DEITEL® DEVELOPER SERIES

python™

Python®

for Programmers

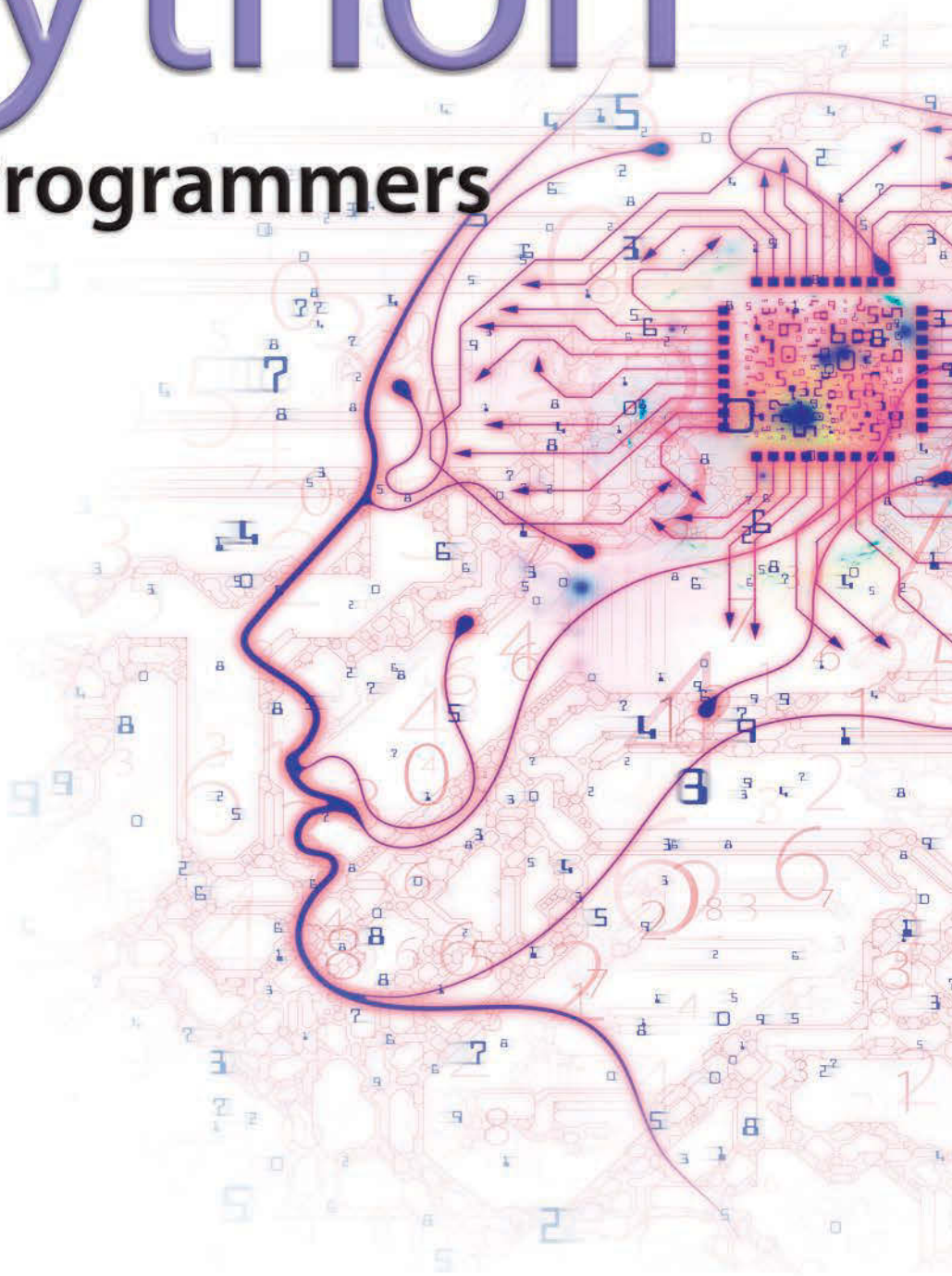
with introductory AI case studies

- Natural Language Processing
- Data Mining Twitter®
- IBM® Watson™
- Machine Learning with scikit-learn®
- Deep Learning with Keras
- Big Data with Hadoop®, Spark™, NoSQL and the Cloud
- Internet of Things (IoT)
- Python Standard Library
- Data Science Libraries: NumPy, Pandas, SciPy, NLTK, TextBlob, Tweepy, Matplotlib, Seaborn, Folium and more

PAUL DEITEL • HARVEY DEITEL

Python®

for Programmers



```
1  # RollDieDynamic.py
2  """Dynamically graphing frequencies of die rolls."""
3  from matplotlib import animation
4  import matplotlib.pyplot as plt
5  import random
6  import seaborn as sns
7  import sys
8
```

Function update

Lines 9–27 define the `update` function that `FuncAnimation` calls once per animation frame. This function must provide at least one argument. Lines 9–10 show the beginning of the function definition. The parameters are:

- `frame_number`—The next value from `FuncAnimation`'s `frames` argument, which we'll discuss momentarily. Though `FuncAnimation` requires the `update` function to have this parameter, we do not use it in this `update` function.
- `rolls`—The number of die rolls per animation frame.
- `faces`—The die face values used as labels along the graph's *x*-axis.
- `frequencies`—The list in which we summarize the die frequencies.

We discuss the rest of the function's body in the next several subsections.

```
9  def update(frame_number, rolls, faces, frequencies):
10     """Configures bar plot contents for each animation frame."""
```

Function update: Rolling the Die and Updating the frequencies List

Lines 12–13 roll the die `rolls` times and increment the appropriate `frequencies` element for each roll. Note that we subtract `1` from the die value (`1` through `6`) before incrementing the corresponding `frequencies` element—as you'll see, `frequencies` is a six-element list (defined in line 36), so its indices are `0` through `5`.

```
11     # roll die and update frequencies
12     for i in range(rolls):
13         frequencies[random.randrange(1, 7) - 1] += 1
14
```

Function update: Configuring the Bar Plot and Text

Line 16 in function `update` calls the `matplotlib.pyplot` module's **`cla`** (clear axes) function to remove the existing bar plot elements before drawing new ones for the current animation frame. We discussed the code in lines 17–27 in the previous chapter's Intro to Data Science section. Lines 17–20 create the bars, set the bar plot's title, set the *x*- and *y*-axis labels and scale the plot to make room for the frequency and percentage text above each bar. Lines 23–27 display the frequency and percentage text.

```
15     # reconfigure plot for updated die frequencies
16     plt.cla()  # clear old contents contents of current Figure
17     axes = sns.barplot(faces, frequencies, palette='bright')  # new bars
18     axes.set_title(f'Die Frequencies for {sum(frequencies):,} Rolls')
19     axes.set(xlabel='Die Value', ylabel='Frequency')
20     axes.set_ylim(top=max(frequencies) * 1.10)  # scale y-axis by 10%
21
```

```
22      # display frequency & percentage above each patch (bar)
23      for bar, frequency in zip(axes.patches, frequencies):
24          text_x = bar.get_x() + bar.get_width() / 2.0
25          text_y = bar.get_height()
26          text = f'{frequency:,}\n{frequency / sum(frequencies):.3%}'
27          axes.text(text_x, text_y, text, ha='center', va='bottom')
28
```

Variables Used to Configure the Graph and Maintain State

Lines 30 and 31 use the `sys` module's `argv` list to get the script's command-line arguments. Line 33 specifies the Seaborn `'whitegrid'` style. Line 34 calls the `matplotlib.pyplot` module's **`figure`** function to get the `Figure` object in which `FuncAnimation` displays the animation. The function's argument is the window's title. As you'll soon see, this is one of `FuncAnimation`'s required arguments. Line 35 creates a list containing the die face values 1–6 to display on the plot's *x*-axis. Line 36 creates the six-element `frequencies` list with each element initialized to `0`—we update this list's counts with each die roll.

```
29  # read command-line arguments for number of frames and rolls per frame
30  number_of_frames = int(sys.argv[1])
31  rolls_per_frame = int(sys.argv[2])
32
33  sns.set_style('whitegrid')  # white background with gray grid lines
34  figure = plt.figure('Rolling a Six-Sided Die')  # Figure for animation
35  values = list(range(1, 7))  # die faces for display on x-axis
36  frequencies = [0] * 6  # six-element list of die frequencies
37
```

Calling the `animation` Module's `FuncAnimation` Function

Lines 39–41 call the Matplotlib `animation` module's `FuncAnimation` function to update the bar chart dynamically. The function returns an object representing the animation. Though this is not used explicitly, you *must* store the reference to the animation; otherwise, Python immediately terminates the animation and returns its memory to the system.

```
38  # configure and start animation that calls function update
39  die_animation = animation.FuncAnimation(
40      figure, update, repeat=False, frames=number_of_frames, interval=33,
41      fargs=(rolls_per_frame, values, frequencies))
42
43  plt.show()  # display window
```

`FuncAnimation` has two required arguments:

- `figure`—the `Figure` object in which to display the animation, and
- `update`—the function to call once per animation frame.

In this case, we also pass the following optional keyword arguments:

- **`repeat`**—`False` terminates the animation after the specified number of frames. If `True` (the default), when the animation completes it restarts from the beginning.
- **`frames`**—The total number of animation frames, which controls how many times `FunctAnimation` calls `update`. Passing an integer is equivalent to passing a `range`—for example, `600` means `range(600)`. `FuncAnimation` passes one value from this range as the first argument in each call to `update`.

- `interval`—The number of milliseconds (33, in this case) between animation frames (the default is 200). After each call to `update`, `FuncAnimation` waits 33 milliseconds before making the next call.
- `fargs` (short for "function arguments")—A tuple of other arguments to pass to the function you specified in `FuncAnimation`'s second argument. The arguments you specify in the `fargs` tuple correspond to `update`'s parameters `rolls`, `faces` and `frequencies` (line 9).

For a list of `FuncAnimation`'s other optional arguments, see

```
https://matplotlib.org/api/_as_gen/
matplotlib.animation.FuncAnimation.html
```

Finally, line 43 displays the window.

6.5 Wrap-Up

In this chapter, we discussed Python's dictionary and set collections. We said what a dictionary is and presented several examples. We showed the syntax of key–value pairs and showed how to use them to create dictionaries with comma-separated lists of key–value pairs in curly braces, `{}`. You also created dictionaries with dictionary comprehensions.

You used square brackets, `[]`, to retrieve the value corresponding to a key, and to insert and update key–value pairs. You also used the dictionary method `update` to change a key's associated value. You iterated through a dictionary's keys, values and items.

You created sets of unique immutable values. You compared sets with the comparison operators, combined sets with set operators and methods, changed sets' values with the mutable set operations and created sets with set comprehensions. You saw that sets are mutable. Frozensets are immutable, so they can be used as set and frozenset elements.

In the Intro to Data Science section, we continued our visualization introduction by presenting the die-rolling simulation with a *dynamic* bar plot to make the law of large numbers "come alive." In addition, to the Seaborn and Matplotlib features shown in the previous chapter's Intro to Data Science section, we used Matplotlib's `FuncAnimation` function to control a frame-by-frame animation. `FuncAnimation` called a function we defined that specified what to display in each animation frame.

In the next chapter, we discuss array-oriented programming with the popular NumPy library. As you'll see, NumPy's `ndarray` collection can be up to two orders of magnitude faster than performing many of the same operations with Python's built-in lists. This power will come in handy for today's big data applications.

7

Array-Oriented Programming with NumPy

Objectives

In this chapter you'll:

- Learn how arrays differ from lists.
- Use the `numpy` module's high-performance `ndarray`s.
- Compare list and `ndarray` performance with the IPython `%timeit` magic.
- Use `ndarray`s to store and retrieve data efficiently.
- Create and initialize `ndarray`s.
- Refer to individual `ndarray` elements.
- Iterate through `ndarray`s.
- Create and manipulate multidimensional `ndarray`s.
- Perform common `ndarray` manipulations.
- Create and manipulate pandas one-dimensional `Series` and two-dimensional `DataFrame`s.
- Customize `Series` and `DataFrame` indices.
- Calculate basic descriptive statistics for data in a `Series` and a `DataFrame`.
- Customize floating-point number precision in pandas output formatting.

Outline

7.1 Introduction

The **NumPy (Numerical Python)** library first appeared in 2006 and is the preferred Python array implementation. It offers a high-performance, richly functional *n*-dimensional array type called **`ndarray`**, which from this point forward we'll refer to by its synonym, `array`. NumPy is one of the many open-source libraries that the Anaconda Python distribution installs. Operations on `arrays` are up to two orders of magnitude faster than those on lists. In a big-data world in which applications may do massive amounts of processing on vast amounts of array-based data, this performance advantage can be critical. According to `libraries.io`, over 450 Python libraries depend on NumPy. Many popular data science libraries such as Pandas, SciPy (Scientific Python) and Keras (for deep learning) are built on or depend on NumPy.

In this chapter, we explore `array`'s basic capabilities. Lists can have multiple dimensions. You generally process multi-dimensional lists with nested loops or list comprehensions with multiple `for` clauses. A strength of NumPy is "array-oriented programming," which uses functional-style programming with *internal* iteration to make array manipulations concise and straightforward, eliminating the kinds of bugs that can occur with the *external* iteration of explicitly programmed loops.

In this chapter's Intro to Data Science section, we begin our multi-section introduction to the *pandas* library that you'll use in many of the data science case study chapters. Big data applications often need more flexible collections than NumPy's `arrays`—collections that support mixed data types, custom indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use. We'll introduce pandas array-like one-dimensional `Series` and two-dimensional `DataFrames` and begin demonstrating their powerful capabilities. After reading this chapter, you'll be familiar with four array-like collections—lists, `arrays`, `Series` and `DataFrames`. We'll add a fifth—tensors—in the "Deep Learning" chapter.

7.2 Creating `arrays` from Existing Data

The NumPy documentation recommends importing the **`numpy` module** as `np` so that you can access its members with `"np."`:

```
In [1]: import numpy as np
```

The numpy module provides various functions for creating arrays. Here we use the **array** function, which receives as an argument an array or other collection of elements and returns a new array containing the argument's elements. Let's pass a list:

```
In [2]: numbers = np.array([2, 3, 5, 7, 11])
```

The array function copies its argument's contents into the array. Let's look at the type of object that function array returns and display its contents:

```
In [3]: type(numbers)
Out[3]: numpy.ndarray

In [4]: numbers
Out[4]: array([ 2,  3,  5,  7, 11])
```

Note that the *type* is numpy.ndarray, but all arrays are output as "array." When outputting an array, NumPy separates each value from the next with a comma and a space and *right-aligns* all the values using the same field width. It determines the field width based on the value that occupies the *largest* number of character positions. In this case, the value 11 occupies the two character positions, so all the values are formatted in two-character fields. That's why there's a leading space between the [and 2.

Multidimensional Arguments

The array function copies its argument's dimensions. Let's create an array from a two-row-by-three-column list:

```
In [5]: np.array([[1, 2, 3], [4, 5, 6]])
Out[5]:
array([[1, 2, 3],
       [4, 5, 6]])
```

NumPy auto-formats arrays, based on their number of dimensions, aligning the columns within each row.

7.3 array Attributes

An array object provides **attributes** that enable you to discover information about its structure and contents. In this section we'll use the following arrays:

```
In [1]: import numpy as np

In [2]: integers = np.array([[1, 2, 3], [4, 5, 6]])

In [3]: integers
Out[3]:
array([[1, 2, 3],
       [4, 5, 6]])

In [4]: floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])

In [5]: floats
Out[5]: array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

NumPy does not display trailing 0s to the right of the decimal point in floating-point values.