Chapter 3

Input and Type Conversion

It is hard to imagine a useful program that doesn't produce some form of output. From the last two chapters we know how to generate text output in Python using `print()`,[1] but the output produced by a program doesn't have to be text. Instead, it can be an image sent to your screen or written to a file; it can be an audio signal sent to your speakers; and it can even be data generated solely to pass along to another program (without needing to be sent to your screen or a file).

In general, programs not only generate output, they also work with input. This input may come from the user via a keyboard or mouse, from a file, or from some other source such as another program or a network connection. In this chapter we implement programs that perform both *input and output* (often abbreviated as I/O) but restrict ourselves to obtaining input only from a keyboard.

3.1 Obtaining Input: `input()`

The built-in function `input()` takes a string argument. This string is used as a prompt. When the `input()` function is called, the prompt is printed and the program waits for the user to provide input via the keyboard. The user's input is sent back to the program once the user types return. `input()` *always returns the user's input as a string*. This is true even if we are interested in obtaining a numeric quantity (we will see how to convert strings to numeric values in the next section).

Before demonstrating the use of the `input()` function, let's expand on what we mean when we say a function *returns* a value. As we will see, functions can appear in expressions. When Python encounters a function in an expression, it evaluates (calls) that function. If a function appears in an expression, it will almost certainly *return* some form of data.[2] As it evaluates the expression, Python will replace the function with whatever the function returned. For the `input()` function, after it has been called, it is as though it disappears and is replaced by whatever string the function returns.

Listing 3.1 demonstrates the behavior of the `input()` function. Here the goal is to obtain

From the file: `input.tex`

[1] For now, our output only goes to a screen. In a later chapter we will learn how to write output to a file.

[2] Not all functions return data. The `print()` function is an example of a function that does not return data—although it generates output, it does not produce data that can be used in an expression. This is considered in greater detail in Chap. 4.

a user's name and age. Based on this information we want to print a personalized greeting and determine how many multiples of 12 are in the user's age (the Chinese zodiac uses a 12-year cycle).

Listing 3.1 Demonstration of the use of the `input()` function to obtain input from the keyboard. The `input()` function always returns a string.

```
1  >>> name = input("Enter your name: ") # Prompt for and obtain name.
2  Enter your name: Ishmael
3  >>> # Greet user.  However, the following produces an undesired space.
4  >>> print("Greetings", name, "!")
5  Greetings Ishmael !
6  >>> # Remove undesired spaces using the sep optional argument.
7  >>> print("Greetings ", name, "!", sep="")
8  Greetings Ishmael!
9  >>> age = input("Enter your age: ")    # Prompt for and obtain age.
10 Enter your age: 37
11 >>> # Attempt to calculate the number of 12-year cycles of the
12 >>> # Chinese zodiac the user has lived.
13 >>> chinese_zodiac_cycles = age // 12
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 TypeError: unsupported operand type(s) for //: 'str' and 'int'
17 >>> age        # Check age.  Looks like a number but actually a string.
18 '37'
19 >>> type(age) # Explicitly check age's type.
20 <class 'str'>
```

In line 1 the `input()` function appears to the right of the assignment operator. As part of Python's evaluation of the expression to the right of the assignment operator, it invokes the `input()` function. `input()`'s string argument is printed as shown on line 2. After this appears, the program waits for the user's response. We see, also in line 2, that the user responds with `Ishmael`. After the user types return, `input()` returns the user's response as a string. So, in this particular example the right side of line 1 ultimately evaluates to the string `Ishmael` which is assigned to the variable `name`.

In line 4 a `print()` statement is used to greet the user using a combination of two string literals and the user's name. As shown on line 5, the output from this statement is less than ideal in that it contains a space between the name and the exclamation point. We can remove this using the optional parameter `sep` as shown in lines 7 and 8.

In line 9 the user is prompted to enter his or her age. The response, shown in line 10, is `37` and this is assigned to the variable `age`. The goal is next to calculate the multiples of 12 in the user's age. In line 13 an attempt it made to use floor division to divide `age` by `12`. This produces a `TypeError` exception as shown in lines 14 through 16. Looking more closely at line 16 we see that Python is complaining about operands that are a `str` and an `int` when doing floor division. This shows us that, even though we wanted a numeric value for the age, at this point the variable `age` points to a string and thus `age` can only be used in operations that are valid for a string. This

leads us to the subject of the next section which shows a couple of the ways data of one type can be converted to another type.

Before turning to the next section, however, it is worth mentioning now that if a user simply types return when prompted for input, the `input()` function will return the empty string. Later we will exploit this to determine when a user has finished entering data.

3.2 Explicit Type Conversion: `int()`, `float()`, and `str()`

We have seen that there are instances in which data of one type are implicitly converted to another type. For example, when adding an integer and a `float`, the integer is implicitly converted to a `float` and the result is a `float`. This might lead us to ask: What happens if we try to add a string and an integer or perhaps multiply a string and a `float`? Our intuition probably leads us to guess that such operations will produce an error if the string doesn't look like a number, but the answer isn't necessarily obvious if we have a string such as `"1492"` or `"1.414"`. We'll try a few of these types of operations.

Listing 3.2 illustrates a couple of attempts to use strings in arithmetic operations.

Listing 3.2 Attempts to add a string and an integer and to multiply a string and a `float`. Both attempts result in errors.[3]

```
1 >>> "1492" + 520
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 TypeError: cannot convert 'int' object to str implicitly
5 >>> "1.414" * 1.414
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8 TypeError: cannot multiply sequence by non-int of type 'float'
```

Line 1 attempts to add the string `"1492"` and the integer `520`. This attempt fails and produces a `TypeError`. However, note carefully the error message on line 4. It says that it cannot convert an integer to a string *implicitly*. This suggests that perhaps *explicit* conversion is possible, i.e., we have to state more clearly the type of conversion we want to occur. (Also, we want the string to be converted to an integer, not the other way around.)

The attempt to multiply a string and a `float` in line 5 also fails and produces a `TypeError`. Here the error message is a bit more cryptic and, at this stage of our knowledge of Python, doesn't suggest a fix. But, the way to "fix" the statement in line 1 is the way to fix the statement in line 4: we need to explicitly convert one of the operands to a different type to obtain a valid statement.

The `int()` function converts its argument to an integer while the `float()` function converts its argument to a `float`. The arguments to these functions can be either a string or a number (or an expression that returns a string or a number). If the argument is a string, it must "appear" to be an appropriate numeric value. In the case of the `int()` function, the string must look like an

[3]The actual error messages produced using Python 3.2.2 have been change slightly in this example. For the sake of formatting and consistency, "`Can't`" and "`can't`" have been changed to `cannot`.

integer (i.e., it cannot look like a `float` with a decimal point). Listing 3.3 illustrates the behavior of these two functions.

Listing 3.3 Demonstration of the `int()` and `float()` functions.

```
1  >>> int("1492")   # Convert string to an int.
2  1492
3  >>> int("1.414") # String must look like an int to succeed.
4  Traceback (most recent call last):
5    File "<stdin>", line 1, in <module>
6  ValueError: invalid literal for int() with base 10: '1.414'
7  >>> # Explicit conversion of string allows following to succeed.
8  >>> int("1492") + 520
9  2012
10 >>> int(1.414)      # Can convert a float to an int.
11 1
12 >>> int(2.999999)   # Fractional part discarded -- rounding not done.
13 2
14 >>> int(-2.999999) # Same behavior for negative numbers.
15 -2
16 >>> float("1.414") # Conversion of string that looks like a float.
17 1.414
18 >>> float("1.414") * 1.414   # Arithmetic operation now works.
19 1.9993959999999997
20 >>> 1.414 * 1.414   # Result is same if we enter floats directly.
21 1.9993959999999997
```

Line 1 demonstrates that a string that looks like an integer is converted to an integer by `int()`. Line 3 shows that a `ValueError` is produced if the argument of the `int()` function is a string that looks like a `float`. Line 8 shows that, by using explicit conversion, this arithmetic operation now succeeds: The string `"1492"` is converted to an integer and added to the integer `520` to produce the integer result `2012`.

As line 10 illustrates, the `int()` function can also take a numeric argument. Here the argument is a `float`. The integer result on line 11 shows the fractional part has been discarded. The `int()` function does not round to the nearest integer. Rather, as illustrated in lines 12 to 15, it merely discards the fractional part.

It might seem the `int()` function's behavior is inconsistent in that it will not accept a string that looks like a `float` but it will accept an actual `float`. However, if one wants to convert a string that looks like a `float` to an integer, there are really two steps involved. First, the string must be converted to a `float` and then the `float` must be converted to an integer. The `int()` function will not do both of these steps. If a programmer wants this type of conversion, some additional code has to be provided. We will return to this shortly.

The expression in line 16 of Listing 3.3 shows that by using explicit conversion we can now multiply the value represented by the string `"1.414"` and the `float 1.414`. If you spend a moment looking at the result of this multiplication given in line 17, you may notice this result is *not* what you would get if you multiplied these values using most calculators. On a calculator you

are likely to obtain `1.999396`. Is the error a consequence of our having converted a string to a `float`? Lines 18 and 19 answer this. In line 18 the `float` values are entered directly into the expression and the identical result is obtained. The "error" is a consequence of these values being `floats` and, as discussed in Sec. 2.1, the fact that `floats` have finite precision.

With the ability to convert strings to numeric quantities, we can now revisit the code in Listing 3.1. Listing 3.4 demonstrates, in lines 5 and 6, the successful conversion of the user's input, i.e., the user's age, to a numeric value, here an integer.

Listing 3.4 Demonstration of the use of the `input()` function to obtain input from the keyboard. The `input()` function always returns a string.

```
1  >>> name = input("Enter your name: ")
2  Enter your name: Captain Ahab
3  >>> print("Greetings ", name, "!", sep="")
4  Greetings Captain Ahab!
5  >>> age = int(input("Enter your age: ")) # Obtain user's age.
6  Enter your age: 57
7  >>> # Calculate the number of complete cycles of the 12-year
8  >>> # Chinese zodiac the user has lived.
9  >>> chinese_zodiac_cycles = age // 12
10 >>> print("You have lived", chinese_zodiac_cycles,
11 ...       "complete cycles of the Chinese zodiac.")
12 You have lived 4 complete cycles of the Chinese zodiac.
```

In line 5 the `int()` function is used to convert the string that is returned by the `input()` function to an integer. Note that the construct here is something new: the functions are *nested*, i.e., the `input()` function is inside the `int()` function. Nesting is perfectly acceptable and is done quite frequently. The innermost function is invoked first and whatever it returns serves as an argument for the surrounding function.

Note that the code in Listing 3.4 is not very *robust* in that reasonable input could produce an error. For example, what if `Captain Ahab` enters his age as `57.5` instead of `57`? In this case the `int()` function would fail. One can spend quite a bit of effort trying to ensure that a program is immune to "incorrect" input. However, at this point, writing robust code is not our primary concern, so we typically will not dwell on this issue.

Nevertheless, along these lines, let's return to the point raised above about the inability of the `int()` function to handle a string argument that looks like a `float`. Listing 3.5 shows that if an integer value is ultimately desired, one can first use `float()` to safely convert the string to a `float` and then use `int()` to convert this numeric value to an integer.

Listing 3.5 Intermediate use of the `float()` function to allow entry of strings that appear to be either `floats` or `ints`.

```
1  >>> # Convert a string to a float and then to an int.
2  >>> int(float("1.414"))
3  1
```

```
>>> # float() has no problem with strings that appear to be ints.
>>> float("14")
14.0
>>> int(float("14"))
14
>>> # Desire an integer age but user enters a float, causing an error.
>>> age = int(input("Enter age: "))
Enter age: 57.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '57.5'
>>> # Can use float() to allow fractional ages.
>>> age = int(float(input("Enter your age: ")))
Enter your age: 57.5
>>> age
57
```

In line 1 the `float()` function, which is nested inside the `int()` function, converts the string `"1.414"` to a `float`. The `int()` function converts this to an integer, discarding the fractional part, and the resulting value of `1` is shown on line 2. As shown in lines 5 through 8, the `float()` function can handle string arguments that appear to be integers.

Line 10 of Listing 3.5 uses the same statement as was used in Listing 3.4 to obtain an age. In this example, the user enters `57.5`. Since the `int()` function cannot accept a string that doesn't appear as an integer, an error is produced as shown in lines 12 through 14. (Although line 10 is shown in red, it does not explicitly contain a bug. Rather, the input on line 11 cannot be handled by the statement on line 10. Thus both line 10 and the input on line 11 are shown in red.)

If one wants to allow the user to enter fractional ages, the statement shown in line 16 can be used. Here three nested functions are used. The innermost function, `input()`, returns a string. This string is passed as an argument to the middle function, `float()`, which returns a `float`. This `float` is subsequently the argument for the outermost function, `int()`, which returns an integer. Line 18 shows that, indeed, the variable `age` has been set to an integer.

We've seen that `int()` and `float()` can convert a string to a numeric value. In some sense, the `str()` function is the converse of these functions in that it can convert a numeric value to a string. In fact, all forms of data in Python have a string representation. At this point, the utility of the `str()` function isn't obvious, but in subsequent chapters we will see that it proves useful in a number of situations (e.g., in forming a suitable prompt for the `input()` function which consists of a combination of text and a numeric value as shown in Listing 6.21). For now, in Listing 3.6, we merely demonstrate the behavior of `str()`. Please read the comments in the listing.

Listing 3.6 Demonstration that the `str()` function converts its argument to a string.

```
>>> str(5)                  # Convert int to a string.
'5'
>>> str(1 + 10 + 100)       # Convert int expression to a string.
'111'
>>> str(-12.34)             # Convert float to a string.
```

```
'-12.34'
>>> str("Hello World!")   # str() accepts string arguments.
'Hello World!'
>>> str(divmod(14, 9))    # Convert tuple to a string.
'(1, 5)'
>>> x = 42
>>> str(x)                # Convert int variable to a string.
'42'
```

3.3 Evaluating Strings: `eval()`

Python provides a powerful function called `eval()` that is sometimes used to facilitate obtaining input. However, because of this function's power, one should be cautious about using it in situations where users could potentially cause problems with "inappropriate" input. The `eval()` function takes a string argument and *evaluates* that string as a Python expression, i.e., just as if the programmer had directly entered the expression as code. The function returns the result of that expression. Listing 3.7 demonstrates `eval()`.

Listing 3.7 Basic demonstration of the `eval()` function.

```
1  >>> string = "5 + 12" # Create a string.
2  >>> print(string)     # Print the string.
3  5 + 12
4  >>> eval(string)      # Evaluate the string.
5  17
6  >>> print(string, "=", eval(string))
7  5 + 12 = 17
8  >>> eval("print('Hello World!')") # Can call functions from eval().
9  Hello World!
10 >>> # Using eval() we can accept all kinds of input...
11 >>> age = eval(input("Enter your age: "))
12 Enter your age: 57.5
13 >>> age
14 57.5
15 >>> age = eval(input("Enter your age: "))
16 Enter your age: 57
17 >>> age
18 57
19 >>> age = eval(input("Enter your age: "))
20 Enter your age: 40 + 17 + 0.5
21 >>> age
22 57.5
```

In line 1 a string is created that looks like an expression. In line 2 this string is printed and the output is the same as the string itself. In line 4, the string is the argument to the `eval()` function.

`eval()` evaluates the expression and returns the result as shown in line 5. The `print()` statement in line 6 shows both the string and the result of evaluating the string. Line 8 shows that one can call a function via the string that `eval()` evaluates—in this statement the string contains a `print()` statement.

In line 11 the `input()` function is nested inside the `eval()` function. In this case whatever input the user enters will be evaluated, i.e., the string returned by `input()` will be the argument of `eval()`, and the result will be assigned to the variable `age`. Because the user entered `57.5` in line 12, we see, in lines 13 and 14, that `age` is the `float` value `57.5`. Lines 15 through 18 show what happens when the user enters `57`. In this case `age` is the integer `57`. Then, in lines 19 through 22, we see what happens when the user enters an expression involving arithmetic operations—`eval()` handles it without a problem.

Recall that, as shown in Sec. 2.4, simultaneous assignment can be used if multiple expressions appear to the right of the assignment operator and multiple variables appear to the left of the assignment operator. The expressions and variables must be separated by commas. This ability to do simultaneous assignment can be coupled with the `eval()` function to allow the entry of multiple values on a single line. Listing 3.8 demonstrates this.

Listing 3.8 Demonstration of how `eval()` can be used to obtain multiple values on a single line.

```
1  >>> eval("10, 32")          # String with comma-separated values.
2  (10, 32)
3  >>> x, y = eval("10, 20 + 12") # Use simultaneous assignment.
4  >>> x, y
5  (10, 32)
6  >>> # Prompt for multiple values.  Must separate values with a comma.
7  >>> x, y = eval(input("Enter x and y: "))
8  Enter x and y: 5 * 2, 32
9  >>> x, y
10 (10, 32)
```

When the string in the argument of the `eval()` function in line 1 is evaluated, it is treated as two integer literals separated by a comma. Thus, these two values appear in the output shown in line 2. We can use simultaneous assignment, as shown in line 3, to set the value of two variables when `eval()`'s string argument contains more than one expression.

Lines 7 through 10 show that the user can be prompted for multiple values. The user can respond with a wide range of expressions if these expressions are separated by a comma.

As an example of multiple input on a single line, let us calculate a user's body mass index (BMI). BMI is a function of height and weight. The formula is

$$\text{BMI} = 703\frac{W}{H^2}$$

where the weight W is in pounds and the height H is in inches. The code shown in Listing 3.9 shows how the height and weight can be obtained from the user and how the BMI can be calculated.

Listing 3.9 Obtaining multiple values on a single line and calculating the BMI.

```
1 >>> print("Enter weight and height separated by a comma.")
2 Enter weight and height separated by a comma.
3 >>> weight, height = eval(input("Weight [pounds], Height [inches]: "))
4 Weight [pounds], Height [inches]: 160, 72
5 >>> bmi = 703 * weight / (height * height)
6 >>> print("Your body mass index is", bmi)
7 Your body mass index is 21.6975308642
```

Line 3 is used to obtain both weight and height. The user's response on line 4 will set these both to integers but it wouldn't have mattered if `float` values were entered. The BMI is calculated in line 5. In the denominator `height` is multiplied by itself to obtain the square of the height. However, one could have instead used the power operator, i.e., `height ** 2`. Because `float` division is used, the result will be a `float`. The BMI is displayed using the `print()` statement in line 6. In all likelihood, the user wasn't interested in knowing the BMI to 10 decimal places. In Sec. 9.7 we will see how the output can be formatted more reasonably.

Again, a word of caution about the use of `eval()`: by allowing the user to enter an expression that may contain calls to other functions, the user's input could potentially have some undesired consequences. Thus, if you know you want integer input, it is best to use the `int()` function. If you know you want `float` input, it is best to use the `float()` function. If you really need to allow multiple values in a single line of input, there are better ways to handle it; these will be discussed in Chap. 9.

3.4 Chapter Summary

`input()`: Prompts the user for input with its string argument and returns the string the user enters.

`int()`: Returns the integer form of its argument.

`float()`: Returns the `float` form of its argument.

`eval()`: Returns the result of evaluating its string argument as any Python expression, including arithmetic and numerical expressions.

Functions such as the four listed above can be *nested*. Thus, for example, **`float(input())`** can be used to obtain input in string form which is then converted to a float value.

3.5 Review Questions

1. The following code is executed

   ```
   x = input("Enter x: ")
   ```

   In response to the prompt the user enters

```
-2 * 3 + 5
```

What is the resulting value of `x`?

(a) `-16`

(b) `-1`

(c) `'-2 * 3 + 5'`

(d) This produces an error.

2. The following code is executed

```
y = int(input("Enter x: ")) + 1
```

In response to the prompt the user enters

```
50
```

What is the resulting value of `y`?

(a) `'50 + 1'`

(b) `51`

(c) `50`

(d) This produces an error.

3. The following code is executed

```
y = int(input("Enter x: ") + 1)
```

In response to the prompt the user enters

```
50
```

What is the resulting value of `y`?

(a) `'50 + 1'`

(b) `51`

(c) `50`

(d) This produces an error.

4. The following code is executed

```
x = input("Enter x: ")
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`

(b) `x = -20`

(c) `x = 2 + 3 * -4`

(d) This produces an error.

(e) None of the above.

5. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`

(b) `x = -20`

(c) `x = 2 + 3 * -4`

(d) This produces an error.

(e) None of the above.

6. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5.0
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

7. The following code is executed

```
x = float(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

8. The following code is executed

```
x = eval(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

9. The following code is executed

```
x = input("Enter x: ")
print("x =", x)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 5.0`

(b) `x = 5`

(c) This produces an error.

(d) None of the above.

10. The following code is executed

```
x = int(input("Enter x: "))
print("x =", x + 1.0)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`

(b) `x = 6`

(c) This produces an error.

(d) None of the above.

11. The following code is executed

```
x = float(input("Enter x: "))
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`

(b) `x = 6`

(c) This produces an error.

(d) None of the above.

12. The following code is executed

```
x = eval(input("Enter x: "))
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`

(b) `x = 6`

(c) This produces an error.

(d) None of the above.

13. The following code is executed

```
x = input("Enter x: ")
print("x =", x + 1)
```

In response to the prompt the user enters

```
5
```

What is the output produced by the `print()` statement?

(a) `x = 6.0`

(b) `x = 6`

(c) This produces an error.

(d) None of the above.

14. The following code is executed

```
x = eval(input("Enter x: "))
print("x =", x)
```

In response to the prompt the user enters

```
2 + 3 * -4
```

What is the output produced by the `print()` statement?

(a) `x = -10`

(b) `x = -20`

(c) `x = 2 + 3 * -4`

(d) This produces an error.

(e) None of the above.

15. What is produced by the `print()` statement in the following code?

```
s = "8 / 4 + 4"
print(s, eval(s), sep=" = ")
```

What is the resulting value of `y`?

(a) This produces an error.

(b) `8 / 4 + 4 = 6`

(c) `6.0 = 6.0`

(d) `8 / 4 + 4 = 6.0`

(e) None of the above.

16. True or False: All of the following are acceptable arguments for the `int()` function: `5`, `5.0`, `"5"`, and `"5.0"` (these arguments are an `int`, a `float`, and two `str`s, respectively).

17. True or False: All of the following are acceptable arguments for the `float()` function: `5`, `5.0`, `"5"`, and `"5.0"`.

18. True or False: All of the following are acceptable arguments for the `eval()` function: `5`, `5.0`, `"5"`, and `"5.0"`.

19. True or False: The string `"5.0, 6.0"` is an acceptable argument for the `eval()` function but not for the `float()` function.

ANSWERS: 1) c; 2) b; 3) d, the addition is attempted before conversion to an `int`; 4) c; 5) d; 6) c; 7) a; 8) b; 9) b; 10) a; 11) a; 12) b; 13) c, cannot add string and integer; 14) a; 15) d; 16) False; 17) True; 18) False, the argument must be a string; 19) True.

3.6 Exercises

1. A commonly used method to provide a rough estimate of the right length of snowboard for a rider is to calculate 88 percent of their height (the actual ideal length really depends on a large number of other factors). Write a program that will help people estimate the length of snowboard they should buy. Obtain the user's height in feet and inches (assume these values will be entered as integers) and display the length of snowboard in centimeters to the user. There are 2.54 centimeters in an inch.

   The following demonstrates the proper behavior of the program:

```
Enter your height.
Feet: 5
Inches: 4

Suggested board length: 143.0528 cm
```

2. Newton's Second Law of motion is expressed in the formula $F = m \times a$ where F is force, m is mass, and a is acceleration. Assume that the user knows the mass of an object and the force on that object but wants to obtain the object's acceleration a. Write a program that prompts the user to enter the mass in kilograms (kg) and the force in Newtons (N). The user should enter both values on the same line separated by a comma. Calculate the acceleration using the above formula and display the result to the user.

   The following demonstrates the proper behavior of the program:

```
Enter the mass in kg and the force in N: 55.4, 6.094

The acceleration is 0.11000000000000001
```

3. Write a program that calculates how much it costs to run an appliance per year and over a 10 year period. Have the user enter the cost per kilowatt-hour in cents and then the number of kilowatt-hours used per year. Assume the user will be entering floats. Display the cost to the user in dollars (where the fractional part indicates the fraction of a dollar and does not have to be rounded to the nearest penny).

   The following demonstrates the proper behavior of the program:

```
Enter the cost per kilowatt-hour in cents: 6.54
Enter the number of kilowatt-hours used per year: 789

The annual cost will be:  51.60060000000001
The cost over 10 years will be:  516.0060000000001
```

4. In the word game Mad Libs, people are asked to provide a part of speech, such as a noun, verb, adverb, or adjective. The supplied words are used to fill in the blanks of a preexisting template or replace the same parts of speech in a preexisting sentence. Although we don't yet have the tools to implement a full Mad Libs game, we can implement code that demonstrates how the game works for a single sentence. Consider this sentence from P. G. Wodehouse:

   > Jeeves lugged my purple socks out of the drawer as if he were a vegetarian fishing a caterpillar out of his salad.

   Write a program that will do the following:

   - Print the following template:

     ```
     Jeeves [verb] my [adjective] [noun] out of the [noun]
     as if he were a vegetarian fishing a [noun] out of his
     salad.
     ```

   - Prompt the user for a verb, an adjective, and three nouns.
   - Print the template with the terms in brackets replaced with the words the user provided.

   Use string concatenation (i.e., the combining of strings with the plus sign) as appropriate.

   The following demonstrates the proper behavior of this code

```
Jeeves [verb] my [adjective] [noun] out of the [noun] as if
he were a vegetarian fishing a [noun] out of his salad.

Enter a verb: bounced
Enter an adjective: invisible
Enter a noun: parka
Enter a noun: watermelon
Enter a noun: lion

Jeeves bounced my invisible parka out of the watermelon as if
he were a vegetarian fishing a lion out of his salad.
```