Chapter 2 - Some notes:

Reference: *Python for Everyone, 2nd Edition, Cay S. Horstmann, Rance D. Necaise, Wiley, 2016.*

Common Error 2.1

Using Undefined Variables

A variable must be created and initialized before it can be used for the first time. For example, a program starting with the following sequence of statements would not be legal:

```
canVolume = 12 * literPerOunce   # Error: literPerOunce has not yet been created.
literPerOunce = 0.0296
```

In your program, the statements are executed in order. When the first statement is executed by the virtual machine, it does not know that `literPerOunce` will be created in the next line, and it reports an "undefined name" error. The remedy is to reorder the statements so that each variable is created and initialized before it is used.

Programming Tip 2.1

Choose Descriptive Variable Names

We could have saved ourselves a lot of typing by using shorter variable names, as in

```
cv = 0.355
```

Compare this declaration with the one that we actually used, though. Which one is easier to read? There is no comparison. Just reading `canVolume` is a lot less trouble than reading `cv` and then *figuring out* it must mean "can volume".

This is particularly important when programs are written by more than one person. It may be obvious to *you* that `cv` stands for can volume and not current velocity, but will it be obvious to the person who needs to update your code years later? For that matter, will you remember yourself what `cv` means when you look at the code three months from now?

Programming Tip 2.2

Do Not Use Magic Numbers

A **magic number** is a numeric constant that appears in your code without explanation. For example,

```
totalVolume = bottles * 2
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```
BOTTLE_VOLUME = 2.0
totalVolume = bottles * BOTTLE_VOLUME
```

© FinnBrandt/Stockphoto.

We prefer programs that are easy to understand over those that appear to work by magic.

There is another reason for using named constants. Suppose circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it meant a bottle volume or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone.

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
DAYS_PER_YEAR = 365
```

Common Error 2.2

Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that phenomenon yourself with manual calculations. If you calculate 1/3 to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, using only digits 0 and 1. As with decimal numbers, you can get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect.

Here is an example:

```
price = 4.35
quantity = 100
total = price * quantity    # Should be 100 * 4.35 = 435
print(total)    # Prints 434.99999999999994
```

In the binary system, there is no exact representation for 4.35, just as there is no exact representation for 1/3 in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.

You can deal with roundoff errors by rounding to the nearest integer or by displaying a fixed number of digits after the decimal separator (see Section 2.5.3).

Common Error 2.3

Unbalanced Parentheses

Consider the expression

```
((a + b) * t / 2 * (1 - t)
```

What is wrong with it? Count the parentheses. There are three (and two). The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

```
(a + b) * t) / (2 * (1 - t)
```

This expression has three (and three), but it still is not correct. In the middle of the expression,

```
(a + b) * t) / (2 * (1 - t)
          ↑
```

there is only one (but two), which is an error. At any point in an expression, the count of (must be greater than or equal to the count of), and at the end of the expression the two counts must be the same.

© CrokoStockphoto.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

```
(a + b) * t) / (2 * (1 - t)
1     0   -1
```

and you would find the error.

simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

```
(a + b) * t) / (2 * (1 - t)
1     0   -1
```

and you would find the error.

Programming Tip 2.3

Use Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) / (2 * a)
```

than

```
x1=(-b+sqrt(b**2-4*a*c))/(2*a)
```

Simply put spaces around all operators (`+ - * / % =`, and so on). However, don't put a space after a *unary* minus: a – used to negate a single quantity, such as `-b`. That way, it can be easily distinguished from a *binary* minus, as in `a - b`.

It is customary not to put a space after a function name. That is, write `sqrt(x)` and not `sqrt (x)`.

Special Topic 2.1

Other Ways to Import Modules

Python provides several different ways to import functions from a module into your program. You can import multiple functions from the same module like this:

```
from math import sqrt, sin, cos
```

You can also import the entire contents of a module into your program:

```
from math import *
```

Alternatively, you can import the module with the statement

```
import math
```

With this form of the `import` statement, you need to add the module name and a period before each function call, like this:

```
y = math.sqrt(x)
```

Some programmers prefer this style because it makes it very explicit to which module a particular function belongs.

Special Topic 2.2

Combining Assignment and Arithmetic

In Python, you can combine arithmetic and assignment. For example, the instruction

```
total += cans
```

is a shortcut for

```
total = total + cans
```

Similarly,

```
total *= 2
```

is another way of writing

```
total = total * 2
```

Many programmers find this a convenient shortcut especially when incrementing or decrementing by 1:

```
count += 1
```

If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book.

Special Topic 2.3

Line Joining

If you have an expression that is too long to fit on a single line, you can continue it on another line *provided the line break occurs inside parentheses*. For example,

```
x1 = ((-b + sqrt(b ** 2 - 4 * a * c))
   / (2 * a))    # Ok
```

However, if you omit the outermost parentheses, you get an error:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c))
   / (2 * a)    # Error
```

The first line is a complete statement, which the Python interpreter processes. The next line, `/ (2 * a)`, makes no sense by itself.

There is a second form of joining long lines. If the *last* character of a line is a backslash, the line is joined with the one following it:

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) \
   / (2 * a)    # Ok
```

You must be very careful not to put any spaces or tabs after the backslash. In this book, we only use the first form of line joining.

Special Topic 2.4

Character Values

A character is stored internally as an integer value. The specific value used for a given character is based on a standard set of codes. You can find the values of the characters that are used in Western European languages in Appendix D. For example, if you look up the value for the character `"H"`, you can see that it is actually encoded as the number 72.

Python provides two functions related to character encodings. The `ord` function returns the number used to represent a given character. The `chr` function returns the character associated with a given code. For example,

```
print("The letter H has a code of", ord("H"))
print("Code 97 represents the character", chr(97))
```

produces the following output

```
The letter H has a code of 72
Code 97 represents the character a
```

Special Topic 2.5

Escape Sequences

Sometimes you may need to include both single and double quotes in a literal string. For example, to include double quotes around the word `Welcome` in the literal string "You're Welcome", precede the quotation marks with a backslash (\), like this:

```
"You're \"Welcome\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence `\"` is called an **escape sequence**.

To include a backslash in a string, use the escape sequence `\\`, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is `\n`, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
print("*\n**\n***")
```

prints the characters

```
*
**
***
```

on three separate lines.

CHAPTER SUMMARY

Declare variables with appropriate names and types.

- A variable is a storage location with a name.
- An assignment statement stores a value in a variable.
- A variable is created the first time it is assigned a value.
- Assigning a value to an existing variable replaces the previously stored value.
- The assignment operator = does *not* denote mathematical equality.
- The data type of a value specifies how the value is stored in the computer and what operations can be performed on the value.

- Integers are whole numbers without a fractional part.
- Floating-point numbers contain a fractional part.

- By convention, variable names should start with a lowercase letter.
- Use constants for values that should remain unchanged throughout your program.
- Use comments to add explanations for humans who read your code. The interpreter ignores comments.

Write arithmetic expressions in Python.

- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.
- The // operator computes floor division in which the remainder is discarded.
- The % operator computes the remainder of a floor division.
- A function can return a value that can be used as if it were a literal value.
- Python has a standard library that provides functions and data types for your code.
- A library module must be imported into your program before it can be used.

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

Write programs that process strings.

- Strings are sequences of characters.
- A string literal denotes a particular string.

- The `len` function returns the number of characters in a string.
- Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.

- A string can be repeated using the * operator.
- The `str` function converts an integer or floating-point value to a string.

- The `int` and `float` functions convert a string containing a number to the numerical value.
- String positions are counted starting with 0.

Write programs that read user input and print formatted output.

- Use the `input` function to read keyboard input.
- To read an integer or floating-point value, use the `input` function followed by the `int` or `float` function.
- Use the string format operator to specify how values should be formatted.

Make simple graphical drawings.

- A graphics window is used for creating graphical drawings.
- Geometric shapes and text are drawn on a canvas that is contained in a graphics window.
- The canvas has methods for drawing lines, rectangles, and other shapes.
- The canvas stores the current drawing parameters used to draw shapes and text.
- Colors can be specified by name or by their red, green, and blue components.

Reference: *Python for Everyone, 2nd Edition, Cay S. Horstmann, Rance D. Necaise, Wiley, 2016.*