2.8 Problems

VL

Problem 2.1 WQ Write a program in a single script that meets the following requirements:

a. It imports the standard library `random` module.
b. It defines a function `rand_sub()` that defines a list of grammatical subjects (e.g., Jim, I, you, skeletons, a tiger, etc.) and returns a random subject; consider using `random.choice()` function.
c. It defines a function `rand_verb()` that defines a list of verbs in past tense (e.g., opened, smashed, ate, became, etc.) and returns a random verb.
d. It defines a function `rand_obj()` that defines a list of grammatical objects (e.g., the closet, her, crumbs, organs) and returns a random object.
e. It defines a function `rand_sen()` that returns a random subject-verb-object sentence as a string beginning with a capital letter and ending with a period.
f. It defines a function `rand_par()` that returns a random paragraph as a string composed of 3 to 5 sentences (the number of sentences should be random—consider using the `random.randint(a, b)` function that generates an `int` between `a` and `b`, inclusively). Sentences should be separated by a space `" "` character.
g. It calls `rand_par()` three times and prints the results.

Problem 2.2 SK Rewrite the program from problem 2.1 such that it meets the following requirements:

a. It defines the functions in a separate module with the file name `rand_speech_parts.py`.
b. Instead of defining the lists of subjects, verbs, and objects inside the functions, it assigns a variable to each list in the module's global namespace and accesses them from within the functions. Why is this preferable?
c. It imports the module into the main script.
d. It print three random paragraphs, as before.

Problem 2.3 YE Write a program in a single script that meets the following requirements:

a. It imports the standard library `random` module.
b. It defines a function `rand_step(x, d, ymax, wrap=True)` that returns a `float` that is the sum of `x` and a uniformly distributed random `float` between `-d` and `d`. Consider using the `random.uniform(a, b)` function that

returns a random `float` between `a` and `b`. If `wrap` is `True`, it maps a stepped value `y > ymax` to `y - ymax` and a stepped value `y < 0` to `ymax + y`. If `wrap` is `False`, it maps a stepped value `y > ymax` to `ymax` and a stepped value `y < 0` to `0`.

c. It defines a function `rand_steps(x0, d, ymax, n, wrap=True)` that returns a `list` of `n` `float`s that are sequentially stepped from `x0`. It passes `wrap` to its call to `rand_step()`.

d. It defines a function `print_slider(k, x)` that prints `k` characters, all of which are `-` except that which has index closest to `x`, for which it prints `|`. For instance, `print_slider(17, 6.8)` should print

```
-------|---------
```

Consider using the built-in `round()` function.

e. It defines a function `rand_sliders(n, k, x0=None, d=3, wrap=True)` that prints `n` random sliders of `k` characters and max step `d` starting at the index closest to `x0`, if provided, and otherwise at the index closest `k/2`.

f. It prints 25 random wrapped sliders of 44 characters with the default step range and starting point 2.

g. It prints 20 random nonwrapped sliders of 44 characters with the step range 5 and starting point 42.

Problem 2.4 8G Rewrite the program from problem 2.3 such that it meets the following requirements:

a. It defines the functions in a separate module with the file name `rand_sliding.py`.

b. It imports the module into the main script.

c. It prints 25 random wrapped sliders of 44 characters with the default step range and starting point 42.

d. It prints 20 random nonwrapped sliders of 44 characters with the step range 5 and starting point 2.

Problem 2.5 UZ Begin with the `Screwdriver`, `Screw`, and `SetScrew` class definitions of section 2.5. Add the following features:

- Improve the `Screwdriver.drive()` method to check that its head matches the screw head and raise a `TypeError` exception if they do not
- Improve the `Screw` class by adding instance attributes `pitch` that stores the thread pitch in mm and `depth` that stores the depth of the screw in its hole

- Improve the `Screw.turn()` method to mutate the `depth` based on the angle it is turned, its handing, and its thread pitch[7]
- Create a subclass `MetricScrew` from the base class `Screw` with the additional *class* data attribute `kind = "Metric"`

Test the new features of the `Screwdriver`, `Screw`, and `MetricScrew` classes with the following steps:

a. Create an instance `ms1` of `MetricScrew` with right-handedness, a flat head, initial angle 0 rad, and thread pitch 2 mm (corresponding to an M14 metric screw)
b. Create an instance `sd1` of `Screwdriver` with a flat head
c. Turn the `ms1` screw 5 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
d. Turn the `ms1` screw 3 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms1`
e. Create an instance `ms2` of `MetricScrew` that is the same as `ms1`, but with *left*-handedness
f. Turn the `ms2` screw 4 complete *counterclockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
g. Turn the `ms2` screw 2 complete *clockwise* revolutions with the `sd1` screwdriver and print the resulting angle and depth of `ms2`
h. Create an instance `sd2` of `Screwdriver` with a hex head and try to turn the `sd1` screw and catch and print the exception

Problem 2.6 VX Improve the bubble sort algorithm of algorithm 1 by adding a test that can return the list if it is sorted before completing all the loops. Implement the improved bubble sort algorithm in a program that it meets the following requirements:

a. It defines a function `bubble_sort(l: list) -> list` that implements the bubble sort algorithm.
b. It demonstrates the `bubble_sort()` function works on three different lists of numbers.
c. It demonstrates that the early return functionality, in fact, saves us from making extra passes through the list.

Problem 2.7 YS **Preprogramming work**: In this problem, *before* writing the program specified, (1) draw a functional design method diagram (see section 2.7.1) and (2) write a pseudocode for each function (see section 2.7.2).

7. A right-handed screw with thread pitch p (mm), turned clockwise an angle α (rad), advances forward $\ell = p\alpha/(2\pi)$ mm. A full turn (i.e., $\alpha = 2\pi$) advances the screw $\ell = p$ mm. Treat clockwise turns as positive angles.

Restrictions: In this problem, most of the functions you will write already exist in the standard libary module `statistics`. You may *not* use this module for this problem, but you may use others, such as the `math` module. You may also use list methods such as `sort()`. Furthermore, you may not use any external packages.

Programming: Write a program in a single script that meets the following requirements:

a. It defines a function `stats(x: list) -> dict` that computes the following basic statistics for input list `x` of real numbers:

   i. The sample mean; for a list x of n values, the sample mean m is

$$m(x) = \frac{1}{n} \sum_{i=0}^{n-1} x_i.$$

   ii. The sample variance; the sample variance s^2 is

$$s^2(x) = \frac{1}{n-1} \sum_{i=0}^{n-1} \left(x_i - m(x)\right)^2 .$$

   iii. The sample standard deviation; the sample standard deviation s is

$$s(x) = \sqrt{s^2(x)}.$$

   iv. The median; the median M of a *sorted* list x of n numbers is value of the list at index $i_M = (n-1)/2$ (i.e., the middle index); more precisely,

$$M(x) = \begin{cases} x_{i_M} & i_M \text{ is an integer} \\ \frac{1}{2}\left(x_{\lfloor i_M \rfloor} + x_{\lceil i_M \rceil}\right) & \text{otherwise} \end{cases}$$

   where $\lfloor \cdot \rfloor$ is the floor function that rounds down and $\lceil \cdot \rceil$ is the ceiling function that rounds up. So in the case that there is no middle index, the mode is the mean of the two middle values.

   The `stats()` function should return a `dict` with the keys `"mean"`, `"var"`, `"std"`, and `"median"` correspond to values for the computed sample mean, variance, standard deviation, and median.

b. It demonstrates the `stats()` function works on three different lists of numbers.

3 Numerical Analysis I: Representations, Input and Output, and Graphics

R3

Engineering design is usually heavily supported by numerical calculations. One of the first and enduring uses of computers is to automatically perform these calculations for engineers; in fact, the first "computers" were humans who performed numerical calculations by hand, as shown in figure 3.1.

Figure 3.1. A "computer room" at the NACA (precursor to NASA) high-speed flight station in 1949 (NASA 2002).

We call engineering numerical calculations **numerical analysis**. Many programming languages and software packages have been used for numerical analysis, but by far the most popular these days are MATLAB and Python. Python's built-in data types (e.g., `list`) and functions (e.g., `sum`) can be used directly for numerical analysis; however, for most engineering problems it is advantageous to use the ubiquitous package NumPy (Harris et al. 2020). The primary reasons this is preferred are that NumPy provides data types, functions, and methods optimized for numerical calculations, which go far beyond Python's built-in modules. In the first several sections of this chapter, we will explore NumPy's data types (most notably the array) and some of its basic functions and methods.

The numerical data represented in NumPy often originates as data from outside the program (e.g., from sensor data gathered via an experiment). Stored in files of various formats, the data must be read from computer memory[1] into the program. This is the most common kind of a program's **inputs**. On the other end, a program can have **outputs**, frequently data files written to computer memory. In this chapter, we will learn how to load input data from files and write output data to files.

Another important kind of program output is a **graphic**—usually a graph, a plot, or a chart. A graphic is often a very important result of a numerical analysis, data visualization being a key component of engineering decision making. In this chapter, we will learn how to use the Python package Matplotlib (Hunter 2007) to generate graphics from data.

3.1 Arrays

WA

NumPy arrays are ubiquitous for representing numerical data. Like lists, arrays are mutable and can represent collections of objects. Unlike for lists, the elements of an array must all be of the same (typically numeric) type. In this section, we learn how to create and manipulate basic arrays. Throughout this book, we will assume that the NumPy package is loaded with the following statement:

```
import numpy as np
```

3.1.1 Creating Arrays

To construct a basic array (i.e., class `np.ndarray`), we often use the function `np.array()`. Although many types of objects can be passed, a list will often do, as follows:

```
x = np.array([0.29, 0.55, -0.31, -0.84, 0.97])
```

The `shape` attribute of the `np.ndarray` object is an integer tuple representing the size (i.e., length) of each of its **dimensions**. For instance, the shape of the 1-dimensional (1D) array of five elements given the name `x` above is printed with

1. The program typically reads a file stored in "secondary" (i.e., long-term) memory and loads it into "main" memory, which is faster to access for calculations. Similarly, when a program writes to a file, it stores data that is in main memory in secondary memory.