múltiplos vetores, considerando cada vetor como uma linha da matriz. Por exemplo, temos a altura, o peso e a idade de mil pessoas e colocamos esses dados em uma matriz 1000 × 3:

```
data = [[70, 170, 40],

[65, 120, 26],

[77, 250, 19],

# ....

]
```

Segundo, como veremos mais diante, podemos usar uma matriz n × k para representar uma função linear que mapeia vetores dimensionais k com relação a vetores dimensionais n. Essas funções integram várias técnicas e conceitos que utilizaremos.

Terceiro, as matrizes podem representar relações binárias. No Capítulo 1, representamos as extremidades de uma rede como uma coleção de pares (i, j). Como alternativa, é possível criar uma matriz A em que A[i][j] será igual a 1 (se os nós i e j estiverem conectados) ou 0 (nos outros casos).

Antes a situação era a seguinte:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),

(4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

Também podemos representar isso da seguinte forma:

```
# usuário 0 1 2 3 4 5 6 7 8 9
#
friend_matrix = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # usuário 0

[1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # usuário 1

[1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # usuário 2

[0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # usuário 3

[0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # usuário 4

[0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # usuário 5

[0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # usuário 6

[0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # usuário 7
```