Chapter 2: Data and Expressions

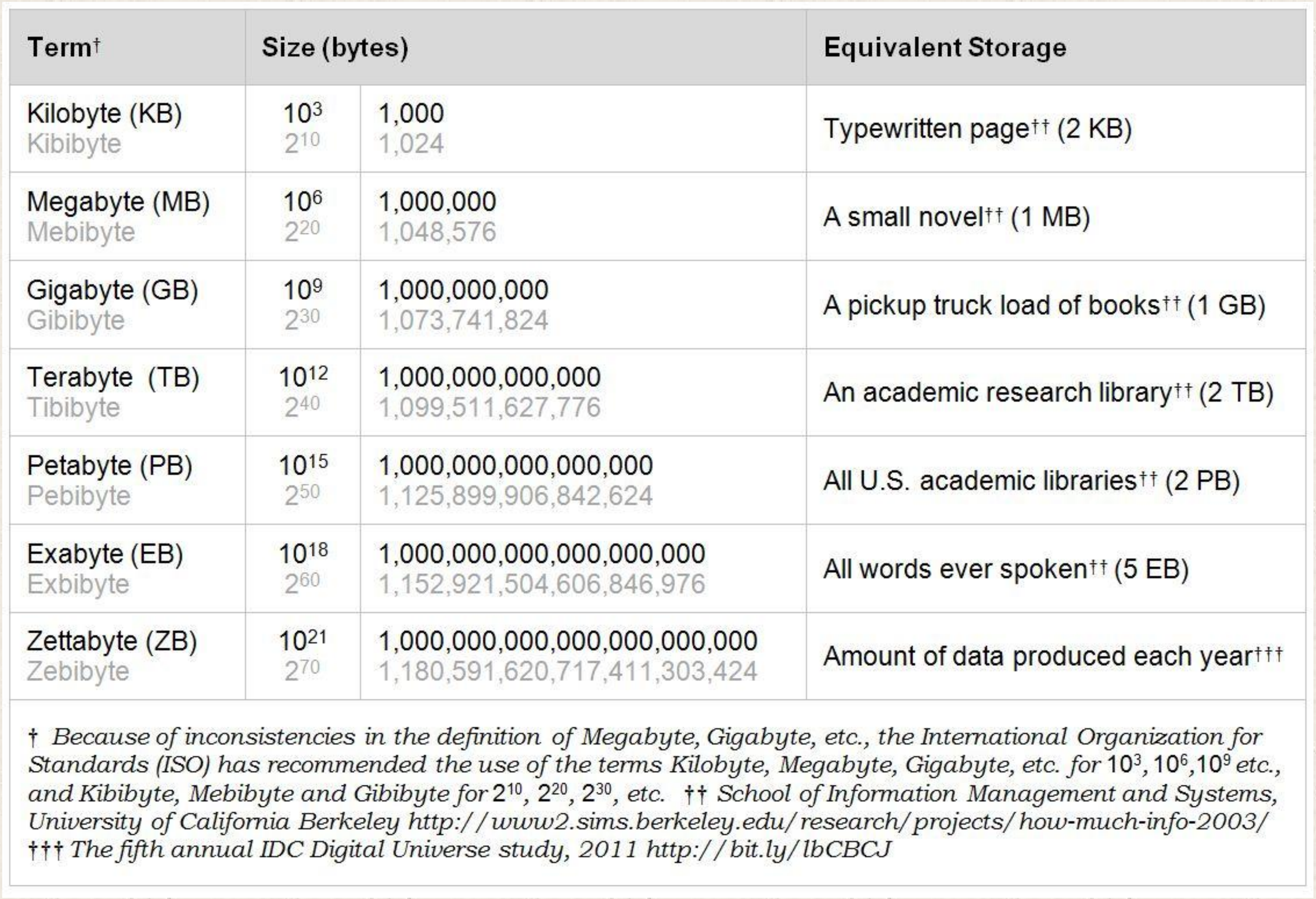
With this chapter, we begin a detailed discussion of the concepts and techniques of computer programming. We start by looking at issues related to the representation, manipulation, and input/output of data—fundamental to all computing.

Motivation

The generation, collection, and analysis of data is a driving force in today's world. The sheer amount of data being created is staggering.

Chain stores generate **terabytes** of customer information, looking for shopping patterns of individuals. Facebook users have created 40 billion photos requiring more than a **petabyte** of storage. A certain radio telescope is expected to generate an **exabyte** of information every four hours. All told, **the current amount of data created each year is estimated to be almost two zettabytes**, **more than doubling every two years**.

In this chapter, we look at how data is represented and operated on in Python.



Term†	Size (bytes)		Equivalent Storage
Kilobyte (KB)	10^3	1,000	Typewritten page†† (2 KB)
Kibibyte	2^{10}	1,024	
Megabyte (MB)	10^6	1,000,000	A small novel†† (1 MB)
Mebibyte	2^{20}	1,048,576	
Gigabyte (GB)	10^9	1,000,000,000	A pickup truck load of books†† (1 GB)
Gibibyte	2^{30}	1,073,741,824	
Terabyte (TB)	10^{12}	1,000,000,000,000	An academic research library†† (2 TB)
Tibibyte	2^{40}	1,099,511,627,776	
Petabyte (PB)	10^{15}	1,000,000,000,000,000	All U.S. academic libraries†† (2 PB)
Pebibyte	2^{50}	1,125,899,906,842,624	
Exabyte (EB)	10^{18}	1,000,000,000,000,000,000	All words ever spoken†† (5 EB)
Exbibyte	2^{60}	1,152,921,504,606,846,976	
Zettabyte (ZB)	10^{21}	1,000,000,000,000,000,000,000	Amount of data produced each year†††
Zebibyte	2^{70}	1,180,591,620,717,411,303,424	

† *Because of inconsistencies in the definition of Megabyte, Gigabyte, etc., the International Organization for Standards (ISO) has recommended the use of the terms Kilobyte, Megabyte, Gigabyte, etc. for* 10^3, 10^6, 10^9 *etc., and Kibibyte, Mebibyte and Gibibyte for* 2^{10}, 2^{20}, 2^{30}, *etc.* †† *School of Information Management and Systems, University of California Berkeley http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/*
††† *The fifth annual IDC Digital Universe study, 2011 http://bit.ly/lbCBCJ*

Literals

To take something literally is to take it at “face value.” The same is true of literals in programming. **A literal is a sequence of one of more characters that stands for itself**, such as the literal 12. We look at numeric literals in Python.

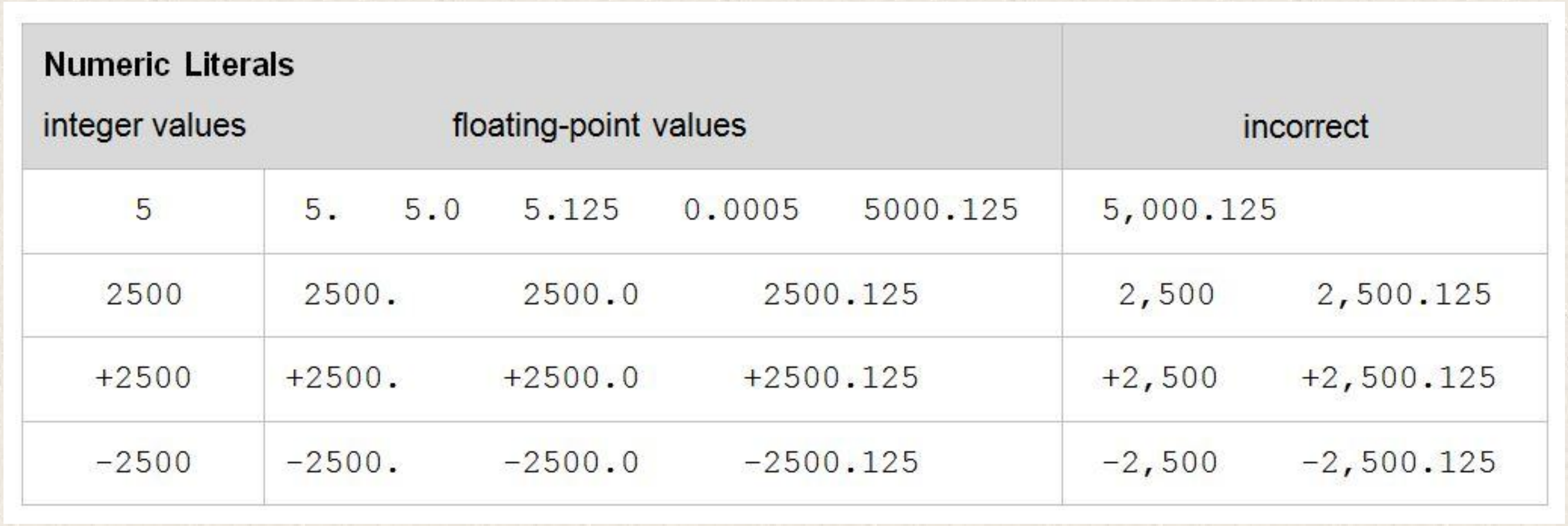
Numeric Literals

A numeric literal **is a literal containing only the digits 0–9, an optional sign character (+ or -), and a possible decimal point**. (The letter **e** is also used in exponential notation, shown next).

If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or " float " (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10).

Commas are never used in numeric literals .

Example Numerical Values



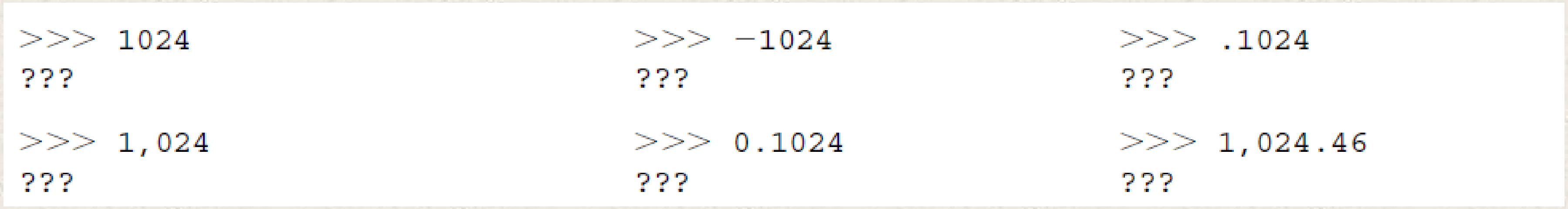
Numeric Literals

integer values	floating-point values	incorrect
5	5. 5.0 5.125 0.0005 5000.125	5,000.125
2500	2500. 2500.0 2500.125	2,500 2,500.125
+2500	+2500. +2500.0 +2500.125	+2,500 +2,500.125
-2500	-2500. -2500.0 -2500.125	-2,500 -2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign is rarely used.

Let’s Try It

Which of the following are valid numeric literals in Python?

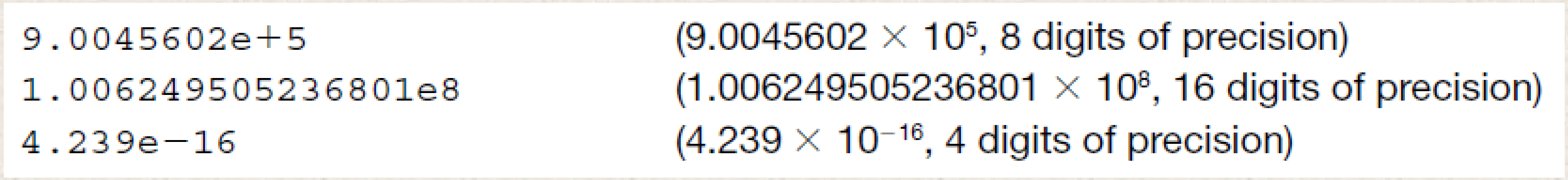


```
>>> 1024                        >>> -1024             >>> .1024
???                             ???                   ???

>>> 1,024                       >>> 0.1024            >>> 1,024.46
???                             ???                   ???
```

Limits of Range in Floating-Point Representation

There is no limit to the size of an integer that can be represented in Python. Floating point values, however, have both a limited ***range*** and a limited ***precision***.

Python uses a double-precision standard format (IEEE 754) providing a range of 10^{-308} to 10^{308} with 16 to 17 digits of precision. To denote such a range of values, floating-points can be represented in scientific notation.



`9.0045602e+5`	(9.0045602×10^5, 8 digits of precision)
`1.006249505236801e8`	($1.006249505236801 \times 10^8$, 16 digits of precision)
`4.239e-16`	(4.239×10^{-16}, 4 digits of precision)

It is important to understand the limitations of floating-point representation. For example, the multiplication of two values may result in ***arithmetic overflow***, **a condition that occurs when a calculated result is too large in magnitude (size) to be represented,**

```
>>> 1.5e200 * 2.0e210
inf
```

This results in the special value **`inf`** ("infinity") rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred.

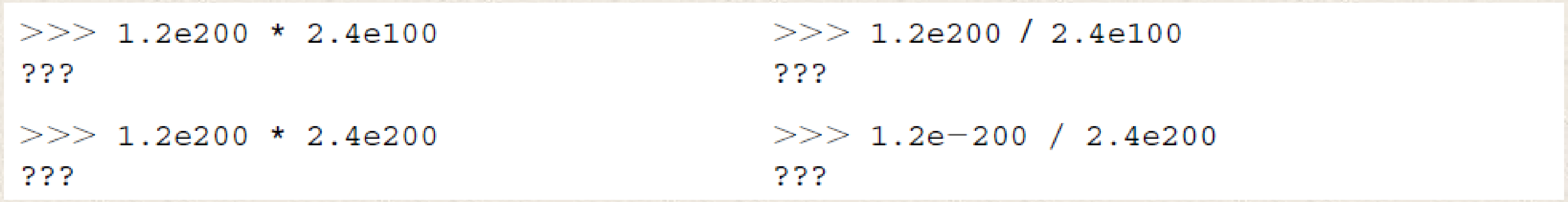
Similarly, the division of two numbers may result in ***arithmetic underflow***, **a condition that occurs when a calculated result is too small in magnitude to be represented,**

```
>>> 1.0e-300 / 1.0e100
0.0
```

This results in 0.0 rather than the arithmetically correct result 1.0e-400, indicating that arithmetic underflow has occurred.

Let’s Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> 1.2e200 * 2.4e100
???

>>> 1.2e200 * 2.4e200
???
```

```
>>> 1.2e200 / 2.4e100
???

>>> 1.2e-200 / 2.4e200
???
```

Limits of Precision in Floating-Point Representation

Arithmetic overflow and arithmetic underflow are easily detected. The loss of precision that can result in a calculated result, however, is a much more subtle issue. For example, 1/3 is equal to the infinitely repeating decimal .33333333 . . ., which also has repeating digits in base two, .010101010. . . . Since any floating-point representation necessarily contains only a finite number of digits, what is stored for many floating-point values is only an *approximation of the true value,* as can be demonstrated in Python,

```
>>> 1/3
.3333333333333333
```

Here, the repeating decimal ends after the 16th digit. Consider also the following,

```
>>> 3 * (1/3)
1.0
```

Given the value of 1/3 above as .3333333333333333, we would expect the result to be .9999999999999999, so what is happening here?

The answer is that **Python displays a rounded result to keep the number of digits displayed manageable**. However, the representation of 1/3 as .3333333333333333 remains the same, as demonstrated by the following,

```
>>> 1/3 + 1/3 + 1/3 + 1/3 + 1/3 1 + 1/3
1.9999999999999998
```

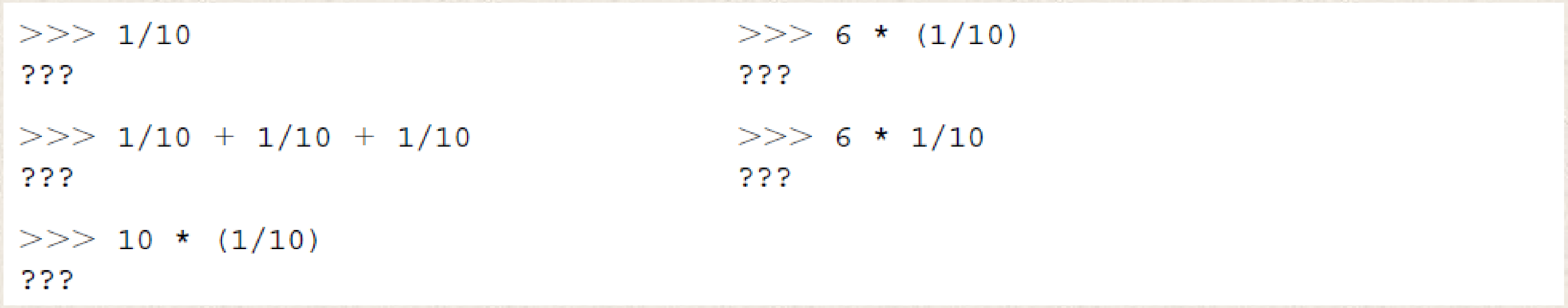
In this case we get a result that reflects the representation of 1/3 as an approximation, since the last digit is 8, and not 9. However, if we use multiplication instead, we again get the rounded value displayed,

```
>>>6 * (1/3)
2.0
```

The bottom line, therefore, is that **no matter how Python chooses to display calculated results, the value stored is limited in both the range of numbers that can be represented and the degree of precision**. For most everyday applications, this slight loss in accuracy is of no practical concern. However, in scientific computing and other applications in which precise calculations are required, this is something that the programmer must be keenly aware of.

Let’s Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> 1/10
???

>>> 1/10 + 1/10 + 1/10
???

>>> 10 * (1/10)
???
```

```
>>> 6 * (1/10)
???

>>> 6 * 1/10
???
```

Built-in Format Function

Because floating-point values may contain an arbitrary number of decimal places, the **built-in format function can be used to produce a numeric *string* version of the value containing a specific number of decimal places,**

```
>>>  12/5                   >>> 5/7
2.4                         0.7142857142857143
>>>  format(12/5, '.2f')    >>> format(5/7, '.2f')
'2.40'                      '0.71'
```

In these examples, **format specifier '`.2f`'** rounds the result to two decimal places of accuracy in the string produced.

For very large (or very small) values '`e`' can be used as a format specifier.

```
>>> format(2 ** 100, '.6e')
'1.267651e+30'
```

Here, the value is formatted in scientific notation, with six decimal places of precision.

Formatted numeric string values are useful when displaying results in which only a certain number of decimal places need to be displayed:

Without use of format specifier:

```
>>> tax = 0.08
>>> print('Your cost: $', (1 + tax) * 12.99)
Your cost: $ 14.029200000000001
```

With use of format specifier:

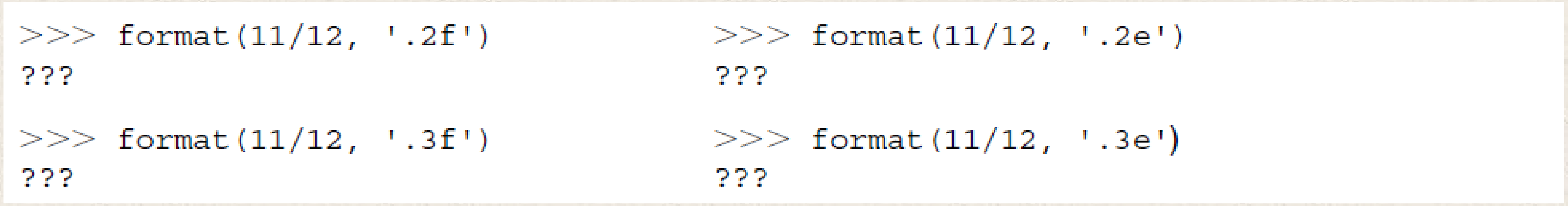
```
>>> print('Your cost: $', format((1 + tax) * 12.99, '.2f'))
Your cost: $ 14.03
```

Finally, a comma in the format specifier adds comma separators to the result.

```
>>> format(13402.25, ',.2f')
13,402.24
```

Let’s Try It

What do each of the following uses of the format function produce?



```
>>> format(11/12, '.2f')          >>> format(11/12, '.2e')
???                               ???

>>> format(11/12, '.3f')          >>> format(11/12, '.3e')
???                               ???
```

String Literals

String literals , or “ strings ,” represent a sequence of characters.

```
'Hello'   'Smith, John'   "Baltimore, Maryland 21210"
```

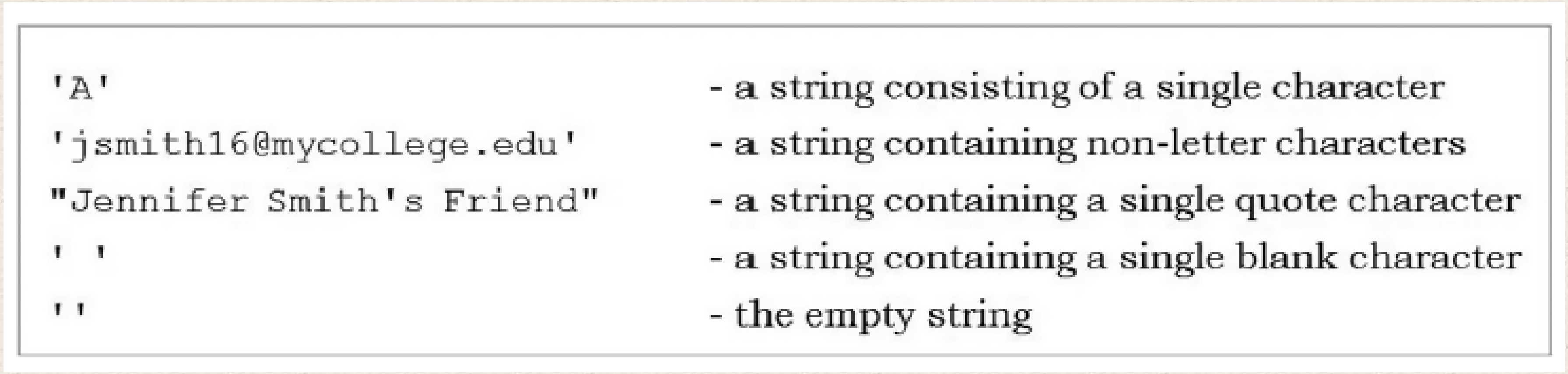
In Python, string literals may be delimited (surrounded) by a matching pair of either single (') or double (") quotes. Strings must be contained all on one line (except when delimited by triple quotes, discussed later).

We have already seen the use of strings in Chapter 1 for displaying screen output,

```
>>> print('Welcome to Python!')
Welcome to Python!
```

A string may contain zero or more characters, including **letters**, **digits**, **special characters**, and **blanks**. A string consisting of only a pair of matching quotes (with nothing in between) is called the **empty string**, which is different from a string containing only blank characters. Both blank strings and the empty string have their uses, as we will see.

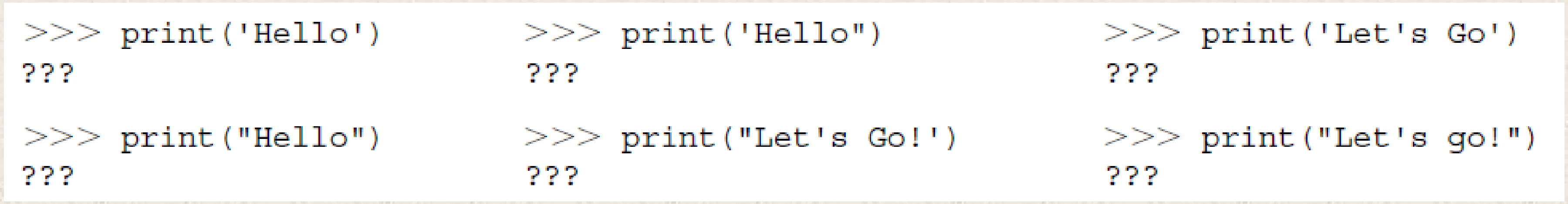
Strings may also contain quote characters as long as different quotes (single vs. double) are used to delimit the string.



`'A'`	- a string consisting of a single character
`'jsmith16@mycollege.edu'`	- a string containing non-letter characters
`"Jennifer Smith's Friend"`	- a string containing a single quote character
`' '`	- a string containing a single blank character
`''`	- the empty string

Let’s Try It

What will be displayed by each of the following?



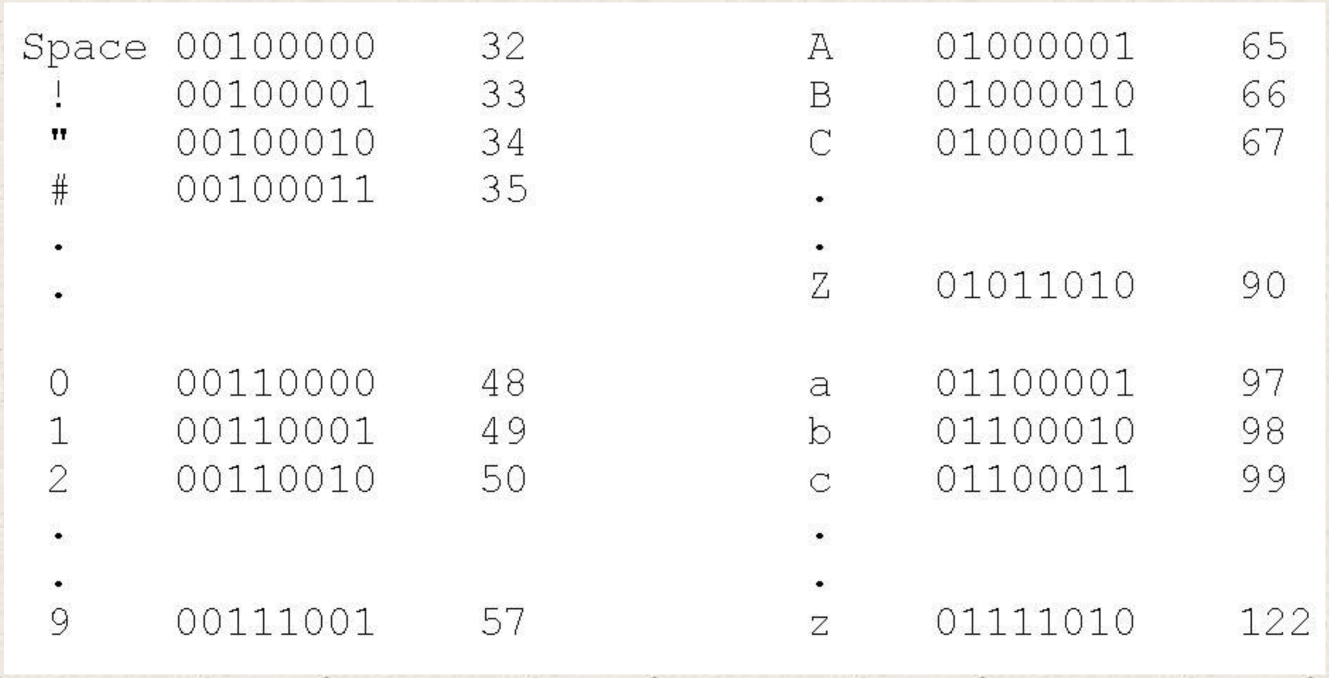
```
>>> print('Hello')          >>> print('Hello")           >>> print('Let's Go')
???                         ???                          ???

>>> print("Hello")          >>> print("Let's Go!')       >>> print("Let's go!")
???                         ???                          ???
```

The Representation of Character Values

There needs to be a way to encode (represent) characters within a computer. Although various encoding schemes have been developed, the **Unicode encoding scheme** is intended to be a universal encoding scheme.

Unicode is actually a collection of different encoding schemes utilizing between 8 and 32 bits for each character. The default encoding in Python uses **UTF-8**, an 8-bit encoding compatible with **ASCII**, an older, still widely used encoding scheme.

Currently, there are over 100,000 Unicode-defined characters for many of the languages around the world. Unicode is capable of defining more than four billion characters. Thus, all the world's languages, both past and present, can potentially be encoded within Unicode.

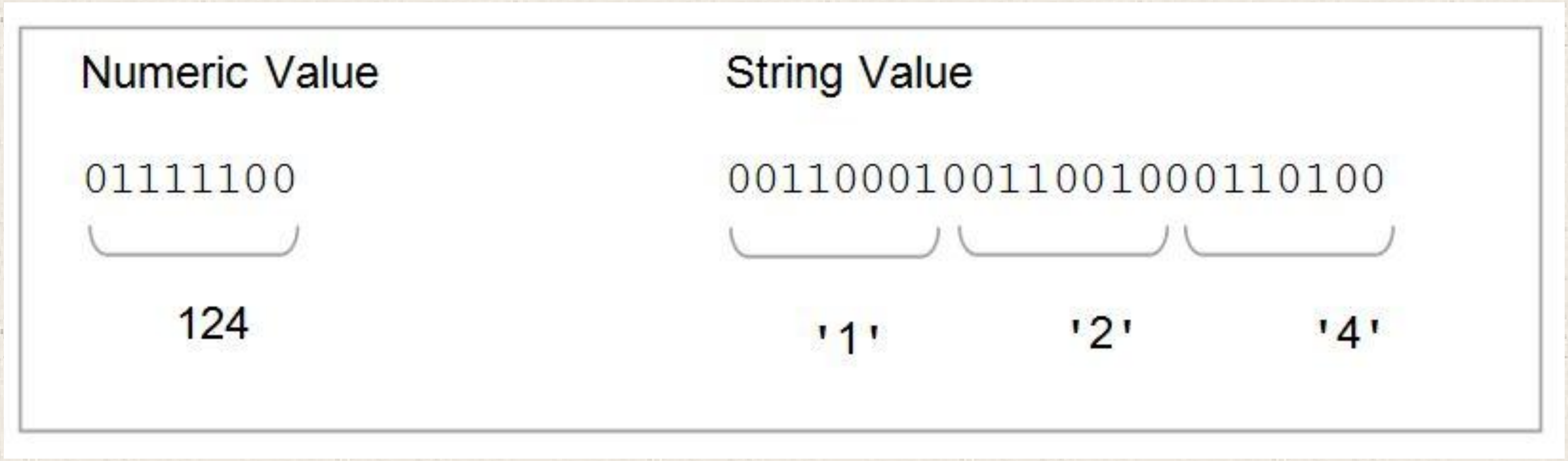


Space	00100000	32	A	01000001	65
!	00100001	33	B	01000010	66
"	00100010	34	C	01000011	67
#	00100011	35	.		
.			.		
.			Z	01011010	90
0	00110000	48	a	01100001	97
1	00110001	49	b	01100010	98
2	00110010	50	c	01100011	99
.			.		
.			.		
9	00111001	57	z	01111010	122

Partial listing of the ASCII-compatible UTF-8 encoding scheme

UTF-8 encodes characters that have an ordering with sequential numerical values. For example, 'A' is encoded as 01000001 (65), 'B' is encoded as 01000010 (66), and so on. This is also true for digit characters, ‘0’ is encoded as 48, ‘1’ as 49, etc.

Note the difference between a numeric representation (that can be used in arithmetic calculations) vs. a number represented as a string of digit characters (not used in calculations).



Here, the binary representation of 124 is the binary number for that value. The string representation ‘124’ consists of a longer sequence of bits, eight bits (one byte) for each digit character.

Converting Between a Character and Its Encoding

Python has a means of converting between a character and its encoding.

The ord function gives the UTF-8 (ASCII) encoding of a given character. For example,

`ord('A')` is 65

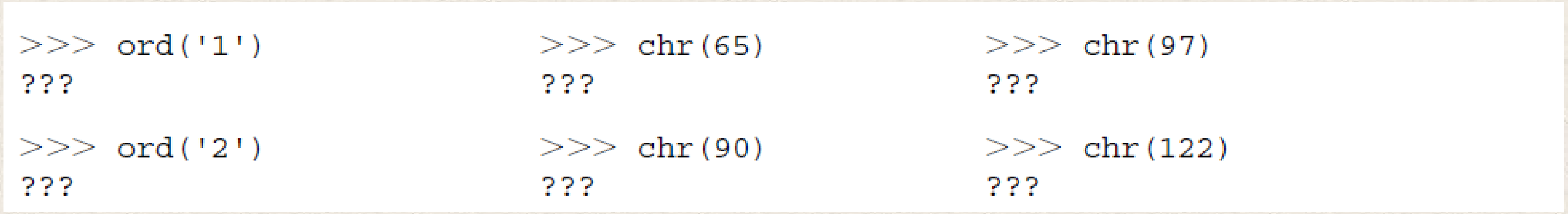
The chr function gives the character for a given encoding value, thus

`chr(65)` is 'A'

While in general there is no need to know the specific encoding of a given character, there are times when such knowledge can be useful.

Let’s Try It

What is the result of each of the following conversions?



```
>>> ord('1')             >>> chr(65)          >>> chr(97)
???                      ???                  ???

>>> ord('2')             >>> chr(90)          >>> chr(122)
???                      ???                  ???
```

Control Characters

Control characters are special characters that are not displayed, but rather *control* the display of output, among other things. Control characters do not have a corresponding keyboard character, and thus are represented by a combination of characters called an ***escape sequence*** .

Escape sequences begin with an ***escape character*** that causes the characters following it to “escape” their normal meaning. **The backslash (\\) serves as the escape character in Python**. For example, **'\n'**, represents the ***newline control character,*** that begins a new screen line,

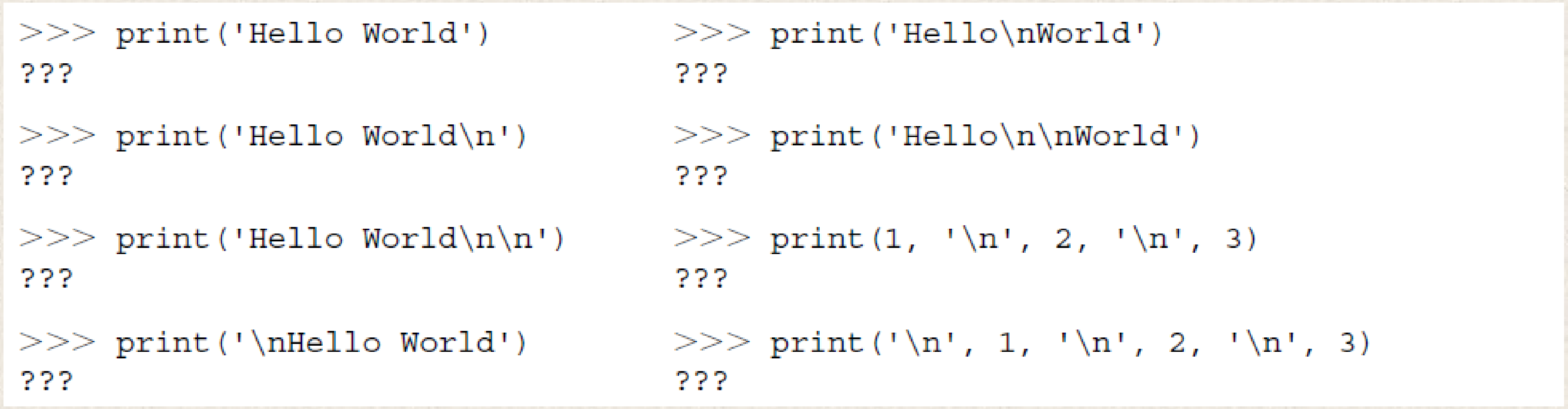
```
print('Hello\nJennifer Smith')
```

which is displayed as follows:

```
Hello
Jennifer Smith
```

Let’s Try It

What is displayed by each of the following?



```
>>> print('Hello World')
???

>>> print('Hello World\n')
???

>>> print('Hello World\n\n')
???

>>> print('\nHello World')
???
```

```
>>> print('Hello\nWorld')
???

>>> print('Hello\n\nWorld')
???

>>> print(1, '\n', 2, '\n', 3)
???

>>> print('\n', 1, '\n', 2, '\n', 3)
???
```

String Formatting

Built-in function format can be used for controlling how strings are displayed, in addition to controlling the display of numerical values,

```
format(value, format_specifier)
```

where ***value*** is the value to be displayed, and ***format_specifier*** can contain a combination of formatting options.

For example, the following produces the string 'Hello' **left-justified in a field width of 20 characters**,

```
format('Hello', '< 20') → 'Hello               '
```

To **right-justify the string**, the following would be used,

```
format('Hello', '> 20') → '               Hello'
```

Formatted strings are left-justified by default. Therefore, the following produce the same result,

```
format('Hello', '< 20') → 'Hello               '
format('Hello', '20') →   'Hello               '
```

To **center a string** the **'^'** character is used:

```
format('Hello', '^20')
```

Another use of the format function is to **create a string of blank characters**, which is sometimes useful,

```
format(' ', '15') → '               '
```

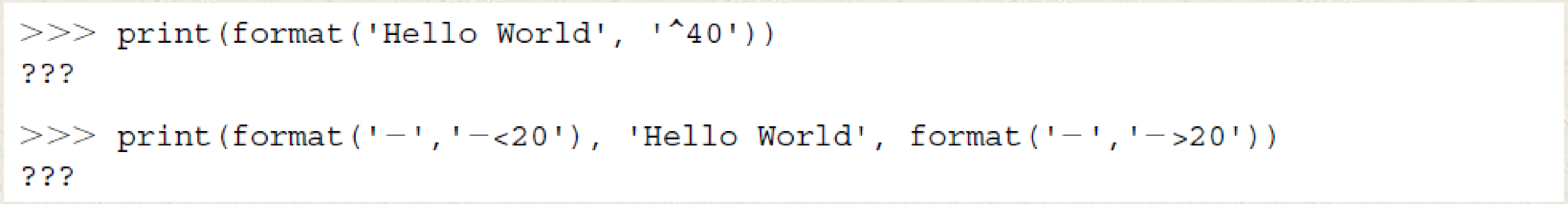
Finally **blanks, by default, are the fill character** for formatted strings. However, **a specific fill character can be specified** as shown below,

```
>>> print('Hello', format('.', '.< 30'), 'Have a Nice Day!')
Hello .............................. Have a Nice Day!
```

Here, a single period is the character printed within a field width of 30, therefore ultimately printing out 30 periods.

Let's Try It

What is displayed by each of the following?



```
>>> print(format('Hello World', '^40'))
???

>>> print(format('-','-<20'), 'Hello World', format('-','->20'))
???
```

Implicit and Explicit Line Joining

Sometimes a program line may be too long to fit in the Python-recommended maximum length of 79 characters. There are two ways in Python to deal with such situations:

- **implicit line joining**
- **explicit line joining**

Implicit Line Joining

There are certain delimiting characters that allow a *logical program line* to span more than one *physical line*. This includes matching parentheses, square brackets, curly braces, and triple quotes.

For example, the following two program lines are treated as one logical line:

```
print('Name:',student_name, 'Address:', student_address,
      'Number of Credits:', total_credits, 'GPA:', current_gpa)
```

Matching quotes (except for triple quotes) must be on the same physical line. For example, the following will generate an error:

```
print('This program will calculate a restaurant tab for a couple
      with a gift certificate, and a restaurant tax of 3%')
```

open quote

close quote

Explicit Line Joining

In addition to implicit line joining, program lines may be explicitly joined by use of the backslash (\\) character. Program lines that end with a backslash that are not part of a literal string (that is, within quotes) continue on the following line.

```
numsecs_1900_dob  =  ((year_birth 2 1900) * avg_numsecs_year)  +  \
                     ((month_birth 2 1) * avg_numsecs_month) +  \
                      (day_birth * numsecs_day)
```

Let’s Apply It

Hello World Unicode Encoding

It is a long tradition in computer science to demonstrate a program that simply displays “Hello World!” as an example of the simplest program possible in a particular programming language. In Python, the complete Hello World program is comprised of one program line:

```
print('Hello World!')
```

We take a twist on this tradition and give a Python program that displays the Unicode encoding for each of the characters in the string “Hello World!” instead. This program utilizes the following programming features:

- **string literals**
- **print**
- **ord function**

```
1  # This program displays the Unicode encoding for 'Hello World!'
2
3  # Program greeting
4  print("The Unicode encoding for 'Hello World!' is:")
5
6  # output results
7  print(ord('H'), ord('e'), ord('l'), ord('l'), ord('o'), ord(' '),
8        ord('W'), ord('o'), ord('r'), ord('l'), ord('d'), ord('!'))
```

Program Execution ...

```
The Unicode encoding for 'Hello World!' is:
72 101 108 108 111 32 87 111 114 108 100 33
```

The statements on **lines 1, 3,** and **6** are **comment statements**. The **print function** on **line 4** displays the message 'Hello World!'. **Double quotes** are used to delimit the corresponding string, since the single quotes within it are to be taken literally. The use of print on **line 7** **prints out the Unicode encoding**, one-by-one, for each of the characters in the "Hello World!" string. Note from the program execution that there is a Unicode encoding for the blank character (32), as well as the exclamation mark (33).

Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
   (a) `1024` (b) `1,024` (c) `1024.0` (d) `0.25` (e) `.45` (f) `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
   (a) `1.89345348392e+301` (c) `2.0424e−320`
   (b) `1.6212343263232 2e+300` (d) `1.323232435342327896452e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
   (a) `6.25e+240 * 1.24e+10` (c) `6.25e+240 / 1.24e+10`
   (b) `2.24e+240 * 1.45e+300` (d) `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
   (a) `24.894` (b) `24.893` (c) `2.48e1`

5. Which of the following are valid string literals in Python.
   (a) `"Hello"` (b) `'hello'` (c) `"Hello'` (d) `'Hello there'` (e) `''`

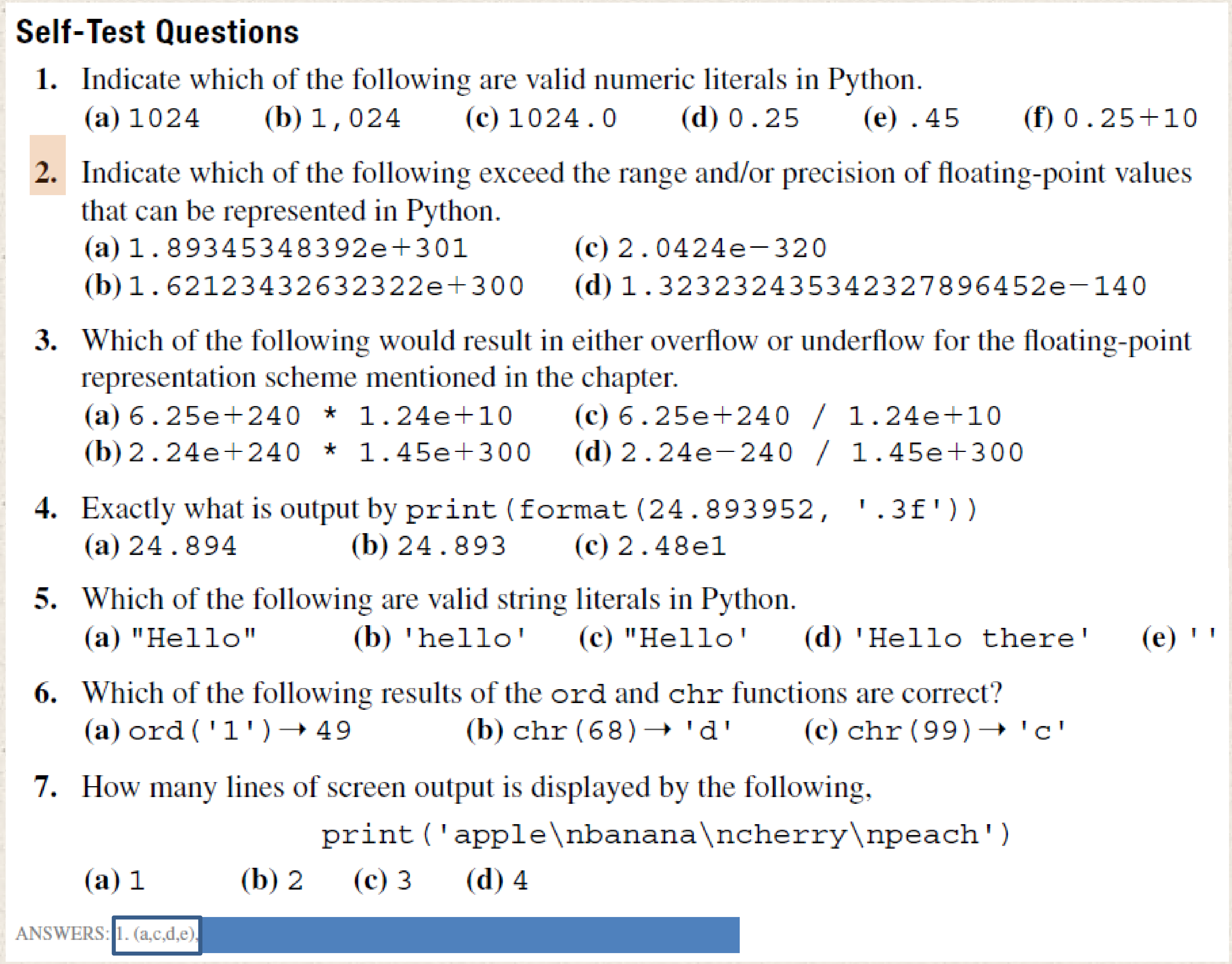
6. Which of the following results of the `ord` and `chr` functions are correct?
   (a) `ord('1')` → `49` (b) `chr(68)` → `'d'` (c) `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

   (a) `1` (b) `2` (c) `3` (d) `4`

ANSWERS: [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
(a) `1024` **(b)** `1,024` **(c)** `1024.0` **(d)** `0.25` **(e)** `.45` **(f)** `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
(a) `1.89345348392e+301` **(c)** `2.0424e-320`
(b) `1.62123432632322e+300` **(d)** `1.3232324353423278964552e-140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
(a) `6.25e+240 * 1.24e+10` **(c)** `6.25e+240 / 1.24e+10`
(b) `2.24e+240 * 1.45e+300` **(d)** `2.24e-240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
(a) `24.894` **(b)** `24.893` **(c)** `2.48e1`

5. Which of the following are valid string literals in Python.
(a) `"Hello"` **(b)** `'hello'` **(c)** `"Hello'` **(d)** `'Hello there'` **(e)** `''`

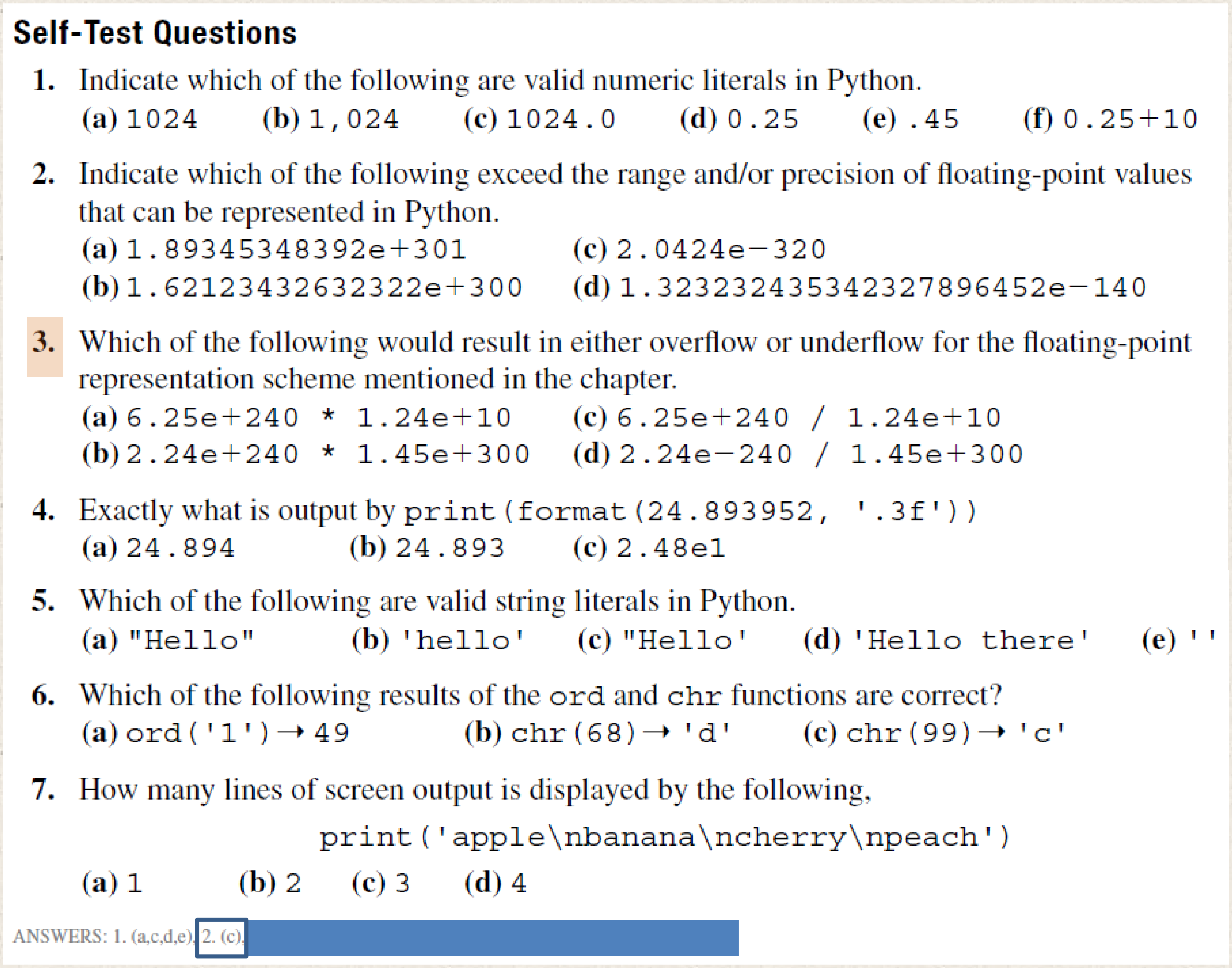
6. Which of the following results of the `ord` and `chr` functions are correct?
(a) `ord('1')` → `49` **(b)** `chr(68)` → `'d'` **(c)** `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

(a) 1 **(b)** 2 **(c)** 3 **(d)** 4

ANSWERS: 1. (a,c,d,e), [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
   (a) `1024` (b) `1,024` (c) `1024.0` (d) `0.25` (e) `.45` (f) `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
   (a) `1.89345348392e+301` (c) `2.0424e−320`
   (b) `1.6212343263232 2e+300` (d) `1.323232435342327896452e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
   (a) `6.25e+240 * 1.24e+10` (c) `6.25e+240 / 1.24e+10`
   (b) `2.24e+240 * 1.45e+300` (d) `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
   (a) `24.894` (b) `24.893` (c) `2.48e1`

5. Which of the following are valid string literals in Python.
   (a) `"Hello"` (b) `'hello'` (c) `"Hello'` (d) `'Hello there'` (e) `''`

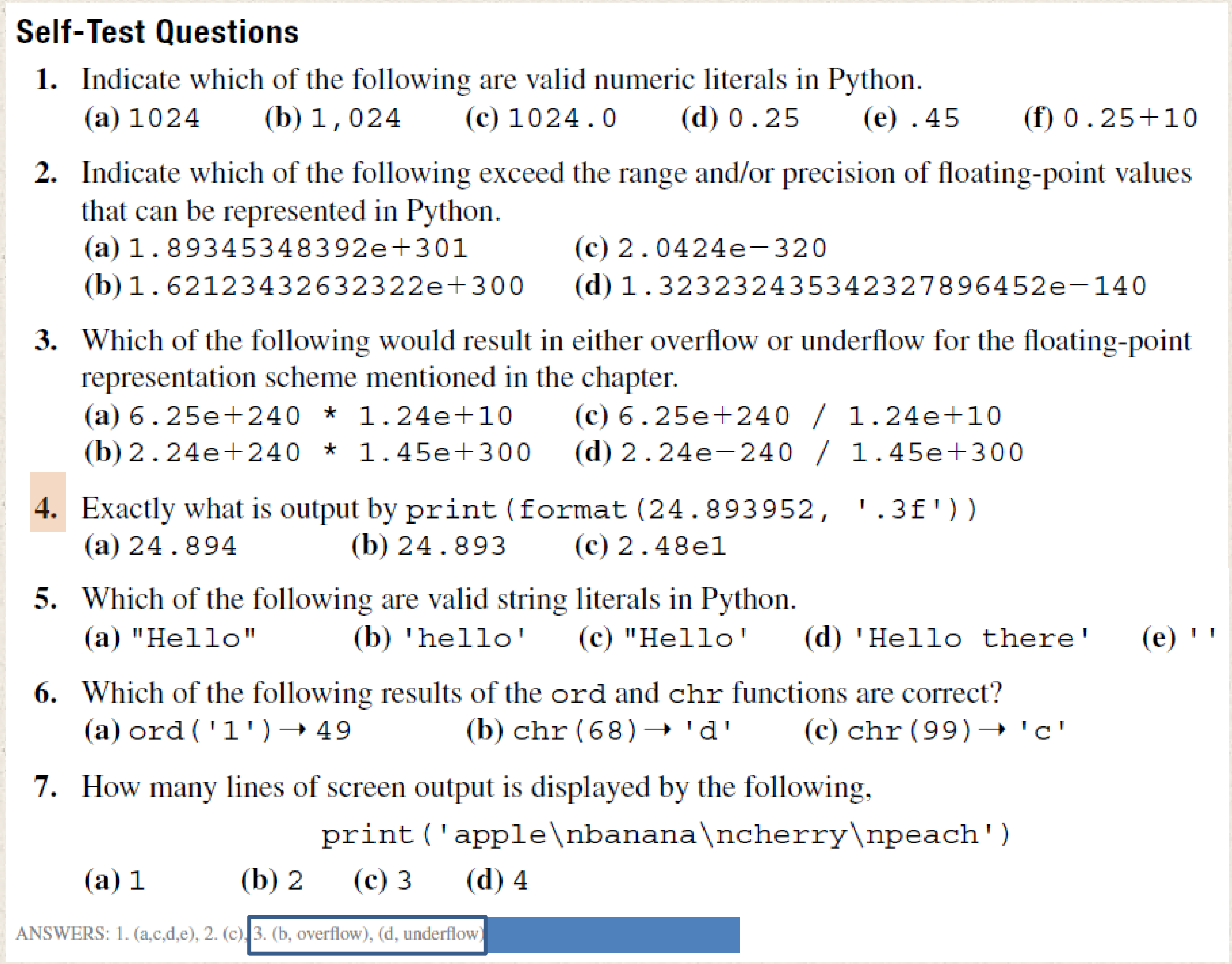
6. Which of the following results of the `ord` and `chr` functions are correct?
   (a) `ord('1')` → `49` (b) `chr(68)` → `'d'` (c) `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

   (a) `1` (b) `2` (c) `3` (d) `4`

ANSWERS: 1. (a,c,d,e), 2. (c), [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
   (a) `1024` (b) `1,024` (c) `1024.0` (d) `0.25` (e) `.45` (f) `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
   (a) `1.89345348392e+301` (c) `2.0424e−320`
   (b) `1.6212343263222e+300` (d) `1.323232435342327896452e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
   (a) `6.25e+240 * 1.24e+10` (c) `6.25e+240 / 1.24e+10`
   (b) `2.24e+240 * 1.45e+300` (d) `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
   (a) `24.894` (b) `24.893` (c) `2.48e1`

5. Which of the following are valid string literals in Python.
   (a) `"Hello"` (b) `'hello'` (c) `"Hello'` (d) `'Hello there'` (e) `''`

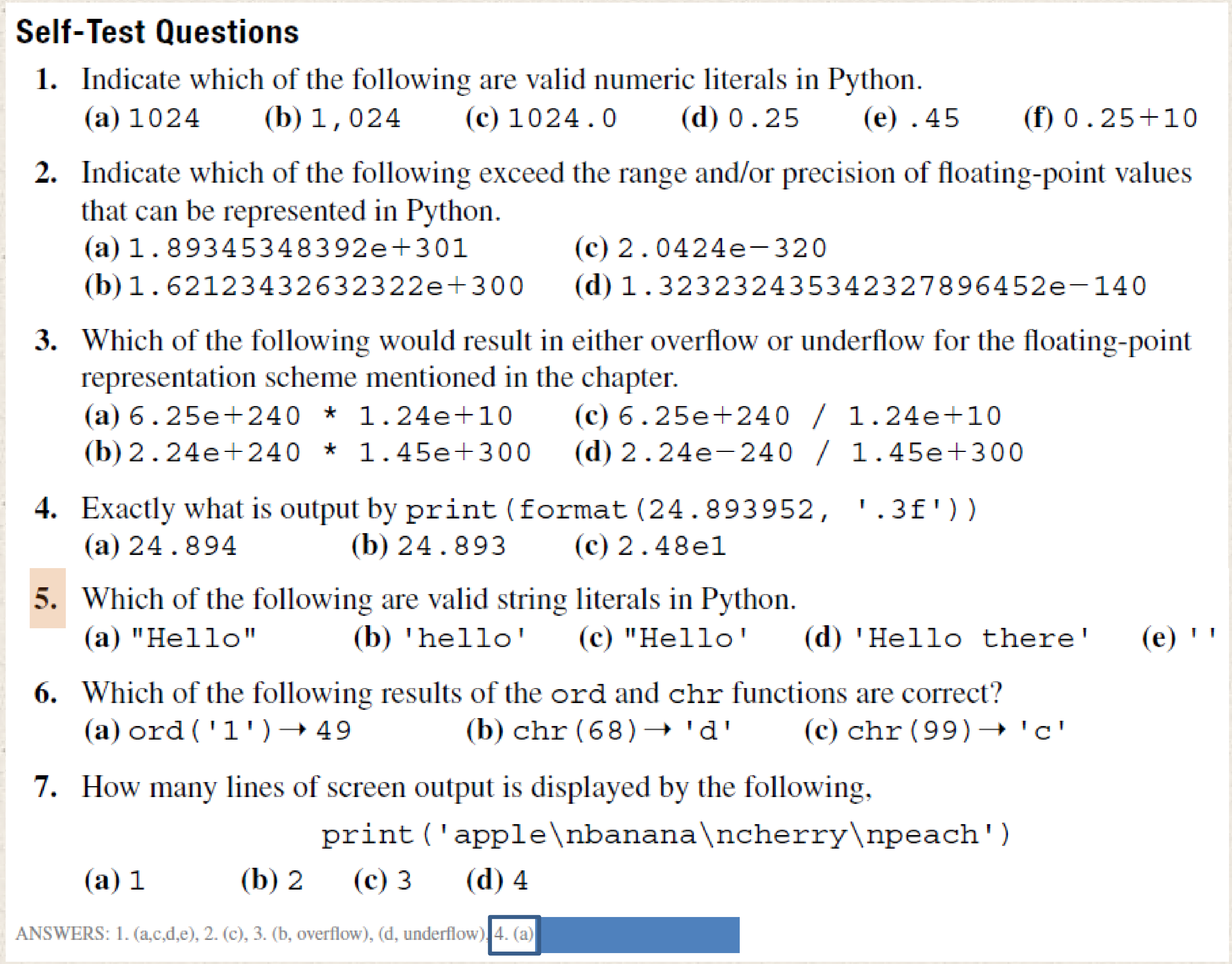
6. Which of the following results of the `ord` and `chr` functions are correct?
   (a) `ord('1')` → `49` (b) `chr(68)` → `'d'` (c) `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

   (a) 1 (b) 2 (c) 3 (d) 4

ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow) [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
   (a) `1024` (b) `1,024` (c) `1024.0` (d) `0.25` (e) `.45` (f) `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
   (a) `1.89345348392e+301` (c) `2.0424e−320`
   (b) `1.6212343263232 2e+300` (d) `1.3232324353423278964 52e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
   (a) `6.25e+240 * 1.24e+10` (c) `6.25e+240 / 1.24e+10`
   (b) `2.24e+240 * 1.45e+300` (d) `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
   (a) `24.894` (b) `24.893` (c) `2.48e1`

5. Which of the following are valid string literals in Python.
   (a) `"Hello"` (b) `'hello'` (c) `"Hello'` (d) `'Hello there'` (e) `''`

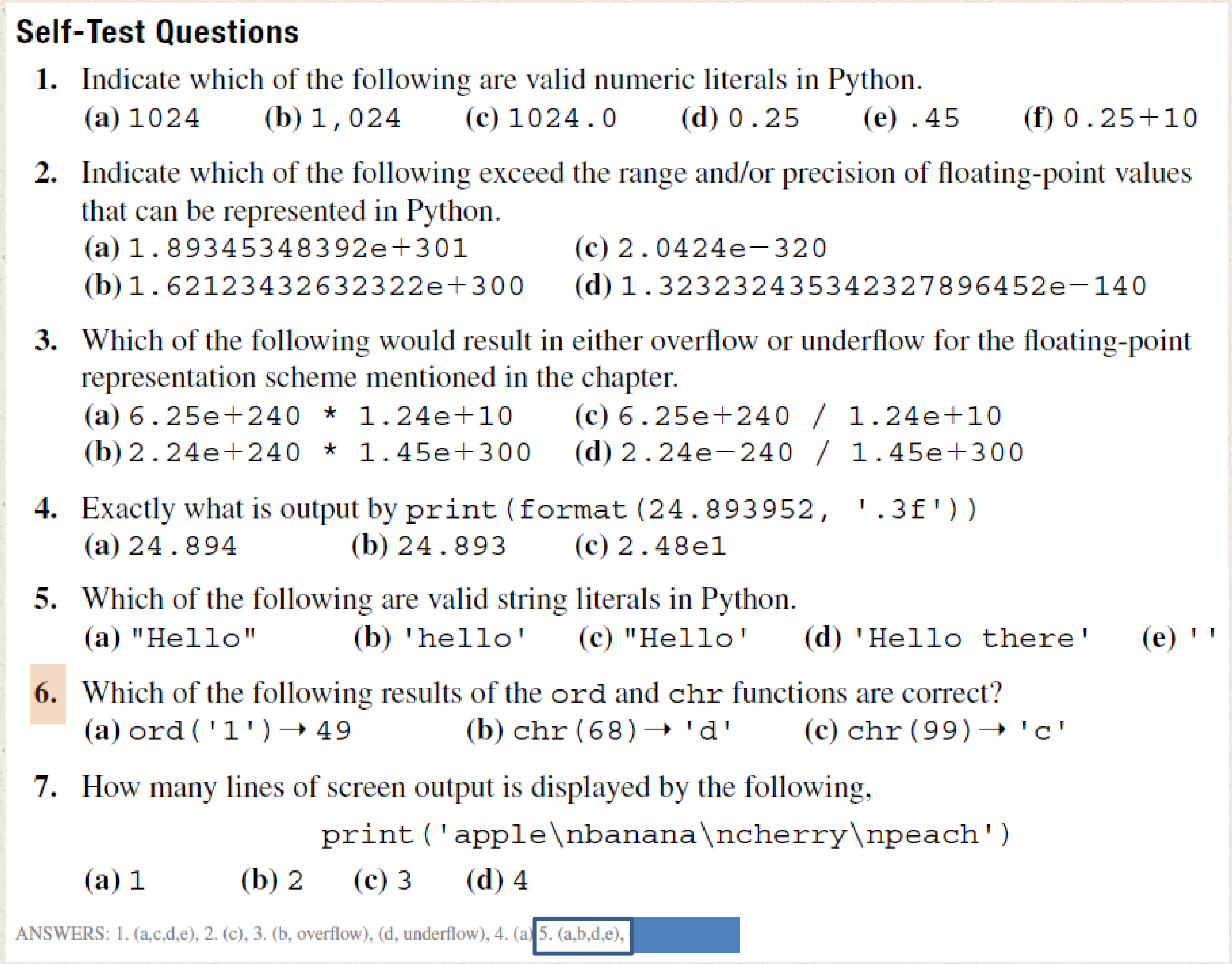
6. Which of the following results of the `ord` and `chr` functions are correct?
   (a) `ord('1')` → `49` (b) `chr(68)` → `'d'` (c) `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

   (a) `1` (b) `2` (c) `3` (d) `4`

ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
(a) `1024` **(b)** `1,024` **(c)** `1024.0` **(d)** `0.25` **(e)** `.45` **(f)** `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
(a) `1.89345348392e+301` **(c)** `2.0424e−320`
(b) `1.62123432632322e+300` **(d)** `1.3232324353423278964 52e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
(a) `6.25e+240 * 1.24e+10` **(c)** `6.25e+240 / 1.24e+10`
(b) `2.24e+240 * 1.45e+300` **(d)** `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
(a) `24.894` **(b)** `24.893` **(c)** `2.48e1`

5. Which of the following are valid string literals in Python.
(a) `"Hello"` **(b)** `'hello'` **(c)** `"Hello'` **(d)** `'Hello there'` **(e)** `''`

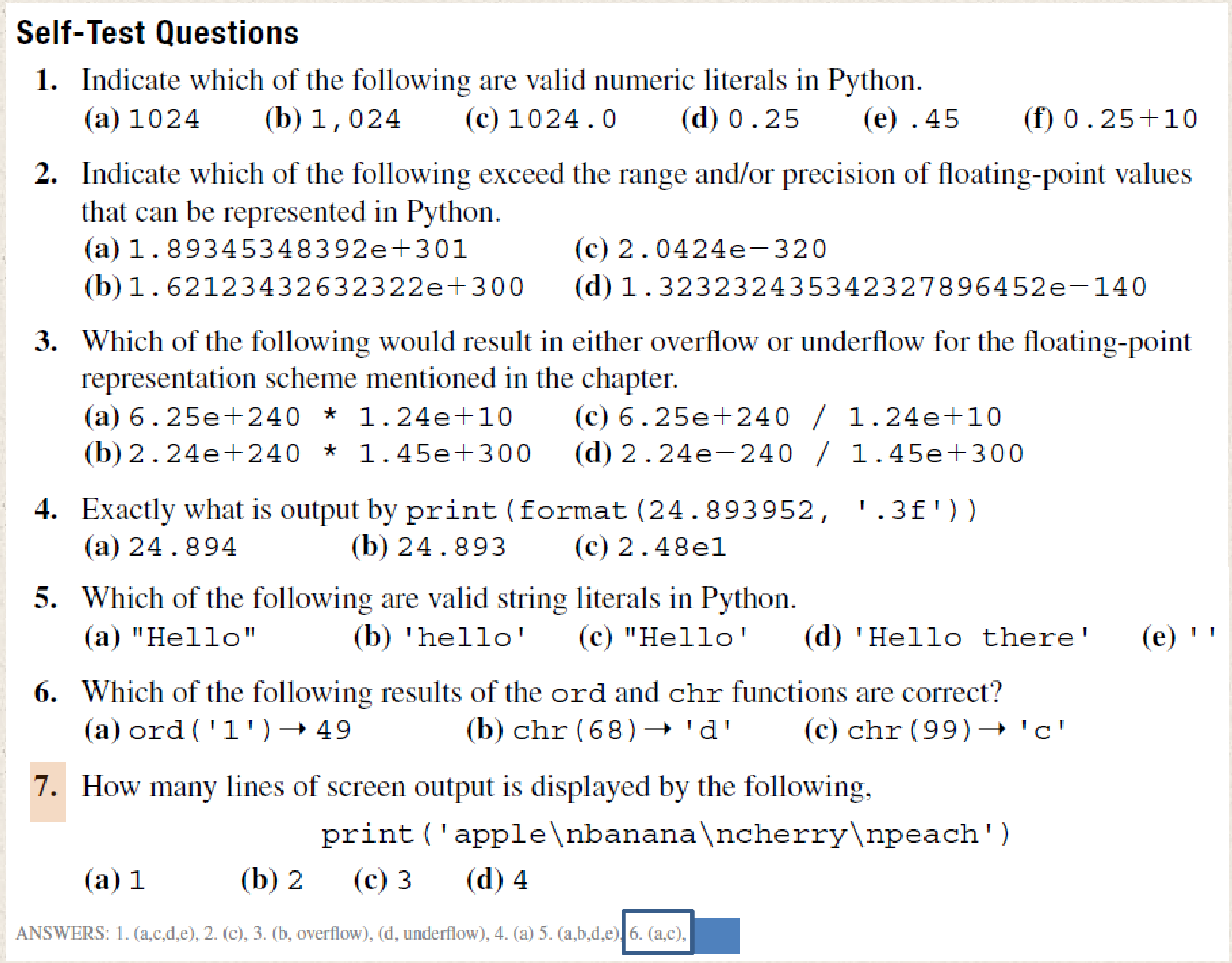
6. Which of the following results of the `ord` and `chr` functions are correct?
(a) `ord('1')` → `49` **(b)** `chr(68)` → `'d'` **(c)** `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

(a) `1` **(b)** `2` **(c)** `3` **(d)** `4`

ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) 5. (a,b,d,e), [redacted]



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
(a) `1024` **(b)** `1,024` **(c)** `1024.0` **(d)** `0.25` **(e)** `.45` **(f)** `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
(a) `1.89345348392e+301` **(c)** `2.0424e−320`
(b) `1.62123432632322e+300` **(d)** `1.323232435342327896452e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
(a) `6.25e+240 * 1.24e+10` **(c)** `6.25e+240 / 1.24e+10`
(b) `2.24e+240 * 1.45e+300` **(d)** `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
(a) `24.894` **(b)** `24.893` **(c)** `2.48e1`

5. Which of the following are valid string literals in Python.
(a) `"Hello"` **(b)** `'hello'` **(c)** `"Hello'` **(d)** `'Hello there'` **(e)** `''`

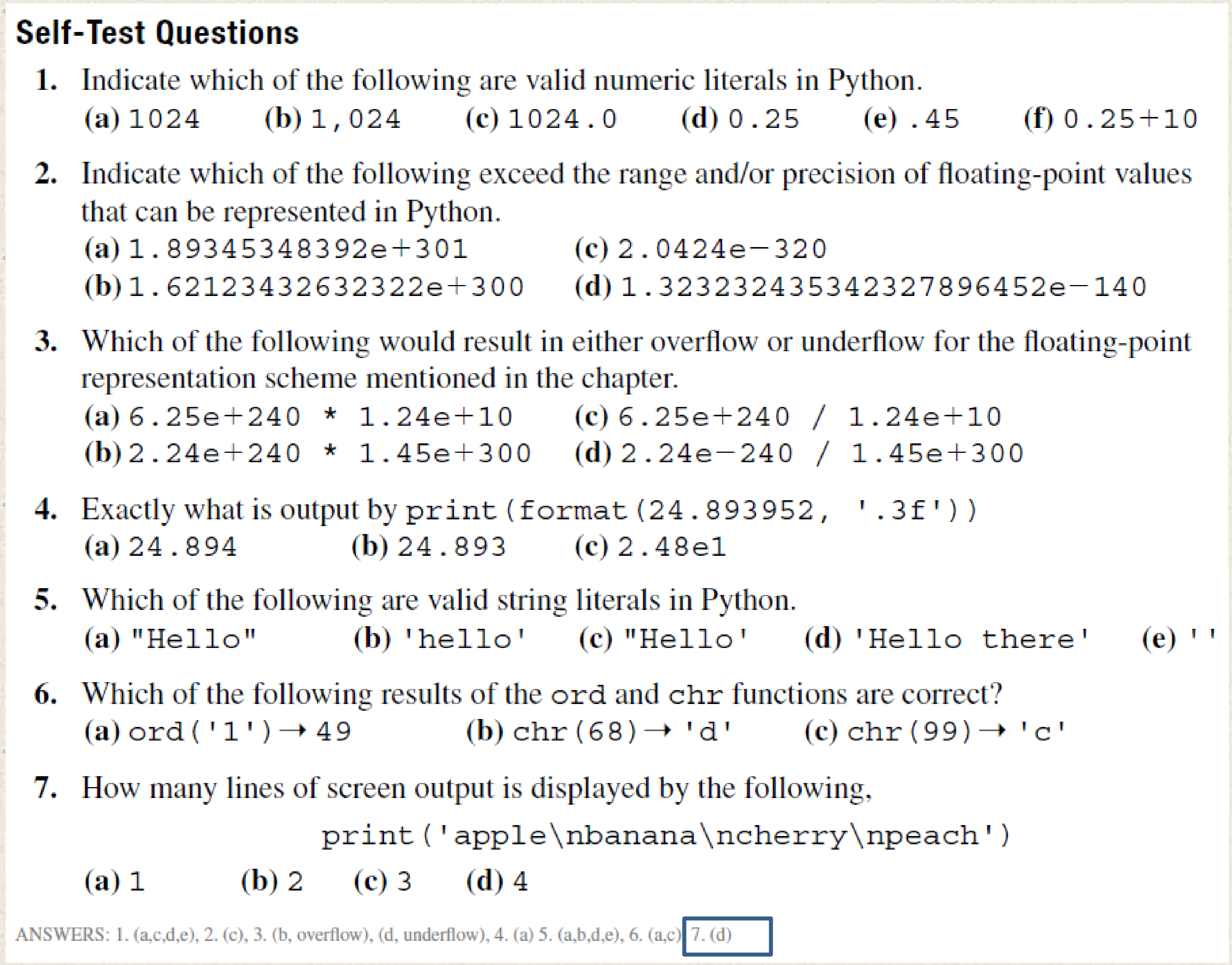
6. Which of the following results of the `ord` and `chr` functions are correct?
(a) `ord('1')` → `49` **(b)** `chr(68)` → `'d'` **(c)** `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

(a) `1` **(b)** `2` **(c)** `3` **(d)** `4`

ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) 5. (a,b,d,e), 6. (a,c),



Self-Test Questions

1. Indicate which of the following are valid numeric literals in Python.
   (a) `1024` (b) `1,024` (c) `1024.0` (d) `0.25` (e) `.45` (f) `0.25+10`

2. Indicate which of the following exceed the range and/or precision of floating-point values that can be represented in Python.
   (a) `1.89345348392e+301` (c) `2.0424e−320`
   (b) `1.62123432632322e+300` (d) `1.3232324353423278964 52e−140`

3. Which of the following would result in either overflow or underflow for the floating-point representation scheme mentioned in the chapter.
   (a) `6.25e+240 * 1.24e+10` (c) `6.25e+240 / 1.24e+10`
   (b) `2.24e+240 * 1.45e+300` (d) `2.24e−240 / 1.45e+300`

4. Exactly what is output by `print(format(24.893952, '.3f'))`
   (a) `24.894` (b) `24.893` (c) `2.48e1`

5. Which of the following are valid string literals in Python.
   (a) `"Hello"` (b) `'hello'` (c) `"Hello'` (d) `'Hello there'` (e) `''`

6. Which of the following results of the `ord` and `chr` functions are correct?
   (a) `ord('1')` → `49` (b) `chr(68)` → `'d'` (c) `chr(99)` → `'c'`

7. How many lines of screen output is displayed by the following,

```
print('apple\nbanana\ncherry\npeach')
```

   (a) `1` (b) `2` (c) `3` (d) `4`

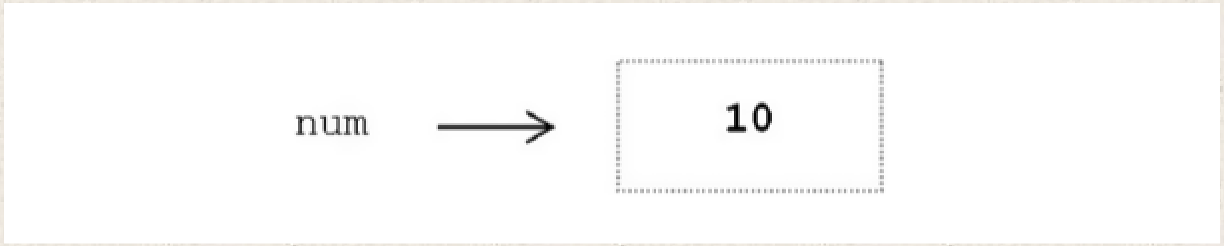
ANSWERS: 1. (a,c,d,e), 2. (c), 3. (b, overflow), (d, underflow), 4. (a) 5. (a,b,d,e), 6. (a,c) $\boxed{\text{7. (d)}}$

Variables and Identifiers

So far, we have only looked at literal values in programs. However, **the true usefulness of a computer program is the ability to operate on different values each time the program is executed**. This is provided by the notion of a ***variable***. We look at variables and identifiers next.

What Is a Variable?

A variable is a name (identifier) that is associated with a value,



A variable can be assigned different values during a program's execution—hence, the name "variable."

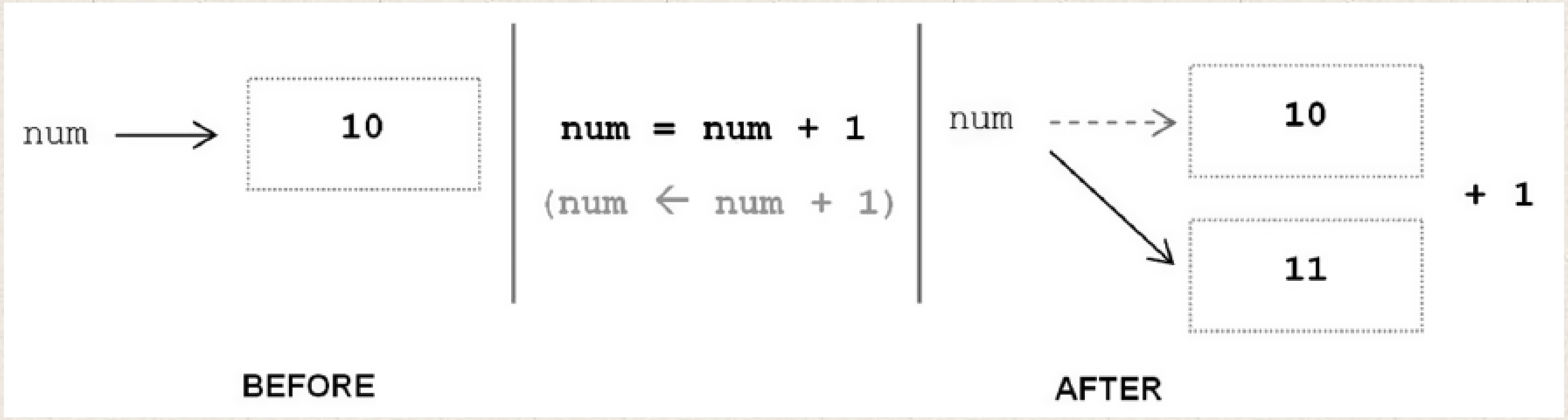
Wherever a variable appears in a program (except on the left-hand side of an assignment statement), **it is the value associated with the variable that is used**, and not the variable's name,

```
num + 1 → 10 + 1 → 11
```

Variables are assigned values by use of the **assignment operator , = ,**

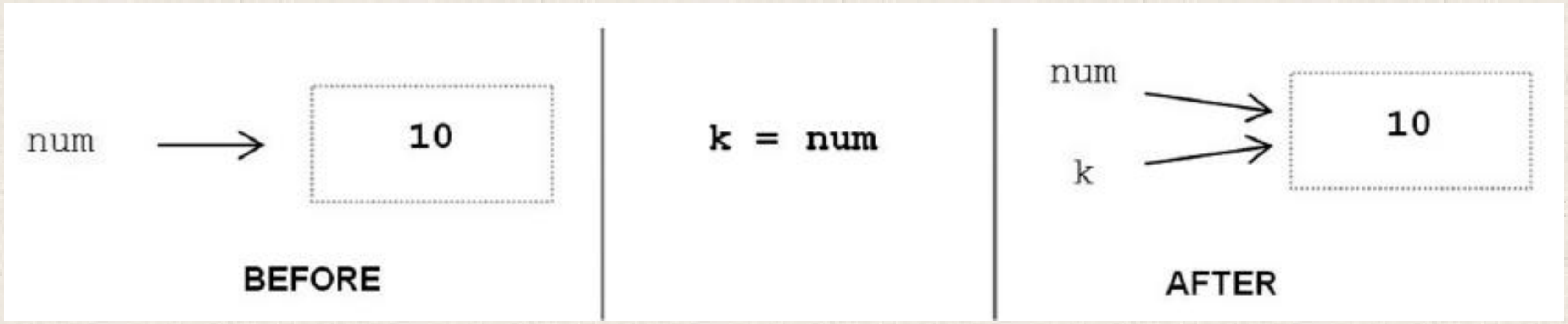
```
num = 10        num = num + 1
```

Assignment statements often look wrong to novice programmers. Mathematically, **num = num + 1** does not make sense. In computing, however, it is used to **increment** the value of a given variable by one. It is more appropriate, therefore, to think of the **=** symbol as an arrow symbol

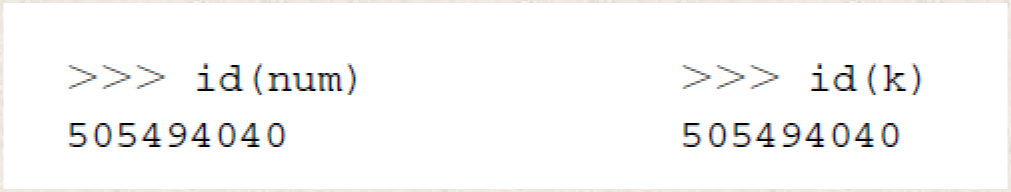


When thought of this way, it makes clear that **the right side of an assignment is evaluated first, then the result is assigned to the variable on the left**. An arrow symbol is not used simply because there is no such character on a standard computer keyboard.

Variables may also be assigned to the value of another variable,



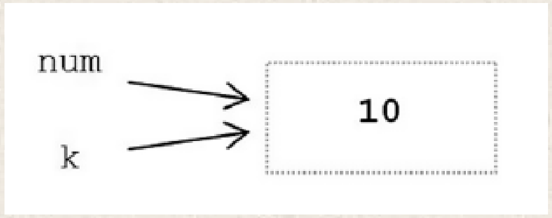
Variables **num** and **k** are both associated with the same literal value 10 in memory. One way to see this is by use of **built-in function** id,



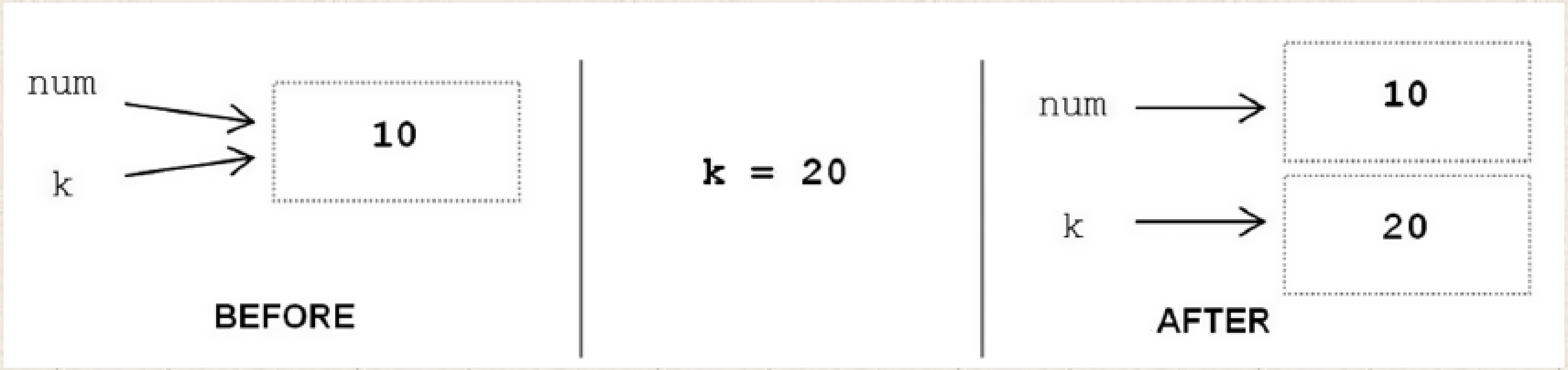
```
>>> id(num)                 >>> id(k)
505494040                   505494040
```

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems.

If the value of **num** changed, would variable **k** change along with it?



This cannot happen in this case because the variables refer to integer values, and integer values are *immutable*. An **immutable value** is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned,



If no other variable references the memory location of the original value, the memory location is ***deallocated*** *(that is, it is made available for reuse).*

Finally, in Python **the same variable can be associated with values of different type** during program execution,

```
var = 12              integer
var = 12.45           float
var = 'Hello'         string
```

(The ability of a variable to be assigned values of different type is referred to as ***dynamic typing***, introduced later in the discussions of data types.)

Let’s Try It

What do each of the following evaluate to?

```
>>> num = 10
>>> num
???
>>> id(num)
???

>>> num = 20
>>> num
???
>>> id(num)
???

>>> k = num
>>> k
???
>>> id(k)
???
>>> id(num)
???
```

```
>>> k = 30
>>> k
???
>>> num
???
>>> id(k)
???
>>> id(num)
???

>>> k = k + 1
>>> k
???
>>> id(num)
???
>>> id(k)
???
```

Variable Assignment and Keyboard Input

The value that is assigned to a given variable does not have to be specified in the program. The value can come from the user by use of the **input function** introduced in Chapter 1,

```
>>> name = input('What is your first name?')
What is your first name? John
```

In this case, the variable name is assigned the string `'John'`. If the user hit return without entering any value, name would be assigned to the **empty string** (`''`).

The input function returns a string type. For input of numeric values, the response must be converted to the appropriate type. Python provides built-in **type conversion functions int()** and **float ()** for this purpose,

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

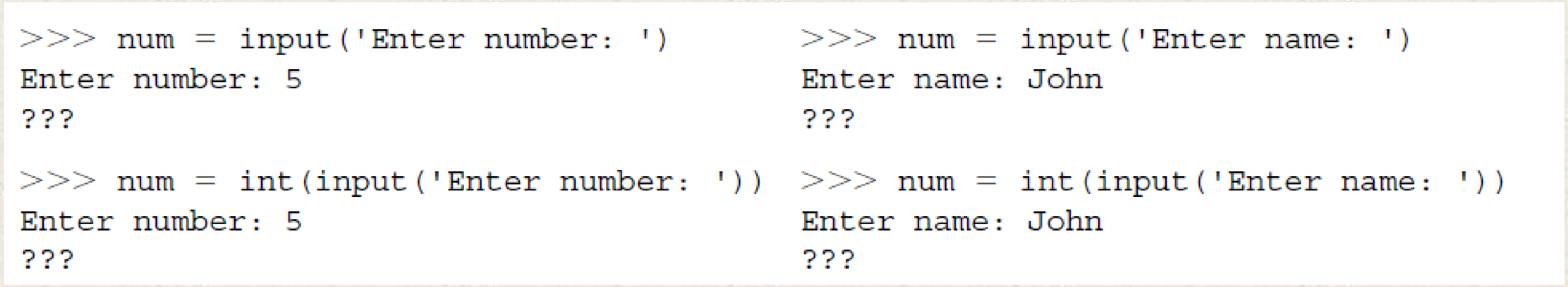
The entered number of credits, say `'24'`, is converted to the equivalent integer value, `24`, before being assigned to variable `num_credits`. The input value of the gpa, say `'3.2'`, is converted to the equivalent floating-point value, `3.2`.

Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

Let’s Try It

What is displayed by each of the following?



```
>>> num = input('Enter number: ')
Enter number: 5
???
```

```
>>> num = input('Enter name: ')
Enter name: John
???
```

```
>>> num = int(input('Enter number: '))
Enter number: 5
???
```

```
>>> num = int(input('Enter name: '))
Enter name: John
???
```

What Is an Identifier?

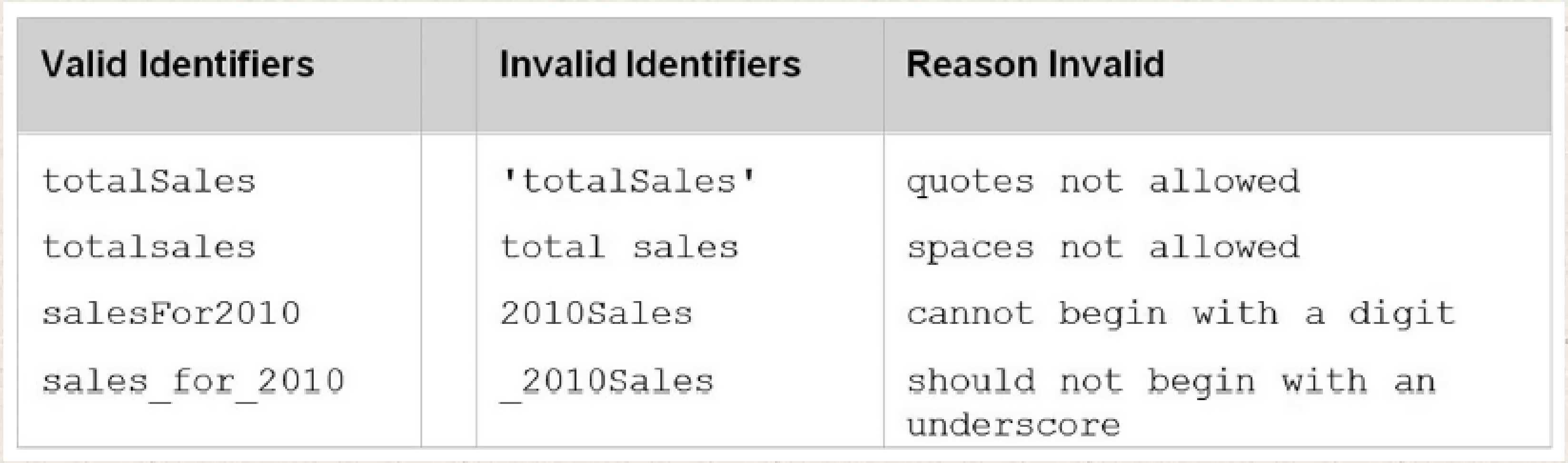
An **identifier** is a sequence of one or more characters used to provide a name for a given program element. Variable names `line`, `num_credits`, and `gpa` are each identifiers.

Python is **case sensitive** , thus, `Line` is different from `line`. Identifiers may contain letters and digits, but cannot begin with a digit.

The **underscore character**, **_**, is also allowed to aid in the readability of long identifier names. **It should not be used as the first character in user defined identifiers**, **however,** **as identifiers beginning with underscores and ending with underscores are valid and have special meaning in Python**.

Spaces are not allowed as part of an identifier. This is a common error since some operating systems allow spaces within file names. In programming languages, however, **spaces are used to delimit (separate) distinct syntactic entities.** Thus, any identifier containing a space character would be considered two separate identifiers.

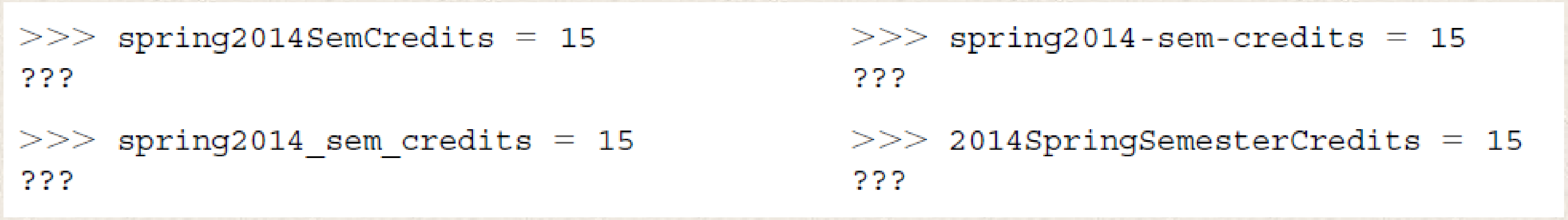
Examples of valid and invalid identifiers in Python



Valid Identifiers	Invalid Identifiers	Reason Invalid
`totalSales`	`'totalSales'`	quotes not allowed
`totalsales`	`total sales`	spaces not allowed
`salesFor2010`	`2010Sales`	cannot begin with a digit
`sales_for_2010`	`_2010Sales`	should not begin with an underscore

Let’s Try It

What is displayed by each of the following?



```
>>> spring2014SemCredits = 15
???

>>> spring2014_sem_credits = 15
???
```

```
>>> spring2014-sem-credits = 15
???

>>> 2014SpringSemesterCredits = 15
???
```

Keywords and Other Predefined Identifiers in Python

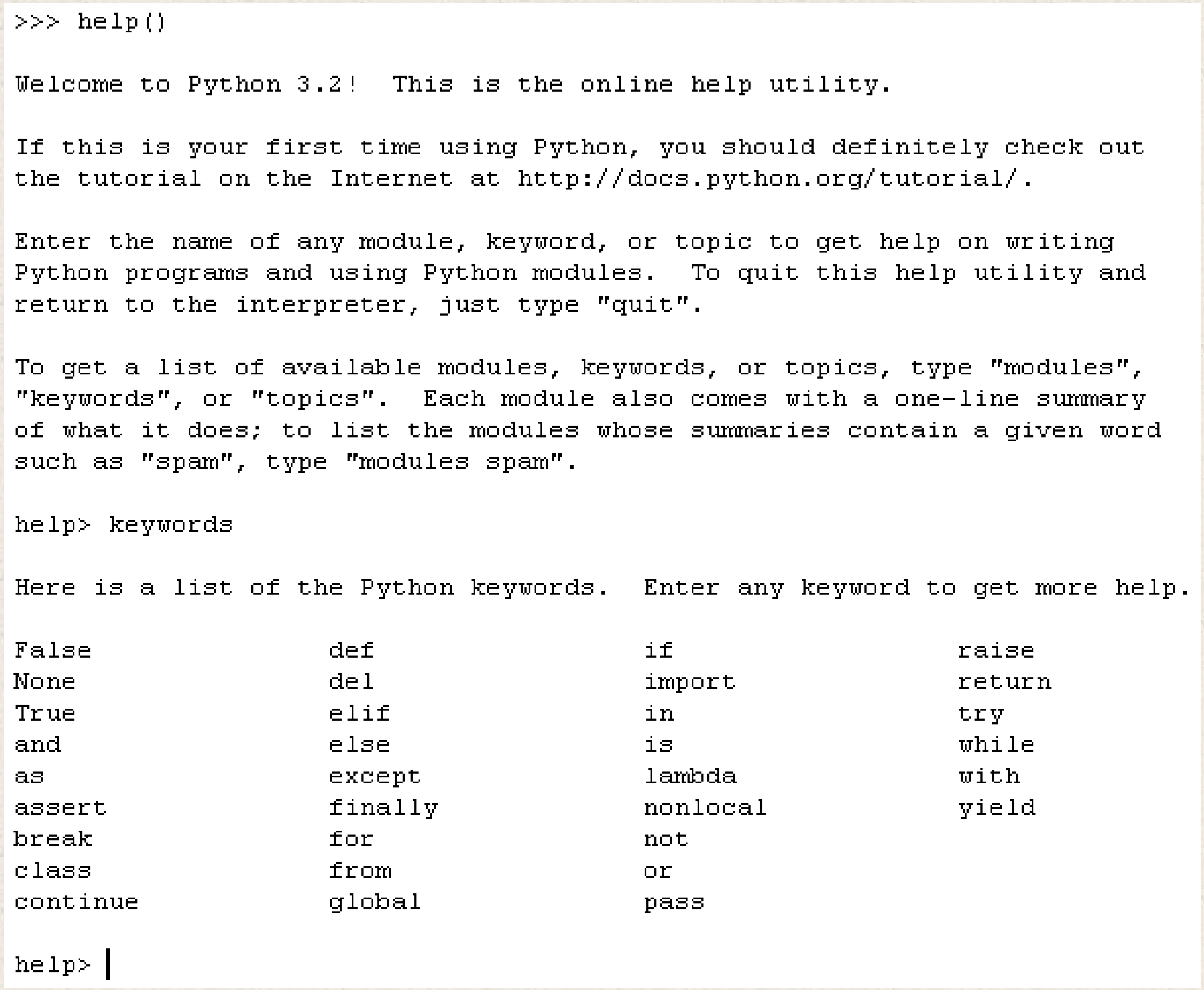
A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, **keywords cannot be used as "regular" identifiers**. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword **and** below,

```
>>> and = 10

SyntaxError: invalid syntax
```

The keywords in Python are listed below.

`and`	`as`	`assert`	`break`	`class`	`continue`	`def`
`del`	`elif`	`else`	`except`	`finally`	`for`	`from`
`global`	`if`	`import`	`in`	`is`	`lambda`	`nonlocal`
`not`	`or`	`pass`	`raise`	`return`	`try`	`while`
`with`	`yield`	`false`	`none`	`true`		

To display the keywords in Python, type **`help()`** in the Python shell, then type **`keywords`** (type 'q' to quit).



```
>>> help()

Welcome to Python 3.2!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass

help> 
```

There are other predefined identifiers that can be used as regular identifiers, but should not be. This includes **float**, **int**, **print**, **exit**, and **quit**, for example.

A simple way to check whether a specific identifier is a keyword in Python is as follows

```
>>> 'exit' in dir(__builtins__)
True

>>> 'exit_program' in dir(__builtins__)
False
```

Let’s Try It

What is displayed by each of the following?

```
>>> yield = 1000
???

>>> Yield == 1000
???
```

```
>>> print('Hello')
???

>>> print = 10
>>> print('Hello')
???
```

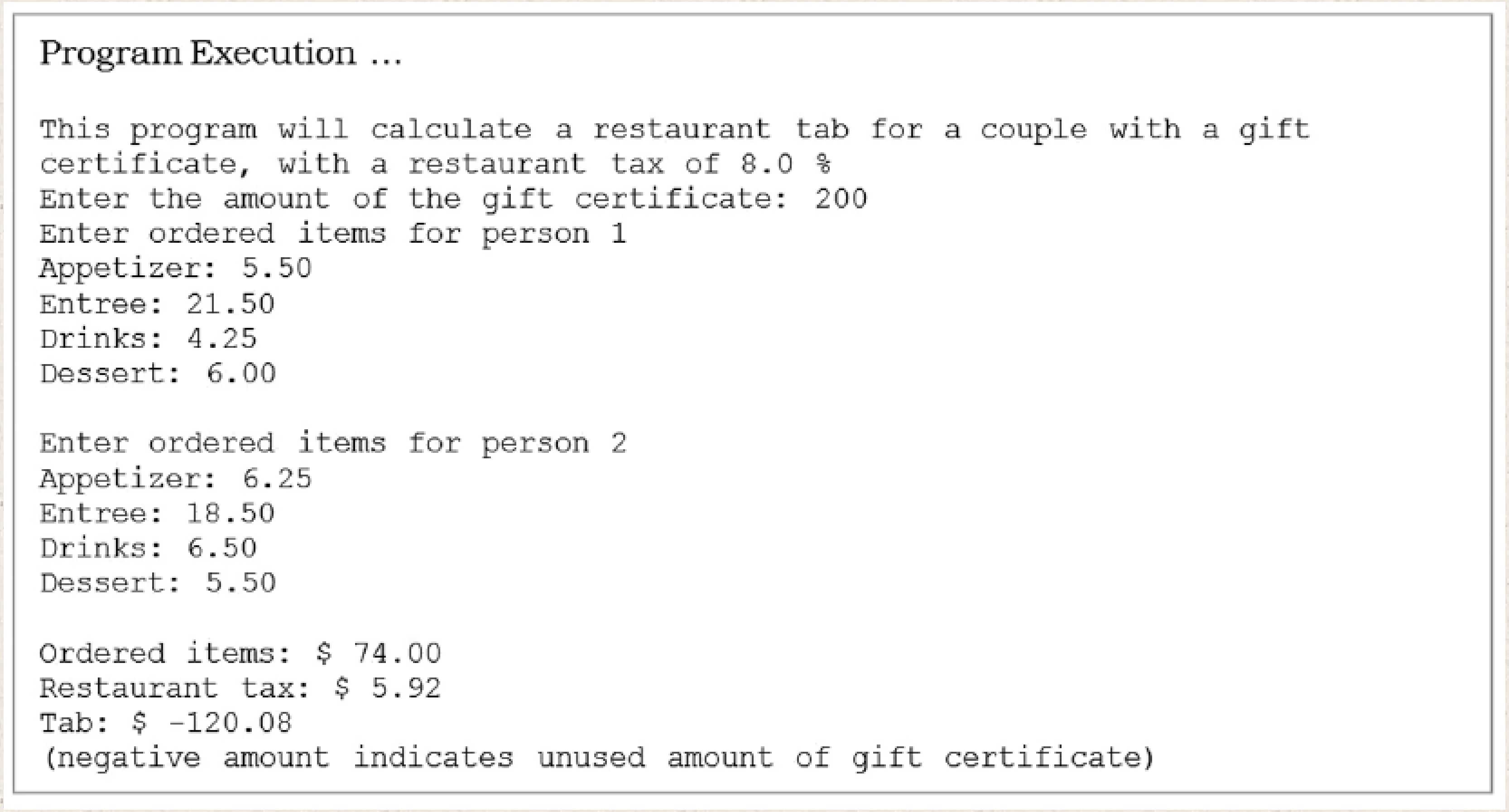
Let’s Apply It

Restaurant Tab Calculation

The program below calculates a restaurant tab for a couple based on the use of a gift certificate and the items ordered.

This program utilizes the following programming features:

- ➢ **variables**
- ➢ **keyboard input**
- ➢ **built-in format function**
- ➢ **type conversion functions**


Program Execution ...

```
This program will calculate a restaurant tab for a couple with a gift
certificate, with a restaurant tax of 8.0 %
Enter the amount of the gift certificate: 200
Enter ordered items for person 1
Appetizer: 5.50
Entree: 21.50
Drinks: 4.25
Dessert: 6.00

Enter ordered items for person 2
Appetizer: 6.25
Entree: 18.50
Drinks: 6.50
Dessert: 5.50

Ordered items: $ 74.00
Restaurant tax: $ 5.92
Tab: $ -120.08
(negative amount indicates unused amount of gift certificate)
```

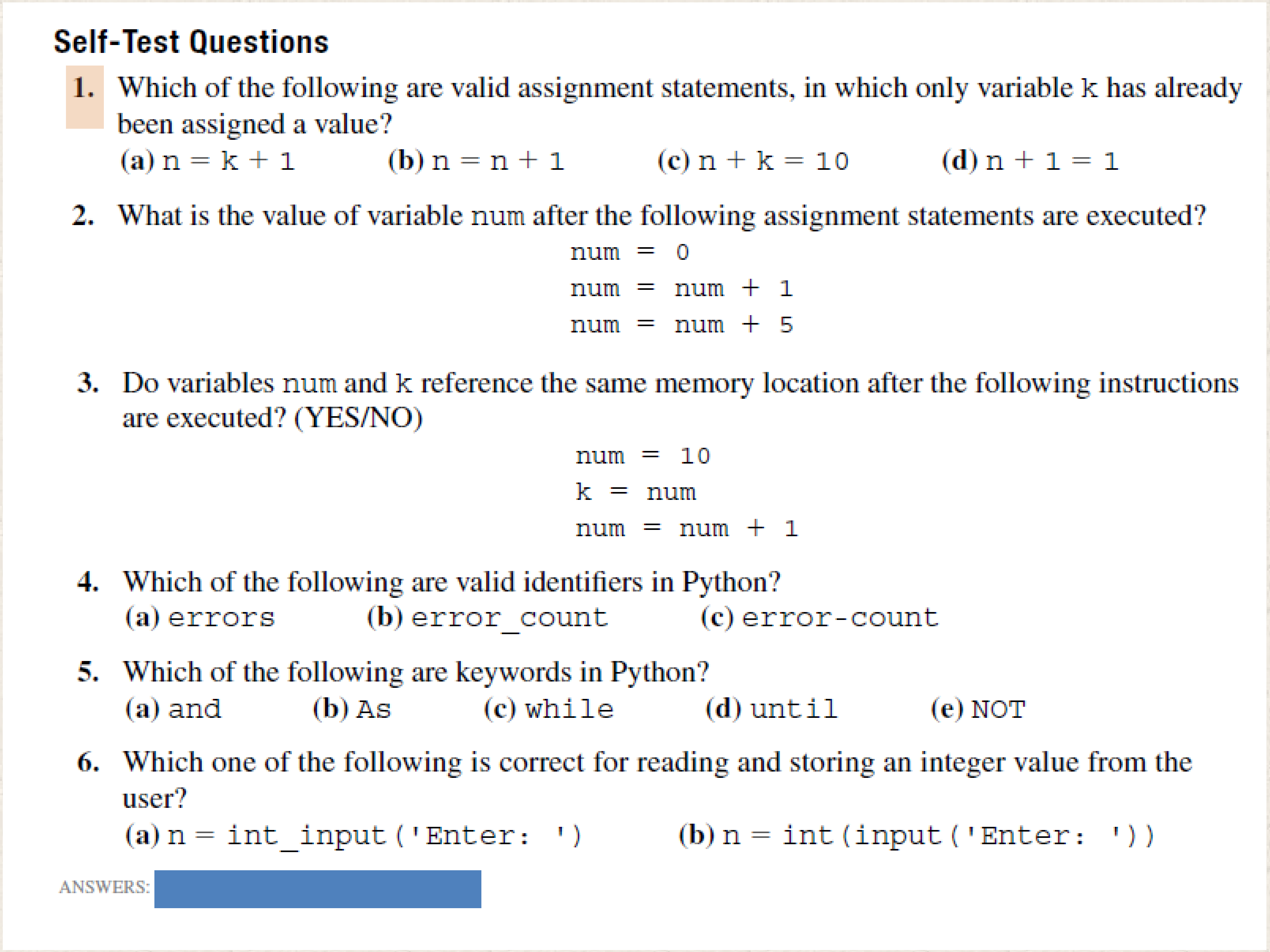
Example Execution

```
 1  # Restaurant Tab Calculation Program
 2  # This program will calculate a restaurant tab with a gift certificate
 3
 4  # initialization
 5  tax = 0.08
 6
 7  # program greeting
 8  print('This program will calculate a restaurant tab for a couple with')
 9  print('a gift certificate, with a restaurant tax of', tax * 100, '%\n')
10
11  # get amount of gift certificate
12  amt_certificate = float(input('Enter amount of the gift certificate: '))
13
14  # cost of ordered items
15  print('Enter ordered items for person 1')
16
17  appetizer_per1 = float(input('Appetizier: '))
18  entree_per1 = float(input('Entree: '))
19  drinks_per1 = float(input('Drinks: '))
20  dessert_per1 = float(input('Dessert: '))
21
22  print('\nEnter ordered items for person 2')
23
24  appetizer_per2 = float(input('Appetizier: '))
25  entree_per2 = float(input('Entree: '))
26  drinks_per2 = float(input('Drinks: '))
27  dessert_per2 = float(input('Dessert: '))
28
29  # total items
30  amt_person1 = appetizer_per1 + entree_per1 + drinks_per1 + dessert_per1
31  amt_person2 = appetizer_per2 + entree_per2 + drinks_per2 + dessert_per2
32
33  # compute tab with tax
34  items_cost = amt_person1 + amt_person2
35  tab = items_cost + items_cost * tax
36
37  # display amount owe
38  print('\nOrdered items: $', format(items_cost, '.2f'))
39  print('Restaurant tax: $', format(items_cost * tax, '.2f'))
40  print('Tab: $', format(tab - amt_certificate, '.2f'))
41  print('(negative amount indicates unused amount of gift certificate)')
```

Line 5 provides the required initialization of variables, with variable `tax` assigned to 8% (.08) used in **lines 9**, **35**, and **39**. To change the restaurant tax, only this line of the program needs to be changed.

Lines 8–9 display what the program does. **Lines 30** and **31** total the cost of the orders for each person, assigned to variables `amt_person1` and `amt_person2`. **Lines 34** and **35** compute the tab, including the tax (stored in variable `tab`).

Finally, **lines 38–41** display the cost of the ordered items, followed by the added restaurant tax and the amount due after deducting the amount of the gift certificate.



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?
   (a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)

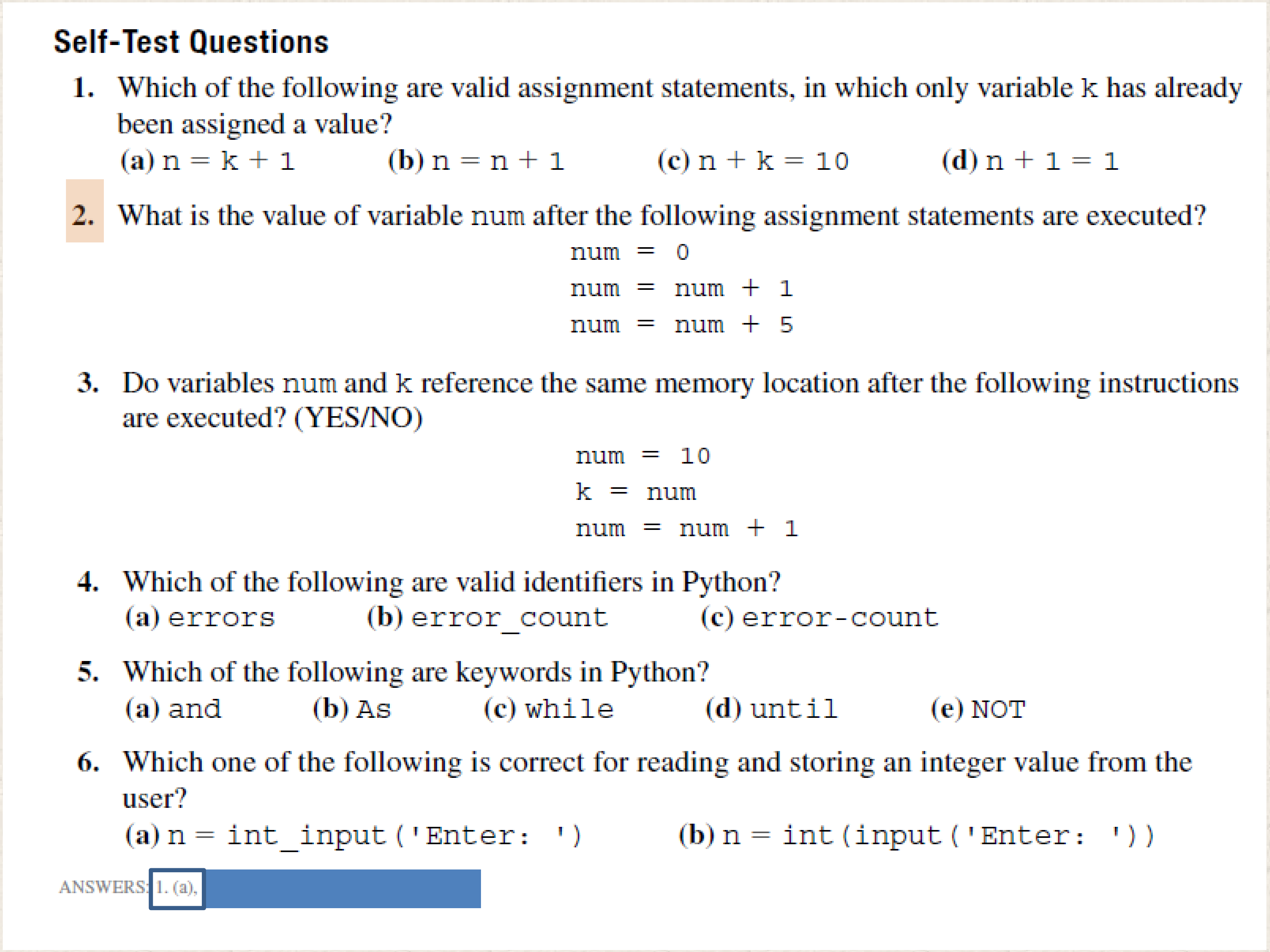
```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors` **(b)** `error_count` **(c)** `error-count`

5. Which of the following are keywords in Python?
   (a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: [redacted]



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable k has already been assigned a value?
   (a) `n = k + 1`　　(b) `n = n + 1`　　(c) `n + k = 10`　　(d) `n + 1 = 1`

2. What is the value of variable num after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables num and k reference the same memory location after the following instructions are executed? (YES/NO)

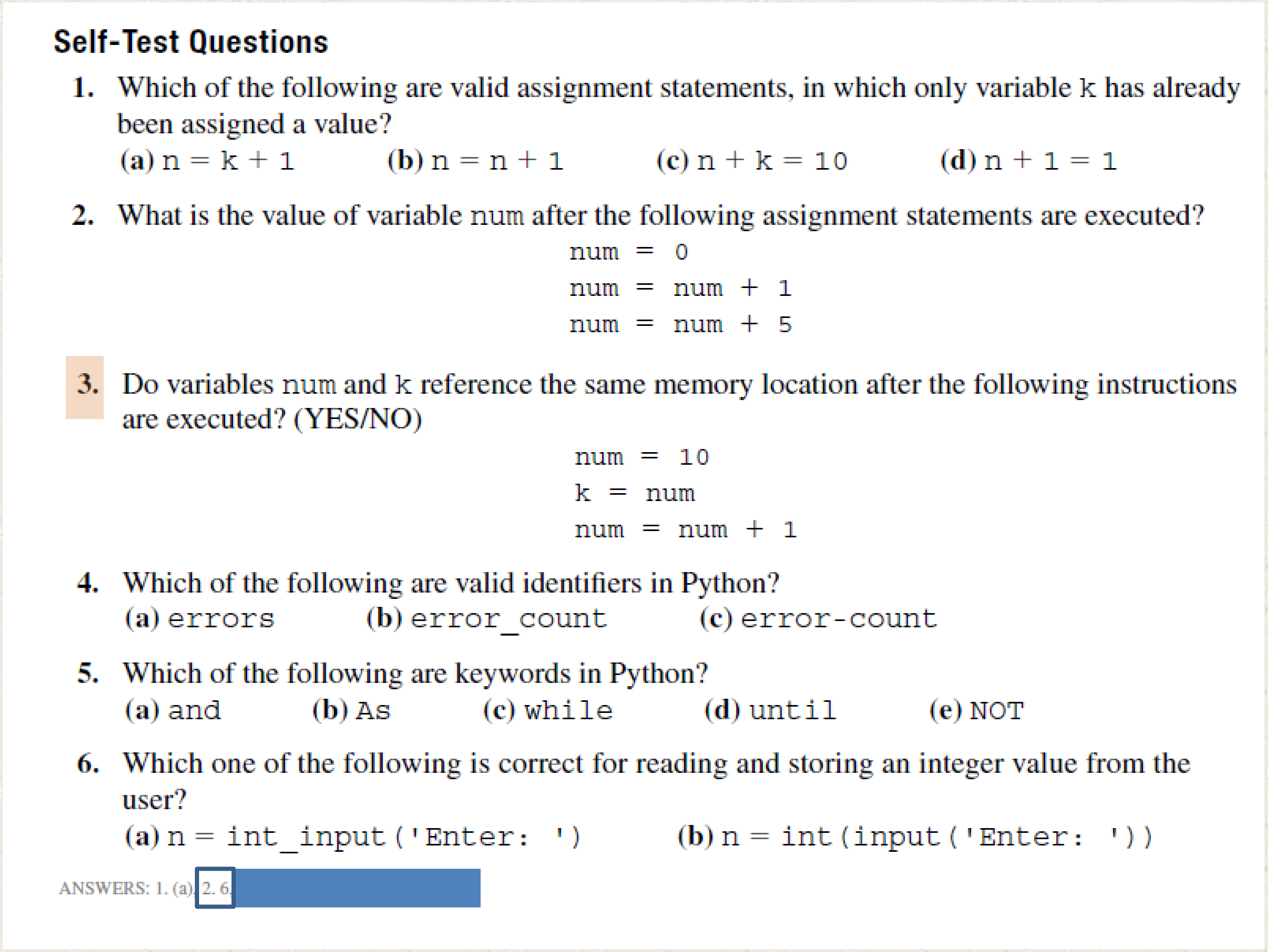
```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors`　　(b) `error_count`　　(c) `error-count`

5. Which of the following are keywords in Python?
   (a) `and`　　(b) `As`　　(c) `while`　　(d) `until`　　(e) `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')`　　(b) `n = int(input('Enter: '))`

ANSWERS: 1. (a), [redacted]



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?
   (a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)

```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors` **(b)** `error_count` **(c)** `error-count`

5. Which of the following are keywords in Python?
   (a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: 1. (a), 2. 6, [redacted]

Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?

(a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)

```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?

(a) `errors` **(b)** `error_count` **(c)** `error-count`

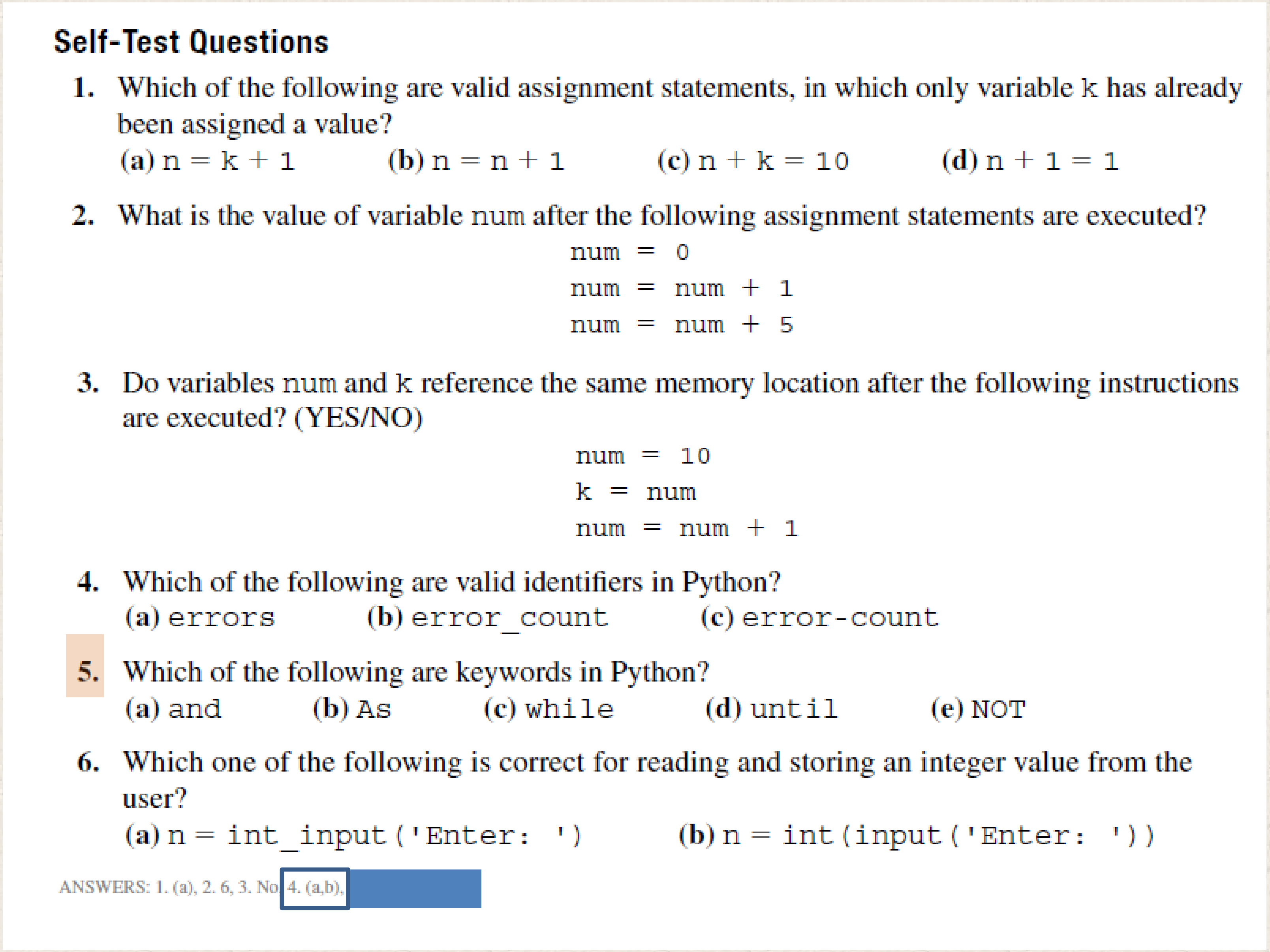
5. Which of the following are keywords in Python?

(a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?

(a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: 1. (a), 2. 6, 3. No, [redacted]



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?
   (a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)

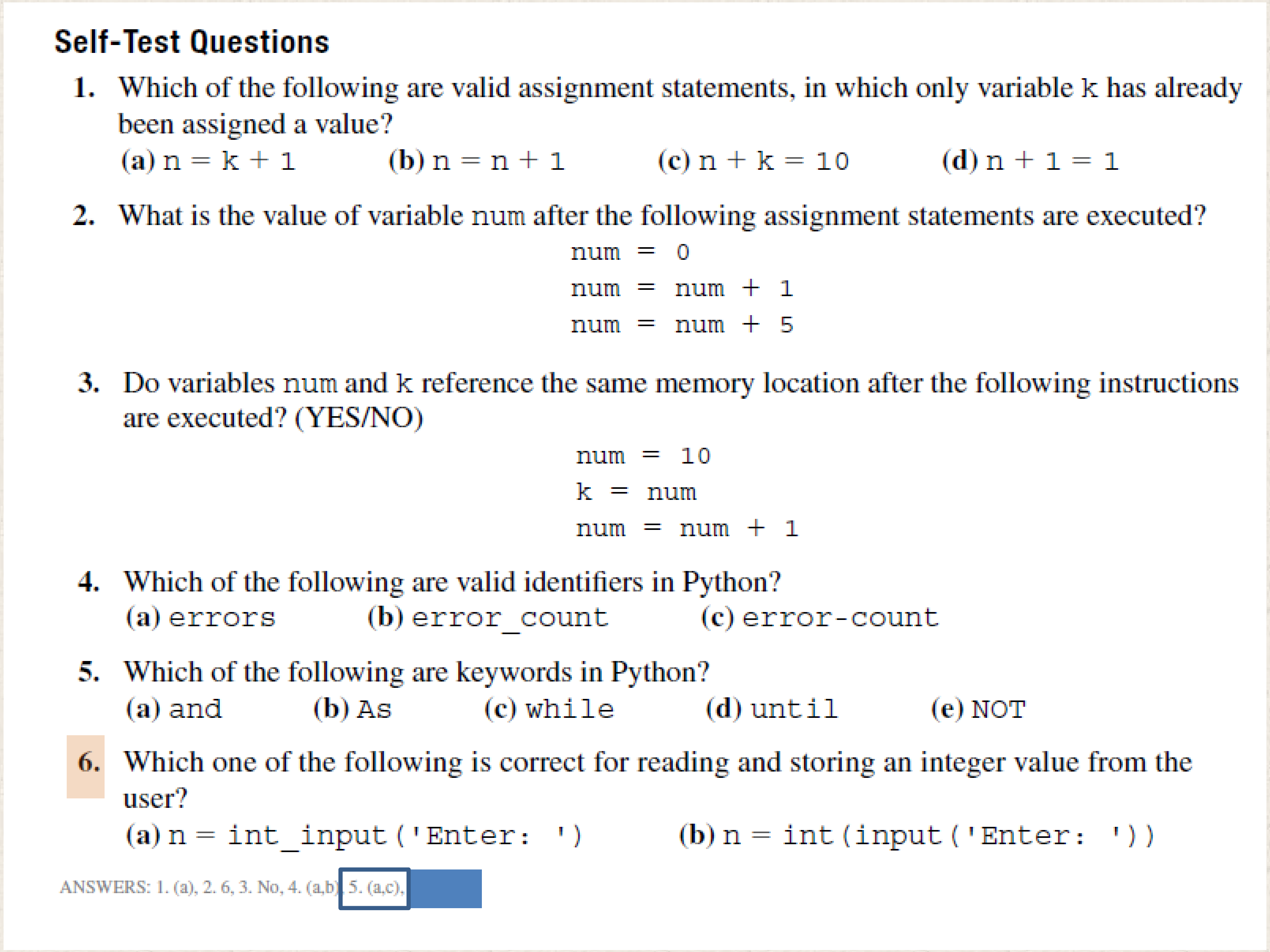
```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors` **(b)** `error_count` **(c)** `error-count`

5. Which of the following are keywords in Python?
   (a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: 1. (a), 2. 6, 3. No 4. (a,b), [redacted]



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?
   (a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?
```
num = 0
num = num + 1
num = num + 5
```

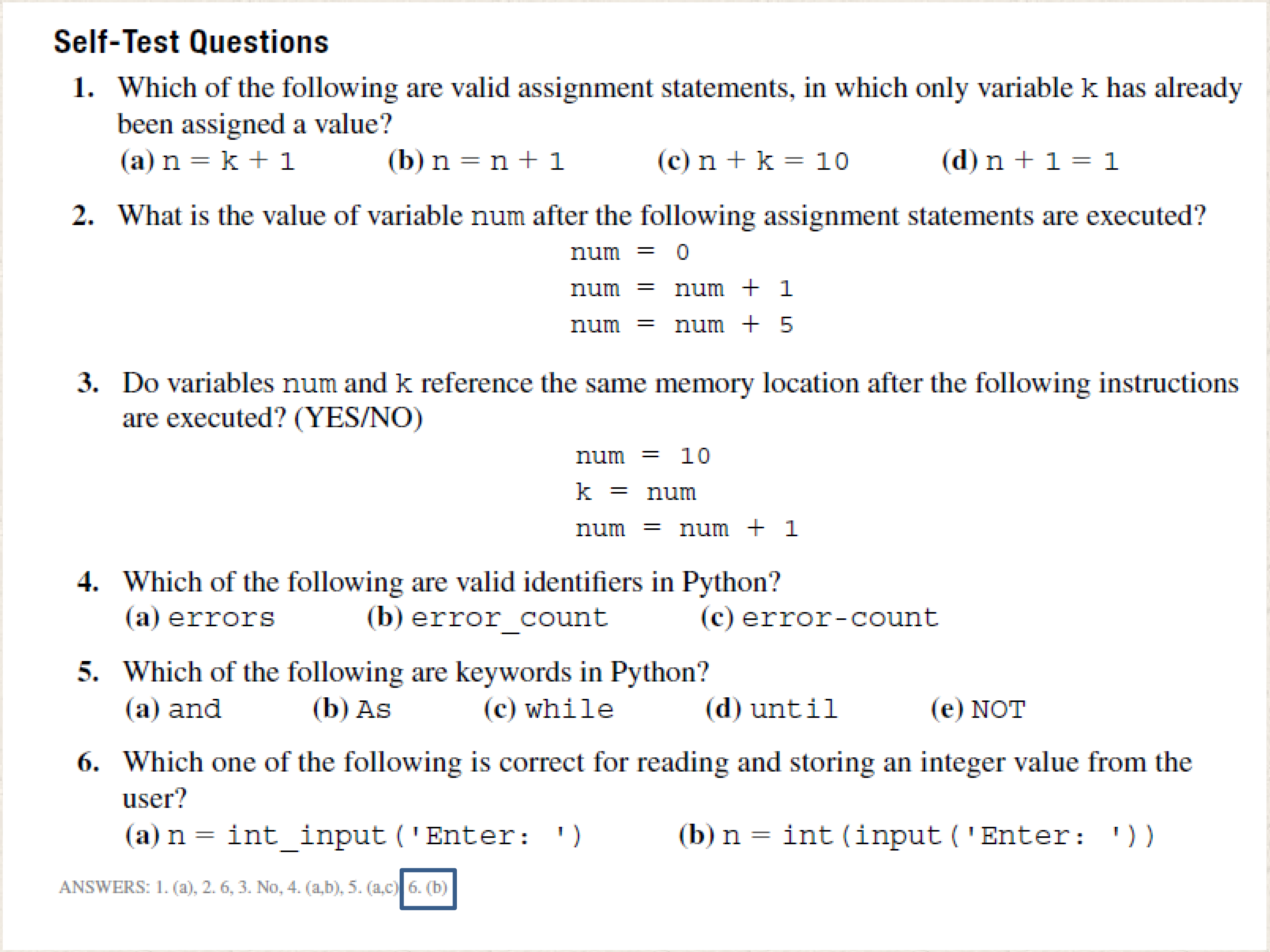
3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)
```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors` **(b)** `error_count` **(c)** `error-count`

5. Which of the following are keywords in Python?
   (a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: 1. (a), 2. 6, 3. No, 4. (a,b), 5. (a,c), [redacted]



Self-Test Questions

1. Which of the following are valid assignment statements, in which only variable `k` has already been assigned a value?
   (a) `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

2. What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

3. Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES/NO)

```
num = 10
k = num
num = num + 1
```

4. Which of the following are valid identifiers in Python?
   (a) `errors` **(b)** `error_count` **(c)** `error-count`

5. Which of the following are keywords in Python?
   (a) `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

6. Which one of the following is correct for reading and storing an integer value from the user?
   (a) `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

ANSWERS: 1. (a), 2. 6, 3. No, 4. (a,b), 5. (a,c) 6. (b)

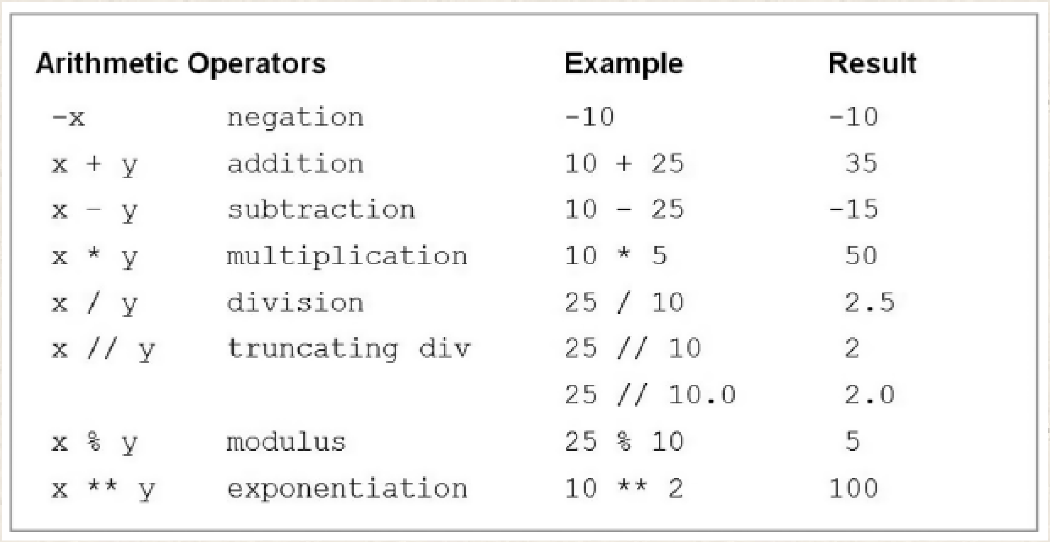
Operators

An **operator** **is a symbol that represents an operation that may be performed on one or more *operands***. For example, the **+** symbol represents the operation of addition. **An** **operand** **is a value that a given operator is applied to,** such as operands 2 and 3 in the expression 2 + 3.

A **unary operator** operates on only one operand, such as the negation operator in the expression - 12.

A **binary operator** **operates on two operands**, as with the addition operator. Most operators in programming languages are binary operators. We look at the arithmetic operators in Python next.

Arithmetic Operators



Arithmetic Operators		Example	Result
`-x`	negation	`-10`	`-10`
`x + y`	addition	`10 + 25`	`35`
`x - y`	subtraction	`10 - 25`	`-15`
`x * y`	multiplication	`10 * 5`	`50`
`x / y`	division	`25 / 10`	`2.5`
`x // y`	truncating div	`25 // 10`	`2`
		`25 // 10.0`	`2.0`
`x % y`	modulus	`25 % 10`	`5`
`x ** y`	exponentiation	`10 ** 2`	`100`

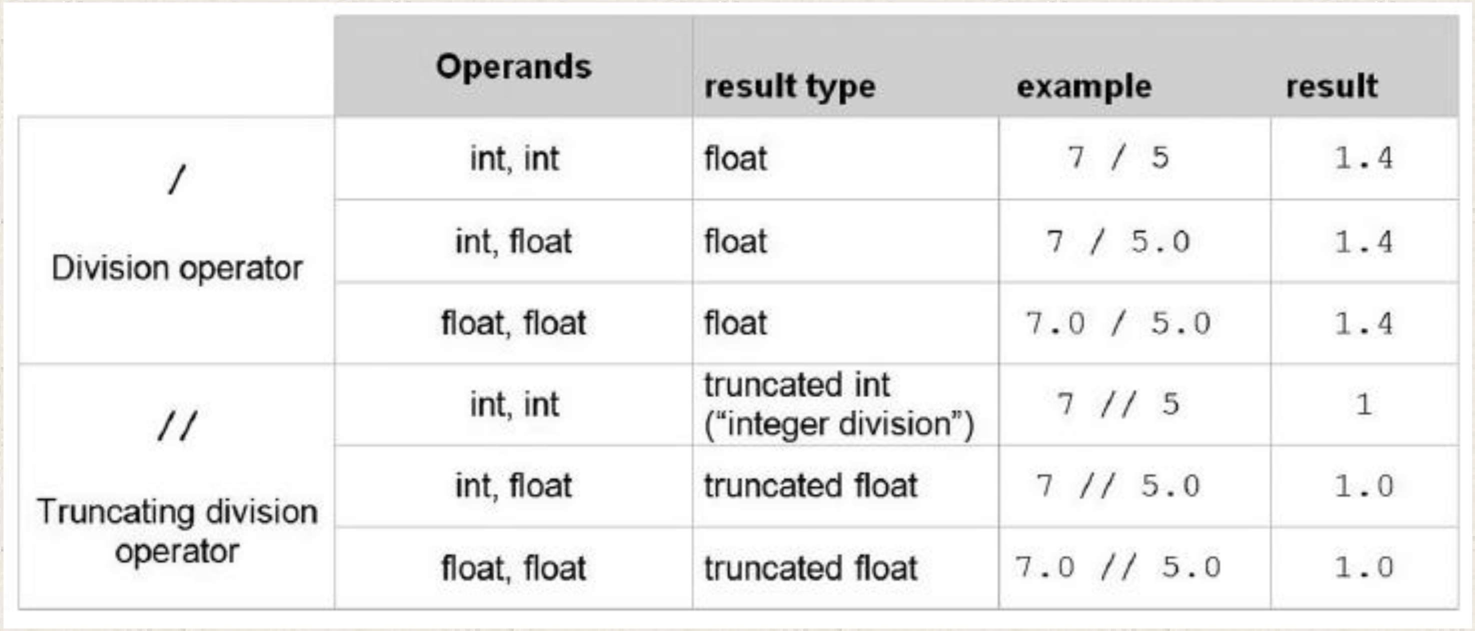
The **+** , **-**, ***** (**multiplication**) and **/** (**division**) arithmetic operators perform the usual operations. Note that the **–** symbol is used both as a unary operator (for negation) and a binary operator (for subtraction),

```
20 - 5 → 15            (– as binary operator)
- 10 * 2 → - 20        (– as unary operator)
```

Python also includes an **exponentiation** (******) operator.

Python provides two forms of division. **"True" division** is denoted by a single slash, **/**. Thus, **25 / 10 evaluates to 2.5**. **Truncating division** is denoted by a double slash, **//**, providing a truncated result based on the type of operands applied to.

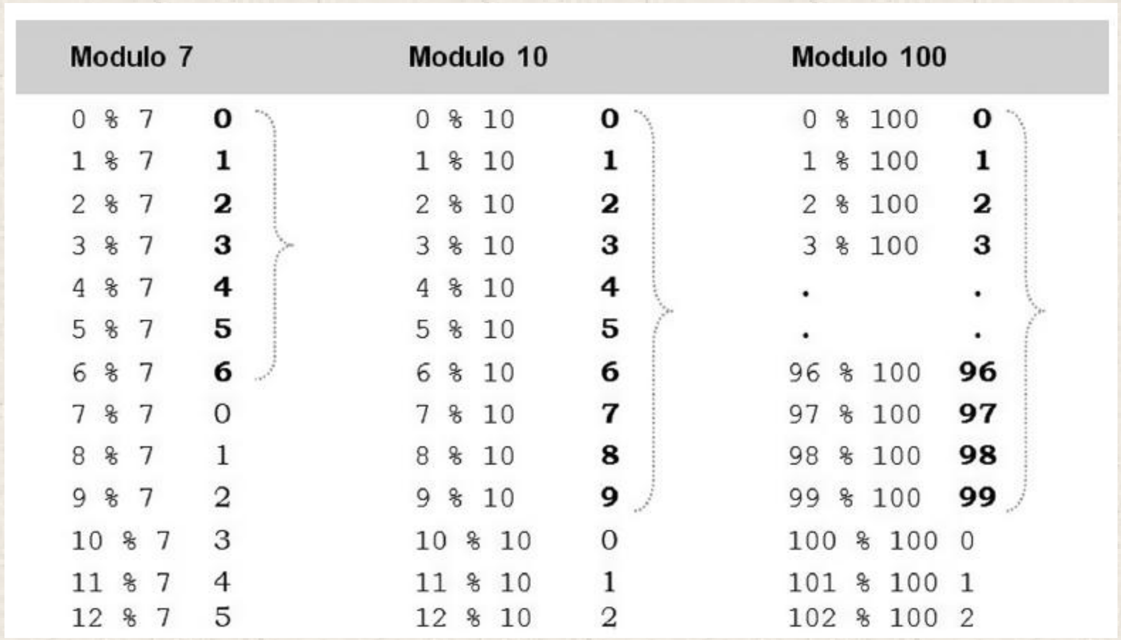
When both operands are integer values, the result is a truncated integer referred to as **integer division**. When as least one of the operands is a float type, the result is a **truncated floating point**. Thus, **25 // 10 evaluates to 2**, while **25.0 // 10 evaluates to 2.0**.



	Operands	result type	example	result
/ Division operator	int, int	float	`7 / 5`	`1.4`
	int, float	float	`7 / 5.0`	`1.4`
	float, float	float	`7.0 / 5.0`	`1.4`
// Truncating division operator	int, int	truncated int ("integer division")	`7 // 5`	`1`
	int, float	truncated float	`7 // 5.0`	`1.0`
	float, float	truncated float	`7.0 // 5.0`	`1.0`

As example use of integer division, the number of dozen doughnuts for variable `numDoughnuts` = 29 is: `numDoughnuts // 12` → `29 // 12` → `2`

Lastly, the **modulus operator** (`%`) **gives the remainder of the division of its operands**, resulting in a cycle of values.

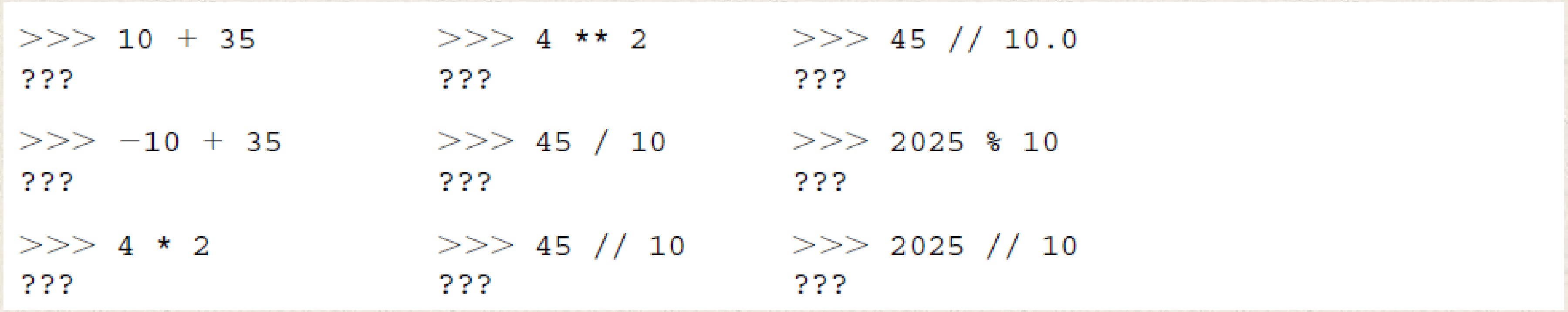


Modulo 7		Modulo 10		Modulo 100	
0 % 7	**0**	0 % 10	**0**	0 % 100	**0**
1 % 7	**1**	1 % 10	**1**	1 % 100	**1**
2 % 7	**2**	2 % 10	**2**	2 % 100	**2**
3 % 7	**3**	3 % 10	**3**	3 % 100	**3**
4 % 7	**4**	4 % 10	**4**	.	.
5 % 7	**5**	5 % 10	**5**	.	.
6 % 7	**6**	6 % 10	**6**	96 % 100	**96**
7 % 7	0	7 % 10	**7**	97 % 100	**97**
8 % 7	1	8 % 10	**8**	98 % 100	**98**
9 % 7	2	9 % 10	**9**	99 % 100	**99**
10 % 7	3	10 % 10	0	100 % 100	0
11 % 7	4	11 % 10	1	101 % 100	1
12 % 7	5	12 % 10	2	102 % 100	2

The modulus and truncating (integer) division operators are complements of each other. For example, `29 // 12` **gives the number of dozen doughnuts**, while `29 % 12` **gives the number of leftover doughnuts** (5).

Let’s Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> 10 + 35          >>> 4 ** 2        >>> 45 // 10.0
???                  ???               ???

>>> -10 + 35         >>> 45 / 10       >>> 2025 % 10
???                  ???               ???

>>> 4 * 2            >>> 45 // 10      >>> 2025 // 10
???                  ???               ???
```

Let’s Apply It

Your Place in the Universe

The following program calculates the approximate number of atoms that the average person contains, and the percentage of the universe that they comprise.

This program utilizes the following programming features:

- ➢ **floating-point scientific notation**
- ➢ **built-in format function**

Program Execution ...

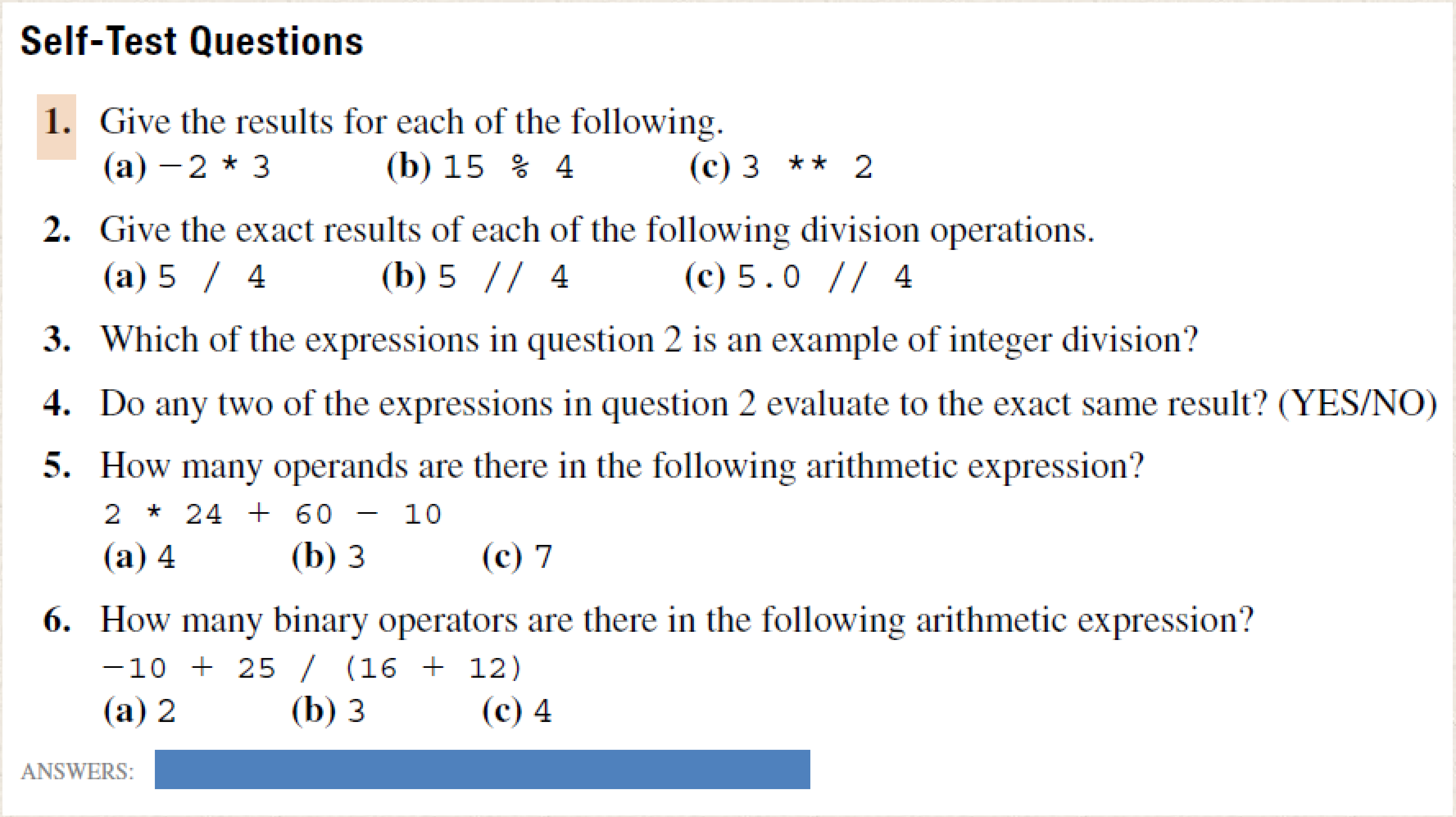
```
This program will determine your place in the universe.
Enter your weight in pounds: 150
You contain approximately 3.30e+28 atoms
Therefore, you comprise 3.30e-51 % of the universe
```

```
 1  # Your Place in the Universe Program
 2
 3  # This program will determine the approximate number of atoms that a
 4  # person consists of and the percent of the universe that they comprise.
 5
 6  # initialization
 7  num_atoms_universe = 10e80
 8  weight_avg_person = 70  # 70 kg (154 lbs)
 9  num_atoms_avg_person = 7e27
10
11  # program greeting
12  print('This program will determine your place in the universe.')
13
14  # prompt for user's weight
15  weight_lbs = int(input('Enter your weight in pounds: '))
16
17  # convert weight to kilograms
18  weight_kg = 2.2 * weight_lbs
19
20  # determine number atoms in person
21  num_atoms = (weight_kg / 70) * num_atoms_avg_person
22  percent_of_universe = (num_atoms / num_atoms_universe) * 100
23
24  # display results
25  print('You contain approximately', format(num_atoms, '.2e'), 'atoms')
26  print('Therefore, you comprise', format(percent_of_universe, '.2e'),
27        '% of the universe')
```

Needed variables **num_atoms_universe**, **weight_avg_person**, and **num_atoms_avg_person** are initialized in **lines 7–9** . The program greeting is on **line 12** .

Line 15 inputs the person's weight. **Line 18** converts the weight to kilograms for the calculations on **lines 21–22,** which compute the desired results.

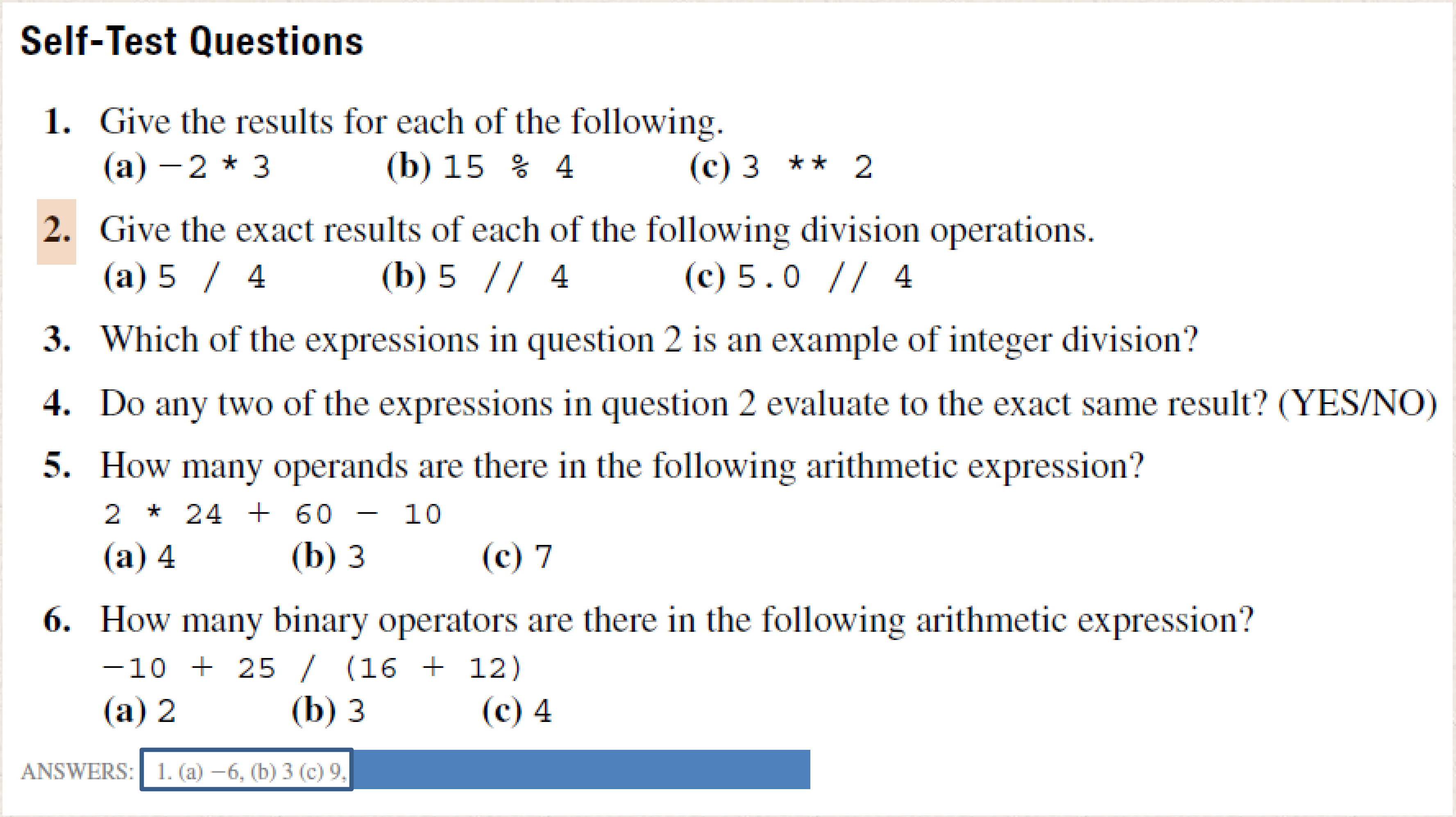
Finally, **lines 25–27** display the results.



Self-Test Questions

1. Give the results for each of the following.
   (a) `−2 * 3` (b) `15 % 4` (c) `3 ** 2`
2. Give the exact results of each of the following division operations.
   (a) `5 / 4` (b) `5 // 4` (c) `5.0 // 4`
3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 − 10`
   (a) 4 (b) 3 (c) 7
6. How many binary operators are there in the following arithmetic expression?
   `−10 + 25 / (16 + 12)`
   (a) 2 (b) 3 (c) 4

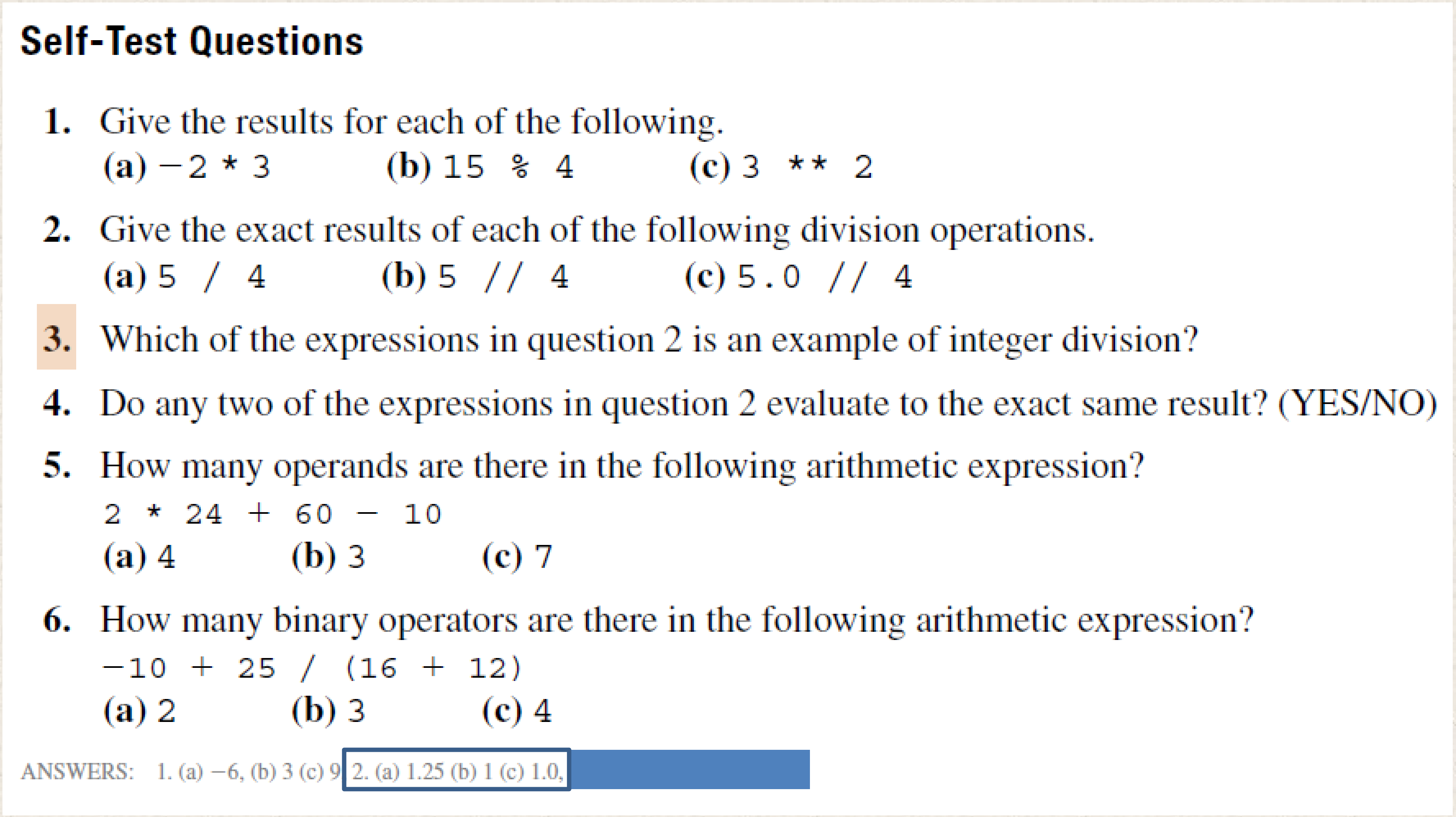
ANSWERS: [redacted]



Self-Test Questions

1. Give the results for each of the following.
   (a) `−2 * 3` **(b)** `15 % 4` **(c)** `3 ** 2`
2. Give the exact results of each of the following division operations.
   (a) `5 / 4` **(b)** `5 // 4` **(c)** `5.0 // 4`
3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 − 10`
   (a) 4 **(b)** 3 **(c)** 7
6. How many binary operators are there in the following arithmetic expression?
   `−10 + 25 / (16 + 12)`
   (a) 2 **(b)** 3 **(c)** 4

ANSWERS: 1. (a) −6, (b) 3 (c) 9, [redacted]



Self-Test Questions

1. Give the results for each of the following.
   (a) `−2 * 3` **(b)** `15 % 4` **(c)** `3 ** 2`

2. Give the exact results of each of the following division operations.
   (a) `5 / 4` **(b)** `5 // 4` **(c)** `5.0 // 4`

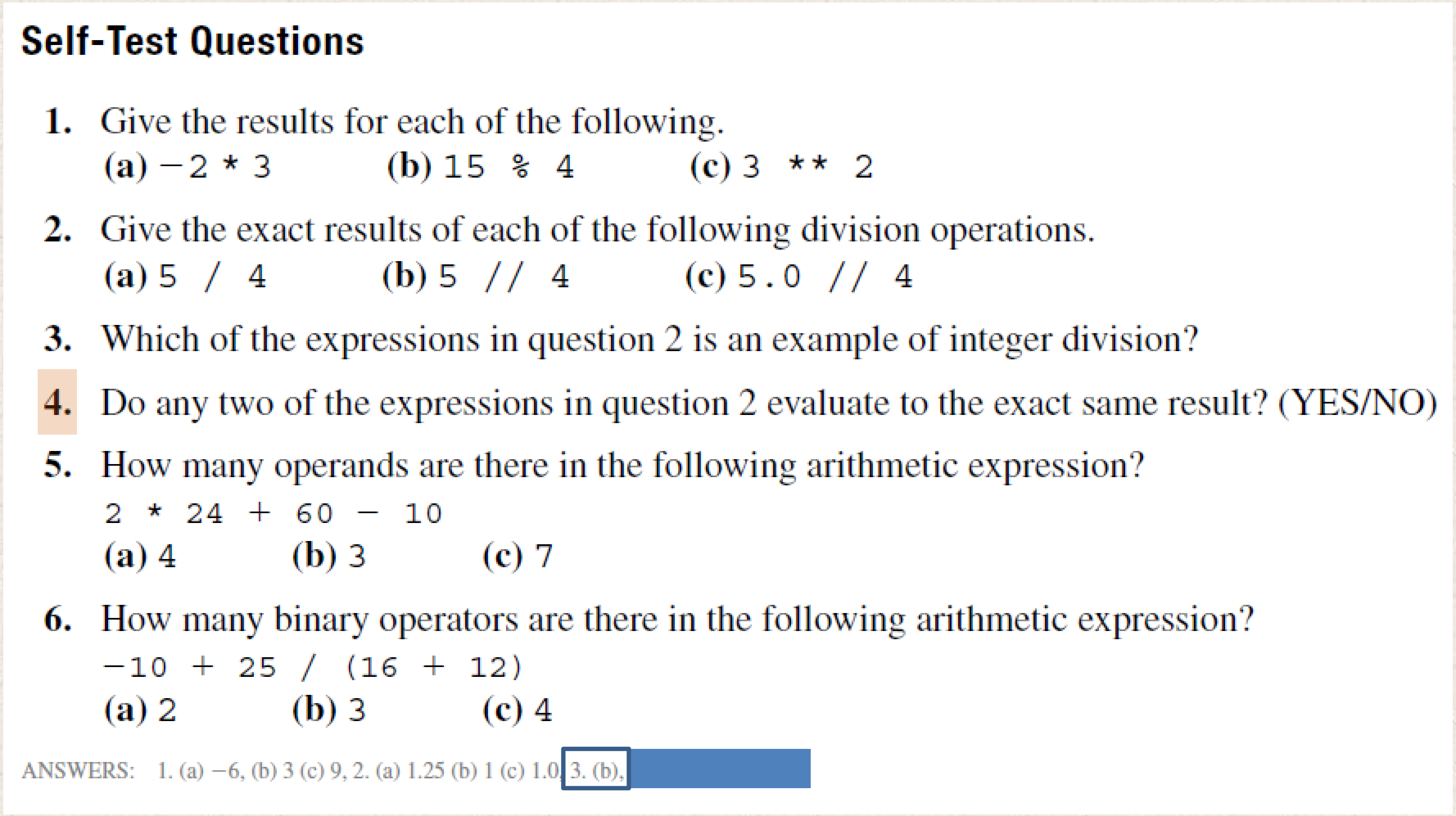
3. Which of the expressions in question 2 is an example of integer division?

4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)

5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 − 10`
   (a) 4 **(b)** 3 **(c)** 7

6. How many binary operators are there in the following arithmetic expression?
   `−10 + 25 / (16 + 12)`
   (a) 2 **(b)** 3 **(c)** 4

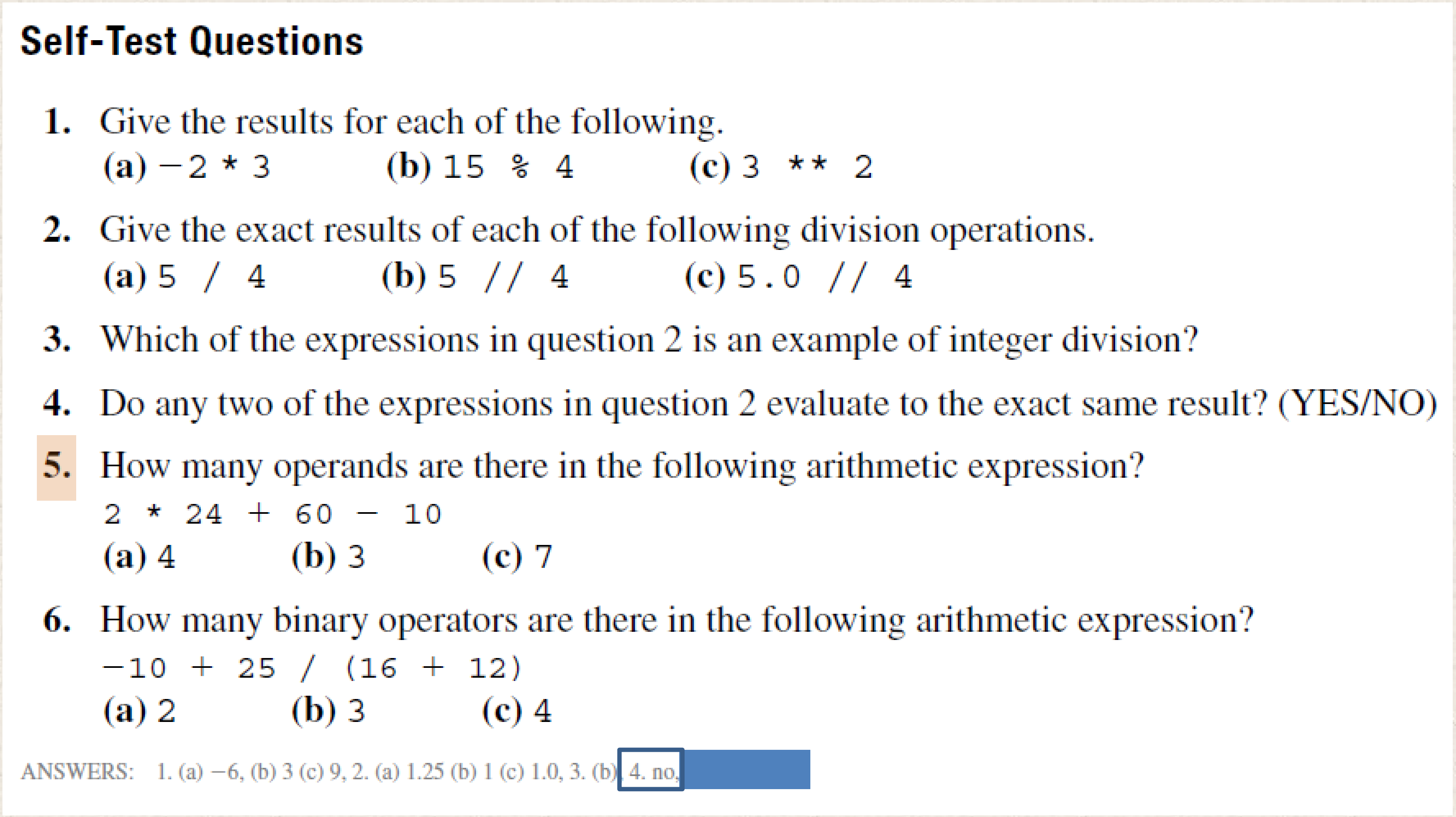
ANSWERS: 1. (a) −6, (b) 3 (c) 9 2. (a) 1.25 (b) 1 (c) 1.0, [redacted]



Self-Test Questions

1. Give the results for each of the following.
   (a) `−2 * 3` (b) `15 % 4` (c) `3 ** 2`
2. Give the exact results of each of the following division operations.
   (a) `5 / 4` (b) `5 // 4` (c) `5.0 // 4`
3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 − 10`
   (a) 4 (b) 3 (c) 7
6. How many binary operators are there in the following arithmetic expression?
   `−10 + 25 / (16 + 12)`
   (a) 2 (b) 3 (c) 4

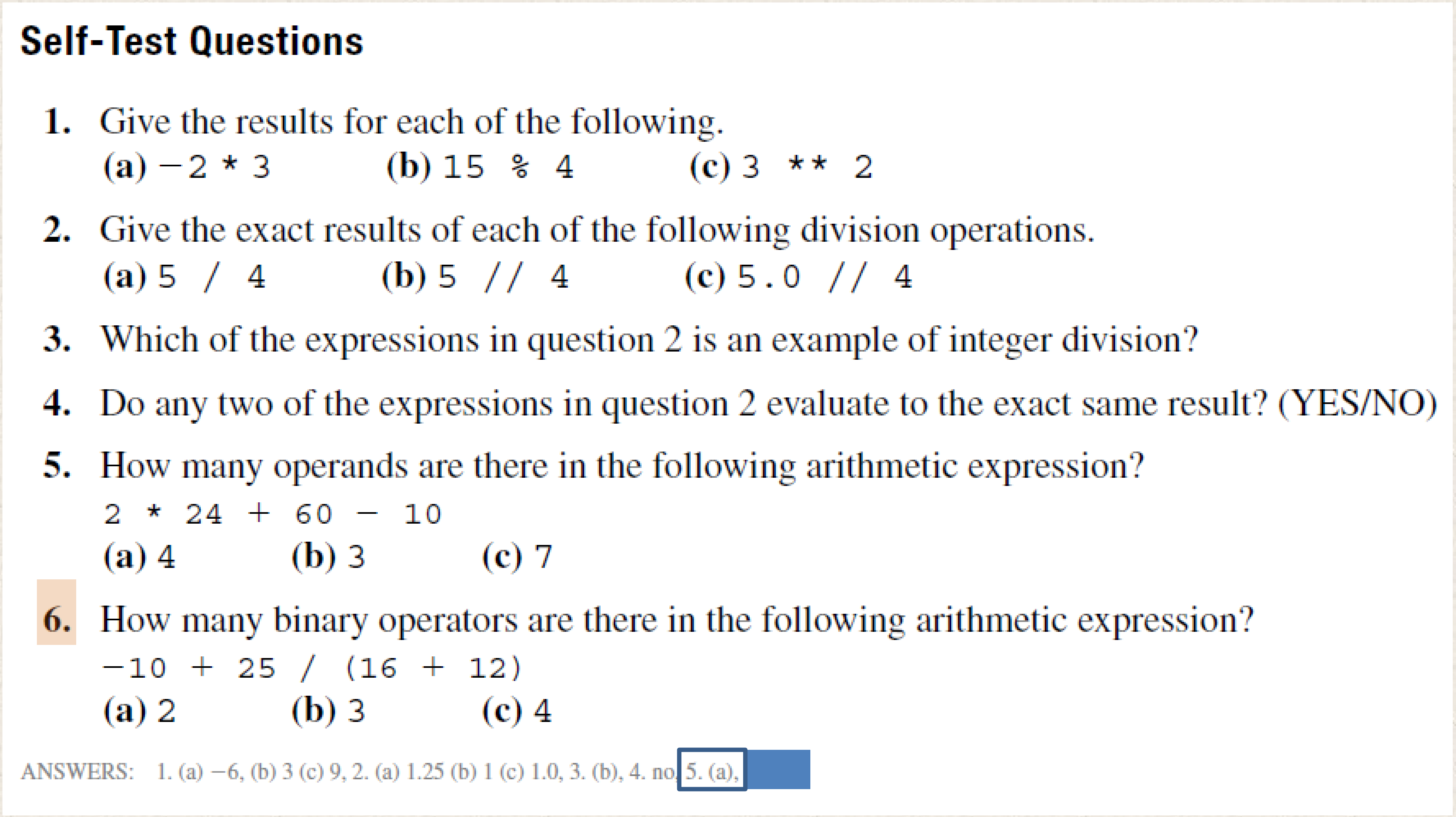
ANSWERS: 1. (a) −6, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, $\boxed{\text{3. (b),}}$ [redacted]



Self-Test Questions

1. Give the results for each of the following.
   (a) `-2 * 3` (b) `15 % 4` (c) `3 ** 2`
2. Give the exact results of each of the following division operations.
   (a) `5 / 4` (b) `5 // 4` (c) `5.0 // 4`
3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 - 10`
   (a) 4 (b) 3 (c) 7
6. How many binary operators are there in the following arithmetic expression?
   `-10 + 25 / (16 + 12)`
   (a) 2 (b) 3 (c) 4

ANSWERS: 1. (a) −6, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, 3. (b), 4. no, [redacted]



Self-Test Questions

1. Give the results for each of the following.
(a) `−2 * 3` **(b)** `15 % 4` **(c)** `3 ** 2`

2. Give the exact results of each of the following division operations.
(a) `5 / 4` **(b)** `5 // 4` **(c)** `5.0 // 4`

3. Which of the expressions in question 2 is an example of integer division?

4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)

5. How many operands are there in the following arithmetic expression?

```
2 * 24 + 60 − 10
```

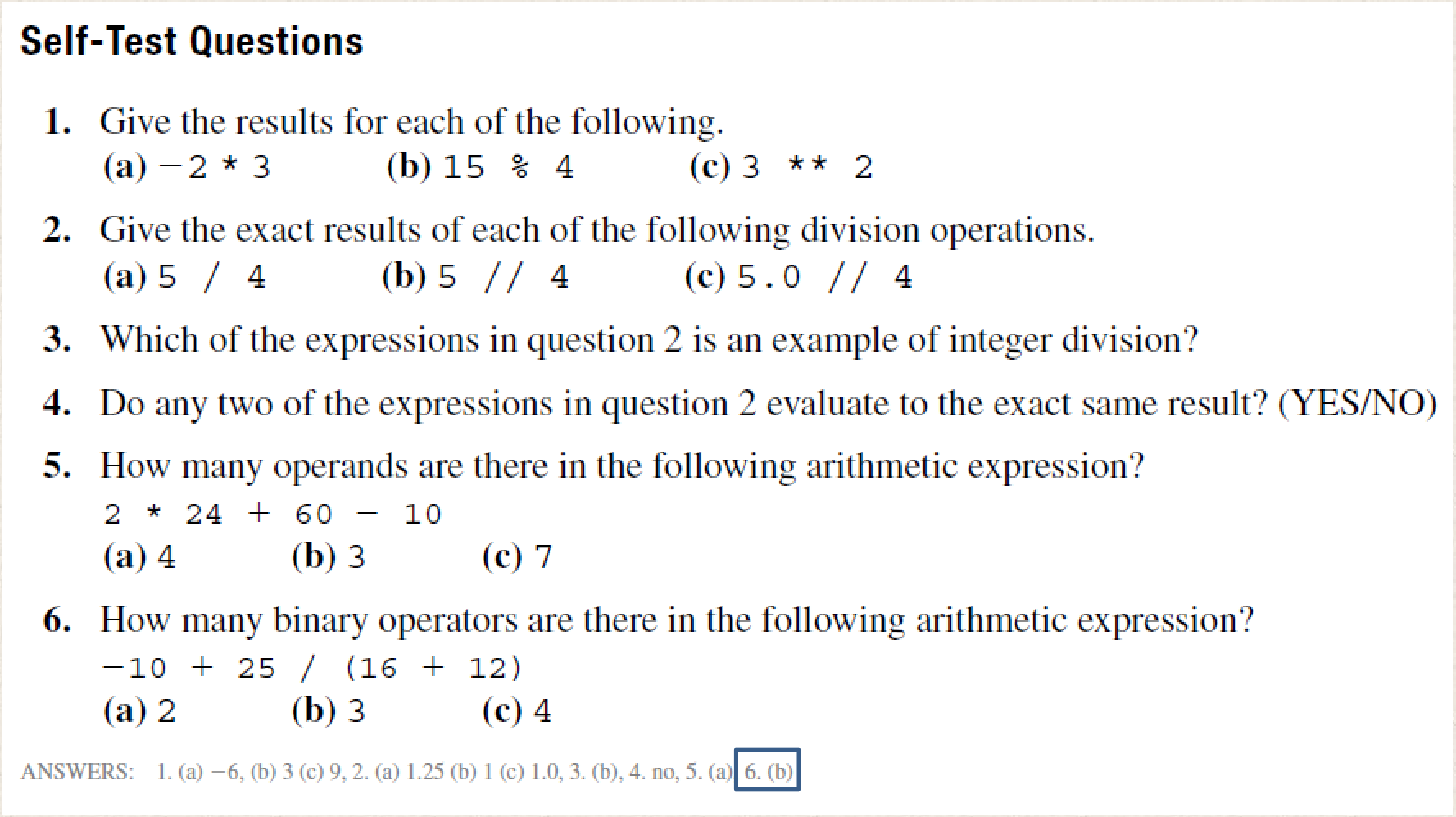
(a) 4 **(b)** 3 **(c)** 7

6. How many binary operators are there in the following arithmetic expression?

```
−10 + 25 / (16 + 12)
```

(a) 2 **(b)** 3 **(c)** 4

ANSWERS: 1. (a) −6, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, 3. (b), 4. no, 5. (a), [redacted]



Self-Test Questions

1. Give the results for each of the following.
   (a) `−2 * 3` (b) `15 % 4` (c) `3 ** 2`
2. Give the exact results of each of the following division operations.
   (a) `5 / 4` (b) `5 // 4` (c) `5.0 // 4`
3. Which of the expressions in question 2 is an example of integer division?
4. Do any two of the expressions in question 2 evaluate to the exact same result? (YES/NO)
5. How many operands are there in the following arithmetic expression?
   `2 * 24 + 60 − 10`
   (a) 4 (b) 3 (c) 7
6. How many binary operators are there in the following arithmetic expression?
   `−10 + 25 / (16 + 12)`
   (a) 2 (b) 3 (c) 4

ANSWERS: 1. (a) −6, (b) 3 (c) 9, 2. (a) 1.25 (b) 1 (c) 1.0, 3. (b), 4. no, 5. (a) $\boxed{\text{6. (b)}}$

Expressions and Data Types

Now that we have looked at arithmetic operators, **we will see how operators and operands can be combined to form *expressions***. In particular, we will look at how arithmetic expressions are evaluated in Python. We also introduce the notion of a ***data type***.

What Is an Expression?

An expression is a combination of symbols that evaluates to a value. Expressions, most commonly, consist of a combination of operators and operands,

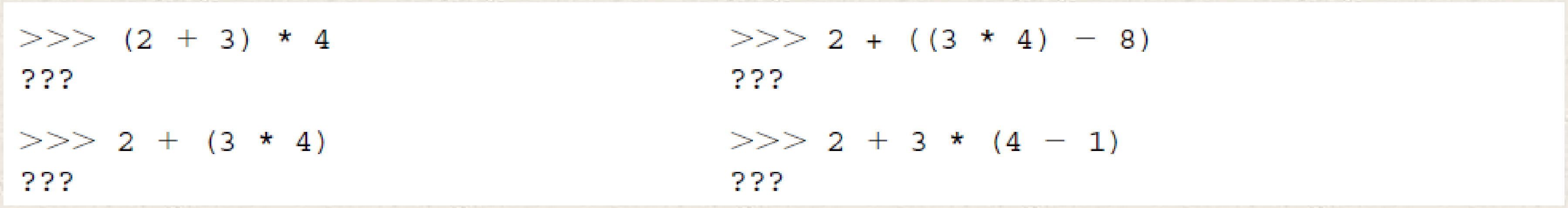
4 + (3 * k)

An expression **can also consist of a single literal or variable**. Thus, 4, 3, and k are each expressions. This expression has two ***subexpressions**, 4 and* (3 * k). Subexpression (3 * k) itself has two subexpressions, 3 and k.

Expressions that evaluate to a numeric type are called arithmetic expressions. A subexpression is any expression that is part of a larger expression. **Subexpressions may be denoted by the use of parentheses**, as shown above. Thus, for the expression 4 + (3 * 2), the two operands of the addition operator are 4 and (3 * 2), and thus the result is equal to 10. If the expression were instead written as (4 + 3) * 2, then it would evaluate to 14.

Let's Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> (2 + 3) * 4
???

>>> 2 + (3 * 4)
???
```

```
>>> 2 + ((3 * 4) - 8)
???

>>> 2 + 3 * (4 - 1)
???
```

Operator Precedence

The way we commonly represent expressions, in which operators appear between their operands, is referred to as infix notation. For example, the expression 4 + 3 is in infix notation since the + operator appears between its two operands, 4 and 3. There are other ways of representing expressions called **prefix** and **postfix notation**, in which operators are placed before and after their operands, respectively.

The expression 4 + (3 * 5) is also in infix notation. It contains two operators, + and *. The parentheses denote that (3 * 5) is a subexpression. Therefore, 4 and (3 * 5) are the operands of the addition operator, and thus the overall expression evaluates to 19. What if the parentheses were omitted, as given below?

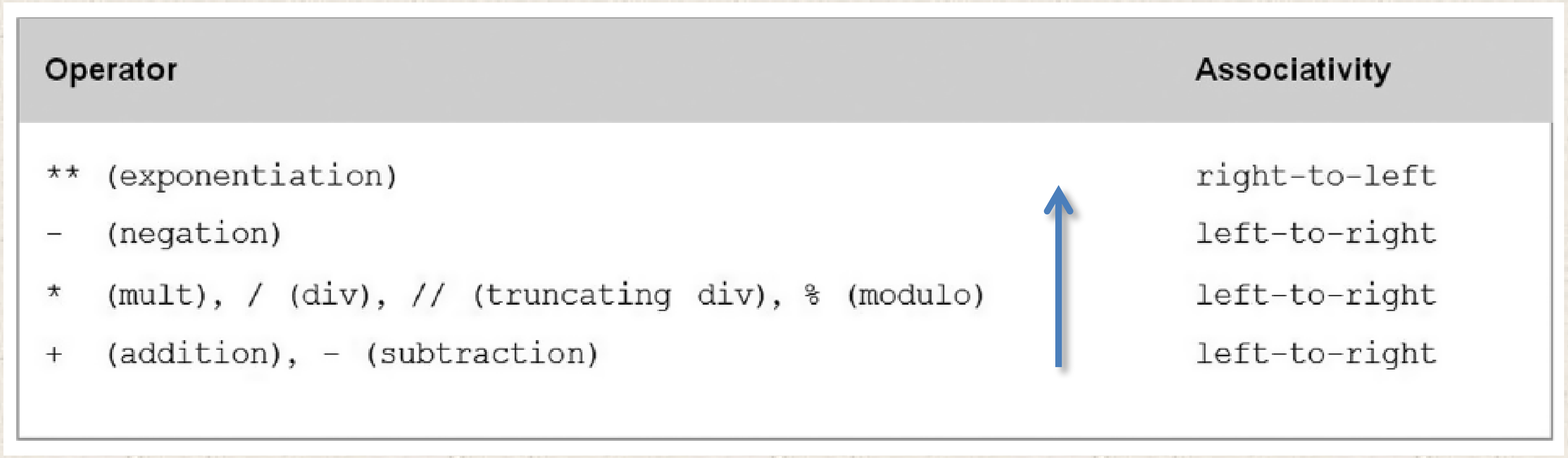
4 + 3 * 5

How would this be evaluated? These are two possibilities.

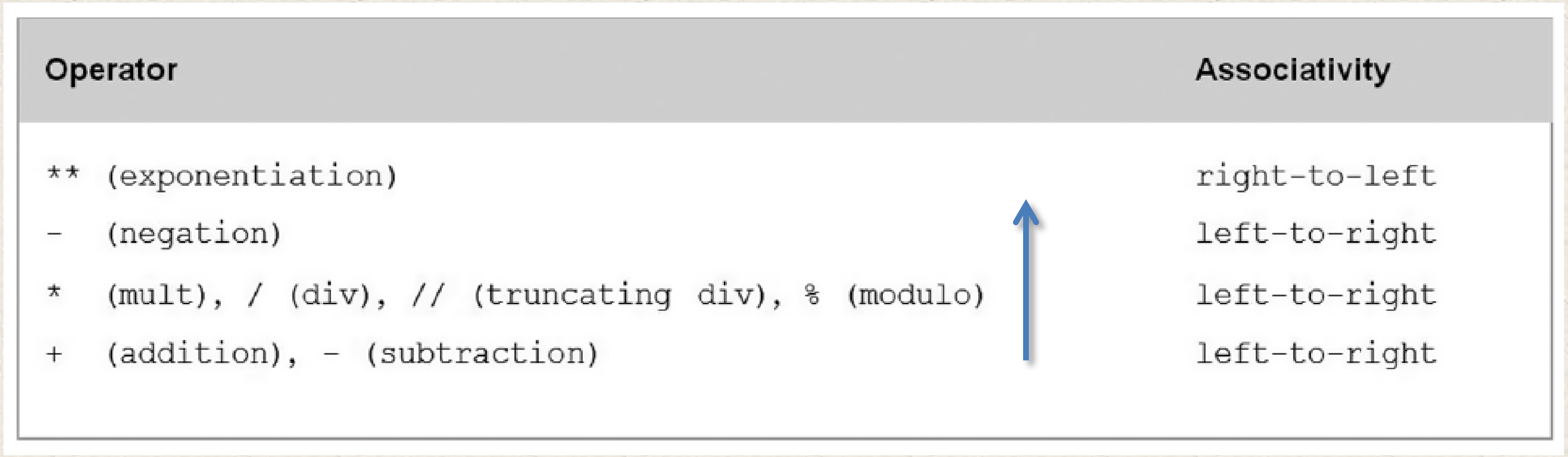
4 + 3 * 5 → 4 + 15 → **19**   4 + 3 * 5 → 7 * 5 → **35**

Some might say that the first version is the correct one by the conventions of mathematics. However, **each programming language has its own rules for the order that operators are applied, called operator precedence**, defined in an **operator precedence table**. This may or may not be the same as in mathematics, although it typically is.

Below is the operator precedence table for the Python operators discussed so far.



Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right



Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right

In the table, higher-priority operators are placed above lower-priority ones. Thus, we see that multiplication is performed before addition when no parentheses are included,

$$4 + 3 * 5 \rightarrow 4 + 15 \rightarrow \mathbf{19}$$

In our example, therefore, if the addition is to be performed first, parentheses would be needed,

$$(4 + 3) * 5 \rightarrow 7 * 5 \rightarrow \mathbf{35}$$

As another example, consider the expression below.

4 + 2 ** 5 // 10 → 4 + 32 // 10 → 4 + 3 → **7**

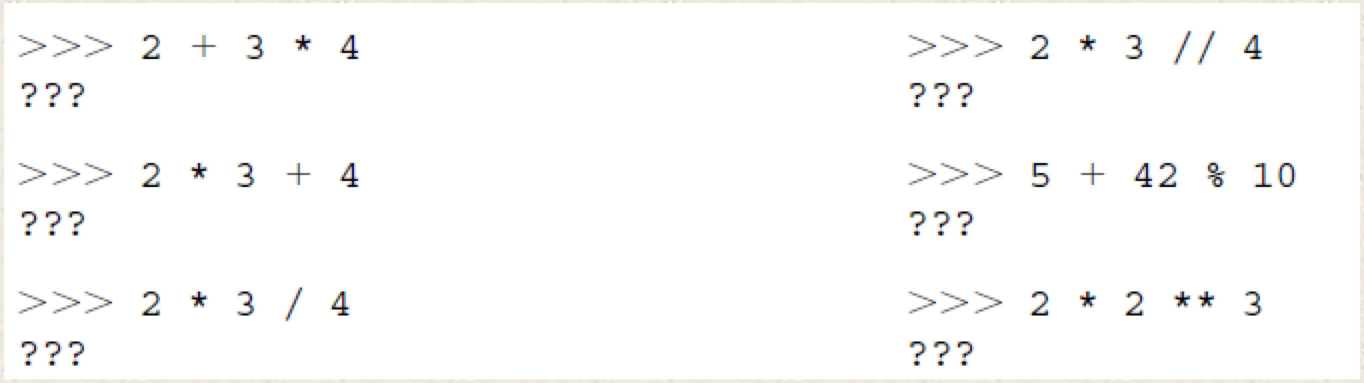
Following Python's rules of operator precedence, the exponentiation operator is applied first, then the truncating division operator, and finally the addition operator.

Operator precedence guarantees a consistent interpretation of expressions. However, it is good programming practice to use parentheses even when not needed if it adds clarity and enhances readability, without overdoing it. Thus, the previous expression would be better written as,

4 + (2 ** 5) // 10

Let’s Try It

What do each of the following arithmetic expressions evaluate to using operator precedence of Python?



```
>>> 2 + 3 * 4
???

>>> 2 * 3 + 4
???

>>> 2 * 3 / 4
???
```

```
>>> 2 * 3 // 4
???

>>> 5 + 42 % 10
???

>>> 2 * 2 ** 3
???
```

Operator Associativity

A question that you may have already had is, **"What if two operators have the same level of precedence, which one is applied first?"** For operators following the associative law (such as addition) the order of evaluation doesn't matter,

(2 + 3) + 4 → **9**   2 + (3 + 4) → **9**

In this case, we get the same results regardless of the order that the operators are applied. Division and subtraction, however, do not follow the associative law,

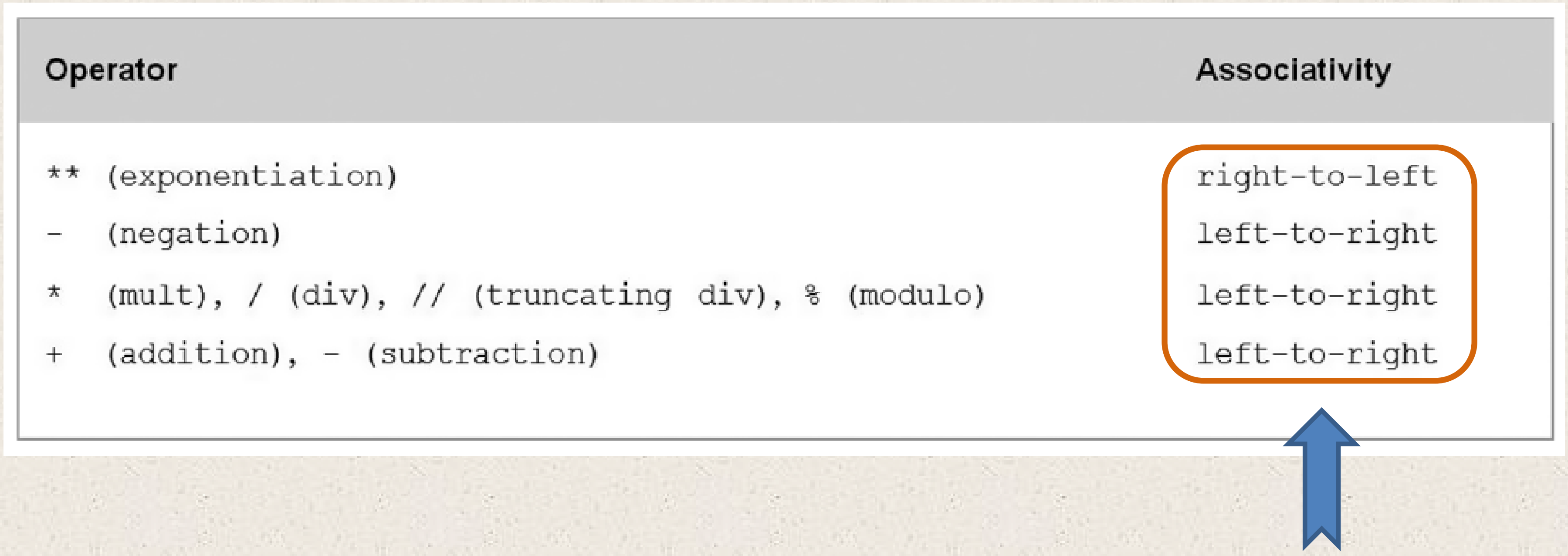
(a) (8 - 4) - 2 → 4 - 2 → **2**   8 - (4 - 2) → 8 - 2 → **6**

(b) (8 / 4) / 2 → 2 / 2 → **1**   8 / (4 / 2) → 8 / 2 → **4**

(c) 2 ** (3 ** 2) → **512**   (2 ** 3) ** 2 → **64**

Here, the order of evaluation does matter.

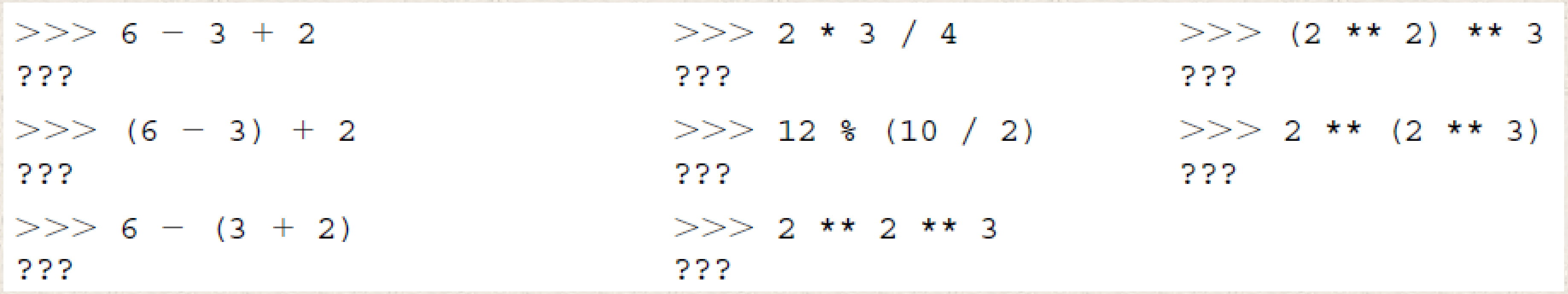
To resolve the ambiguity, each operator has a specified operator associativity that defines the order that it and other operators with the same level of precedence are applied. **All operators given below, except for exponentiation, have left-to-right associativity—exponentiation has right-to-left associativity**.



Operator	Associativity
`**` (exponentiation)	right-to-left
`-` (negation)	left-to-right
`*` (mult), `/` (div), `//` (truncating div), `%` (modulo)	left-to-right
`+` (addition), `-` (subtraction)	left-to-right

Let’s Try It

What do each of the following arithmetic expressions evaluate to?



```
>>> 6 - 3 + 2
???
>>> (6 - 3) + 2
???
>>> 6 - (3 + 2)
???
```

```
>>> 2 * 3 / 4
???
>>> 12 % (10 / 2)
???
>>> 2 ** 2 ** 3
???
```

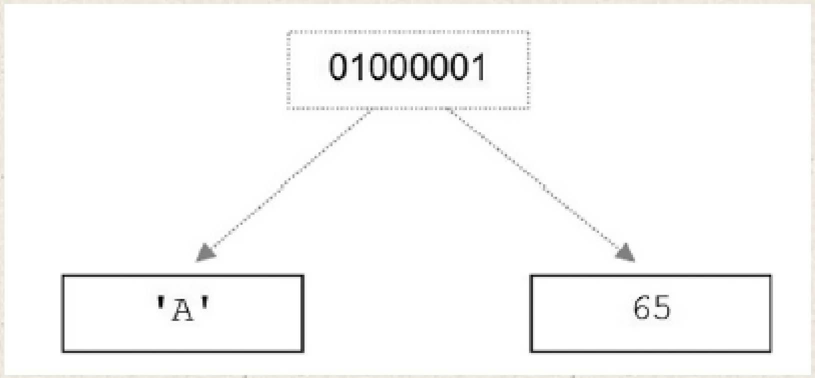
```
>>> (2 ** 2) ** 3
???
>>> 2 ** (2 ** 3)
???
```

Data Types

A **data type is a set of values, and a set of operators that may be applied to those values.** For example, the **integer data type** consists of the set of integers, and operators for addition, subtraction, multiplication, and division, among others. Integers, floats, and strings are part of a set of predefined data types in Python called the **built-in types .**

For example, it does not make sense to try to divide a string by two, **'Hello' / 2**. The programmer knows this by common sense. Python knows it because 'Hello' belongs to the string data type, which does not include the division operation.

The need for data types results from the fact that the same internal representation of data can be interpreted in various ways,



The sequence of bits in the figure can be interpreted as a character ('A') or an integer (65). If a programming language did not keep track of the intended type of each value, then the programmer would have to. This would likely lead to undetected programming errors, and would provide even more work for the programmer. We discuss this further in the following section.

Finally, there are two approaches to data typing in programming languages. In static typing, **a variable is declared as a certain type before it is used**, and can only be assigned values of that type.

In ***dynamic typing****,* ***the data type of a variable depends only*** **on the type of value that the variable is currently holding**. Thus, the same variable may be assigned values of different type during the execution of a program.

Python uses dynamic typing.

Mixed-Type Expressions

A **mixed-type expression is an expression containing operands of different type**. The CPU can only perform operations on values with the same internal representation scheme, and thus only on operands of the same type. **Operands of mixed-type expressions therefore must be converted to a common type**. **Values can be converted in one of two ways**—by implicit (automatic) conversion, called ***coercion***, or by explicit ***type conversion***.

Coercion is the *implicit* (automatic) conversion of operands to a common type. **Coercion is automatically performed on mixed-type expressions only if the operands can be safely converted**, that is, if no loss of information will result.

The conversion of integer 2 to floating-point 2.0 below is a safe conversion—the conversion of 4.5 to integer 4 is not, since the decimal digit would be lost,

2	+	4.5	→	2.0	+	4.5	→	**6.5**	**safe**	(automatic conversion of int to float)
int		**float**		**float**		**float**		**float**		

Type conversion is the *explicit* conversion of operands to a specific type. **Type conversion can be applied even if loss of information results**. Python provides built-in **type conversion functions** `int()` and `float()`, with the `int()` function truncating results

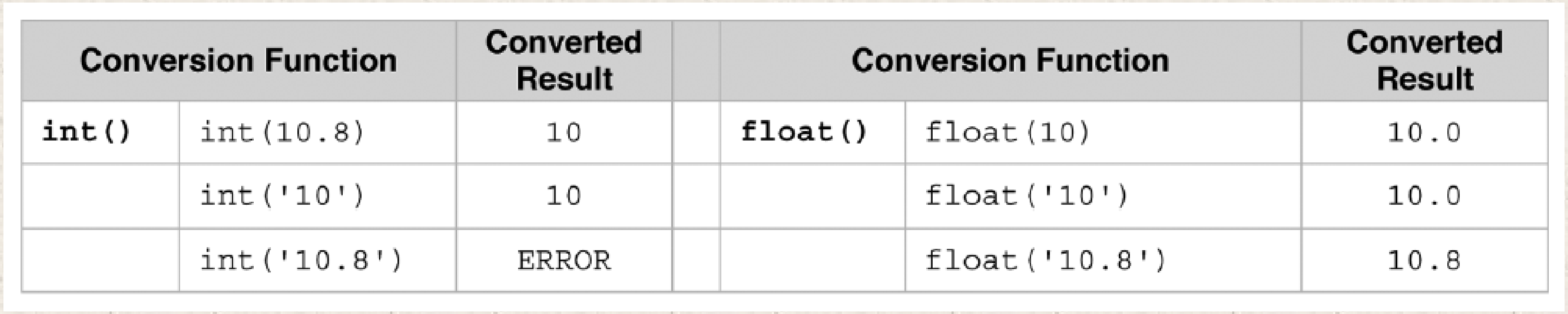
2	+	4.5	→	float(2)	+	4.5	→	2.0	+	4.5	→	**6.5**
int		float		float		float		float		float		float

No loss of information

2	+	4.5	→	2	+	int(4.5)	→	2	+	4	→	**6**
int		float		int		int		int		int		int

Loss of information

Type conversion functions int() and float()



Conversion Function		Converted Result		Conversion Function		Converted Result
`int()`	`int(10.8)`	10		**`float()`**	`float(10)`	10.0
	`int('10')`	10			`float('10')`	10.0
	`int('10.8')`	ERROR			`float('10.8')`	10.8

Note that numeric strings can also be converted to a numeric type. In fact, we have already been doing this when using **`int`** or **`float`** with the input function,

```
num_credits =  int(input('How many credits do you have? '))
```

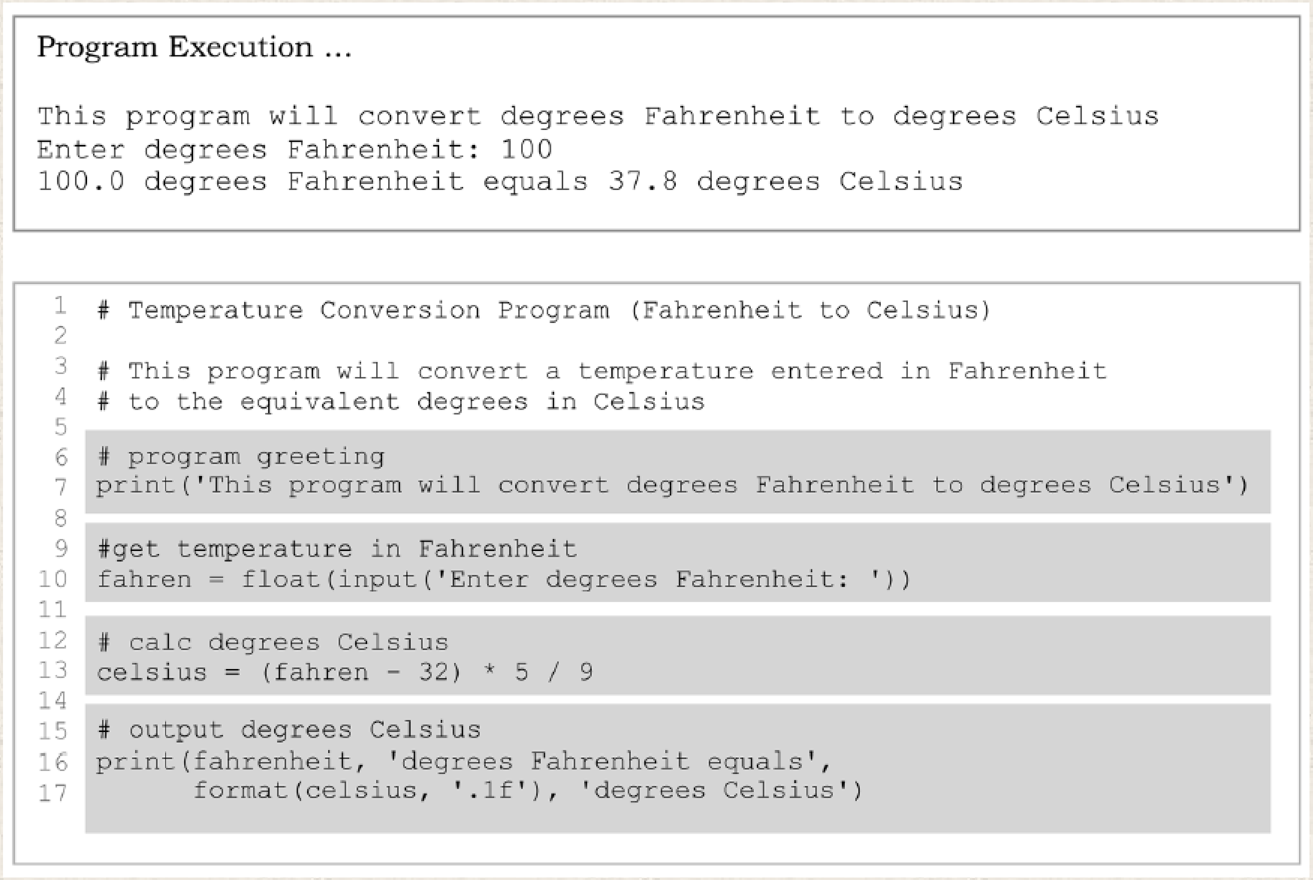
Let’s Apply It

Temperature Conversion Program

The following Python program requests from the user a temperature in degrees Fahrenheit, and displays the equivalent temperature in degrees Celsius.

This program utilizes the following programming features:

- **arithmetic expressions**
- **operator associativity**
- **built-in format function**

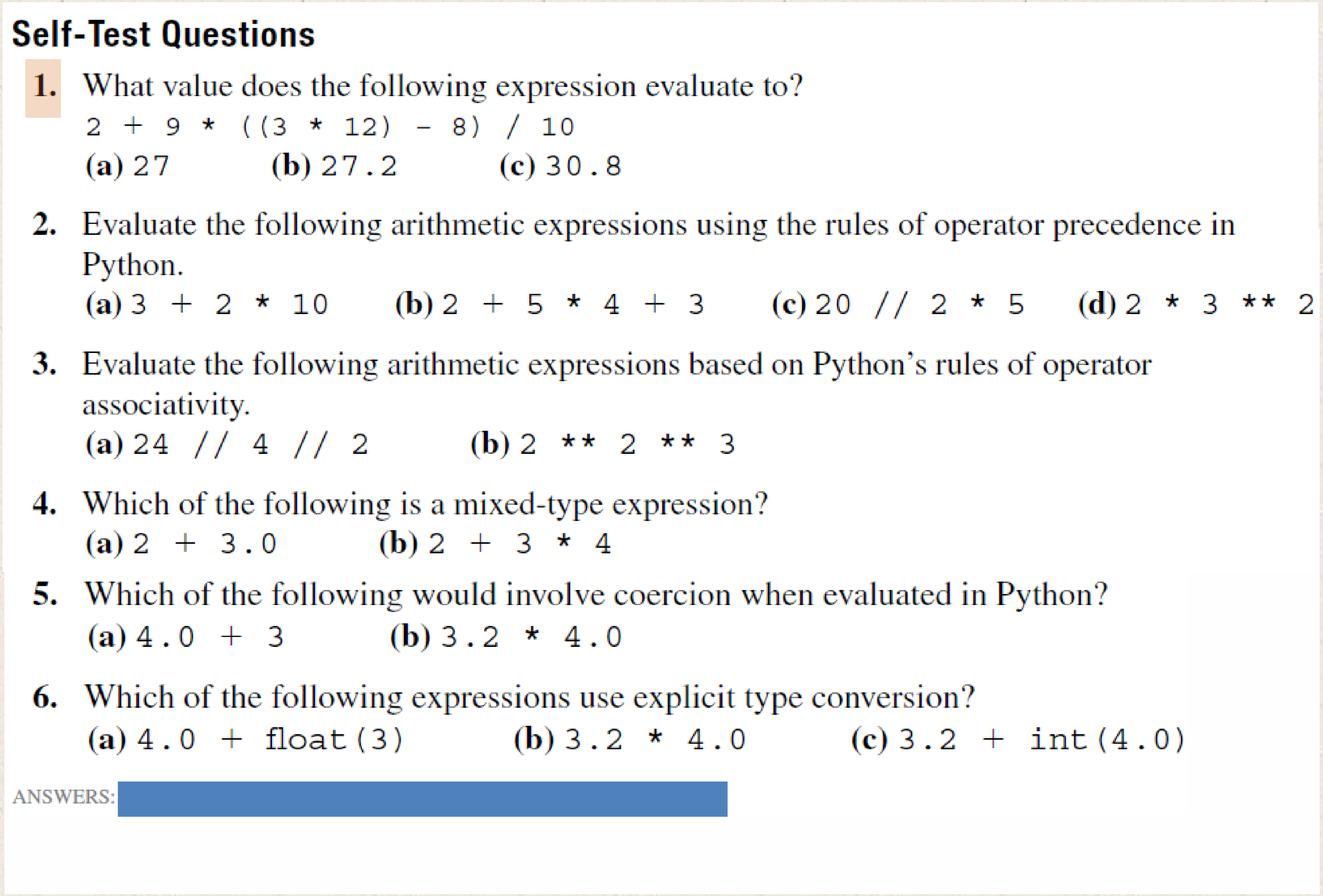


Program Execution ...

```
This program will convert degrees Fahrenheit to degrees Celsius
Enter degrees Fahrenheit: 100
100.0 degrees Fahrenheit equals 37.8 degrees Celsius
```

```
 1  # Temperature Conversion Program (Fahrenheit to Celsius)
 2
 3  # This program will convert a temperature entered in Fahrenheit
 4  # to the equivalent degrees in Celsius
 5
 6  # program greeting
 7  print('This program will convert degrees Fahrenheit to degrees Celsius')
 8
 9  #get temperature in Fahrenheit
10  fahren = float(input('Enter degrees Fahrenheit: '))
11
12  # calc degrees Celsius
13  celsius = (fahren - 32) * 5 / 9
14
15  # output degrees Celsius
16  print(fahrenheit, 'degrees Fahrenheit equals',
17        format(celsius, '.1f'), 'degrees Celsius')
```

Line 10 reads the Fahrenheit temperature entered, assigned to variable **fahren**. Either an integer or a floating-point value may be entered, since the input is converted to float type. **Line 13** performs the calculation for converting Fahrenheit to Celsius. Recall that the division and multiplication operators have the same level of precedence. Since these operators associate left-to-right, the multiplication operator is applied first. Because of the use of the "true" division operator /, the result of the expression will have floating-point accuracy. Finally, **lines 16–17 output the converted temperature** in degrees Celsius.



Self-Test Questions

1. What value does the following expression evaluate to?
   `2 + 9 * ((3 * 12) - 8) / 10`
   (a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.
   (a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.
   (a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?
   (a) `2 + 3.0` **(b)** `2 + 3 * 4`

5. Which of the following would involve coercion when evaluated in Python?
   (a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?
   (a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: [redacted]

Self-Test Questions

1. What value does the following expression evaluate to?

`2 + 9 * ((3 * 12) - 8) / 10`

(a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

(a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

(a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?

(a) `2 + 3.0` **(b)** `2 + 3 * 4`

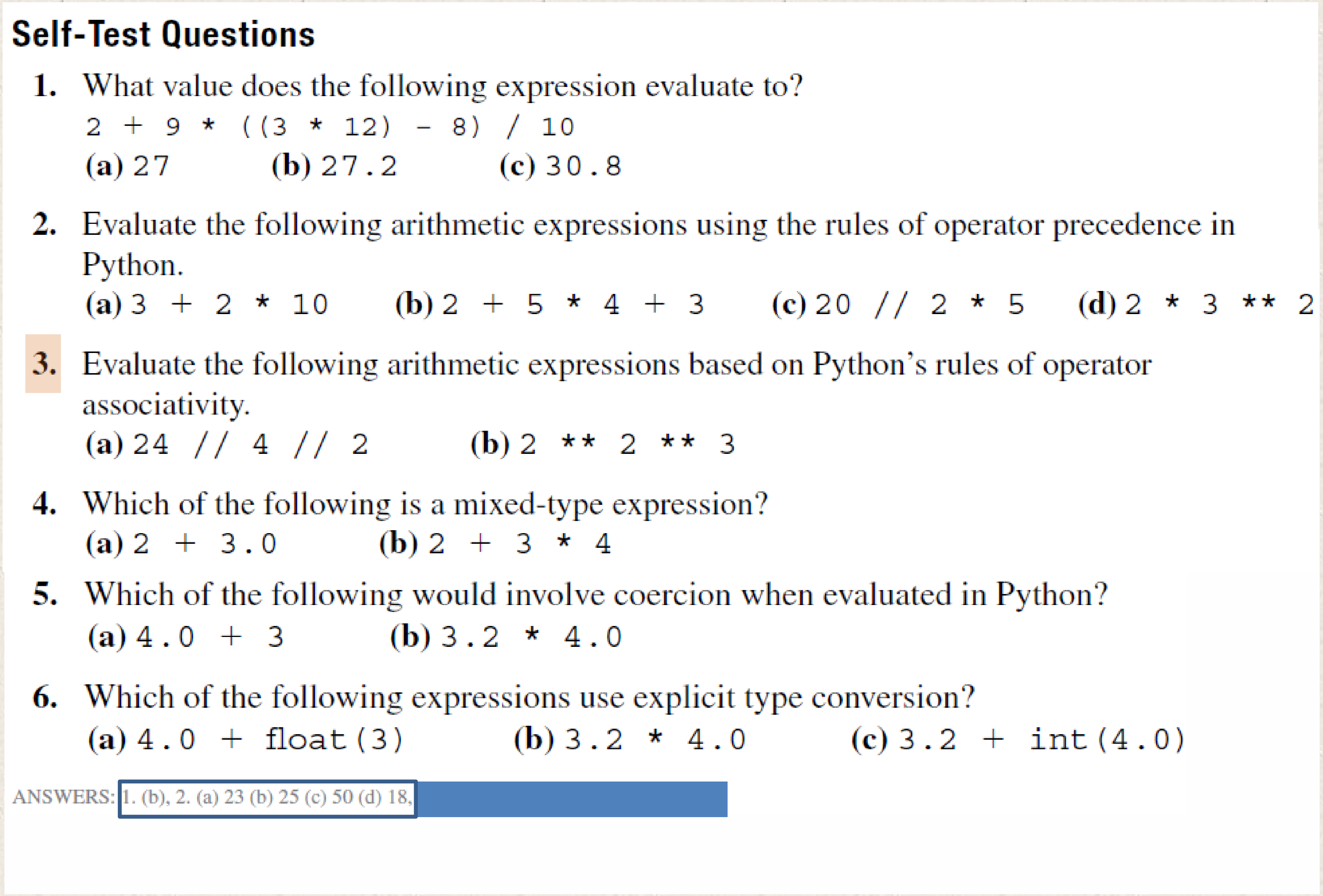
5. Which of the following would involve coercion when evaluated in Python?

(a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?

(a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), [redacted]



Self-Test Questions

1. What value does the following expression evaluate to?

```
2 + 9 * ((3 * 12) - 8) / 10
```

(a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

(a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

(a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?

(a) `2 + 3.0` **(b)** `2 + 3 * 4`

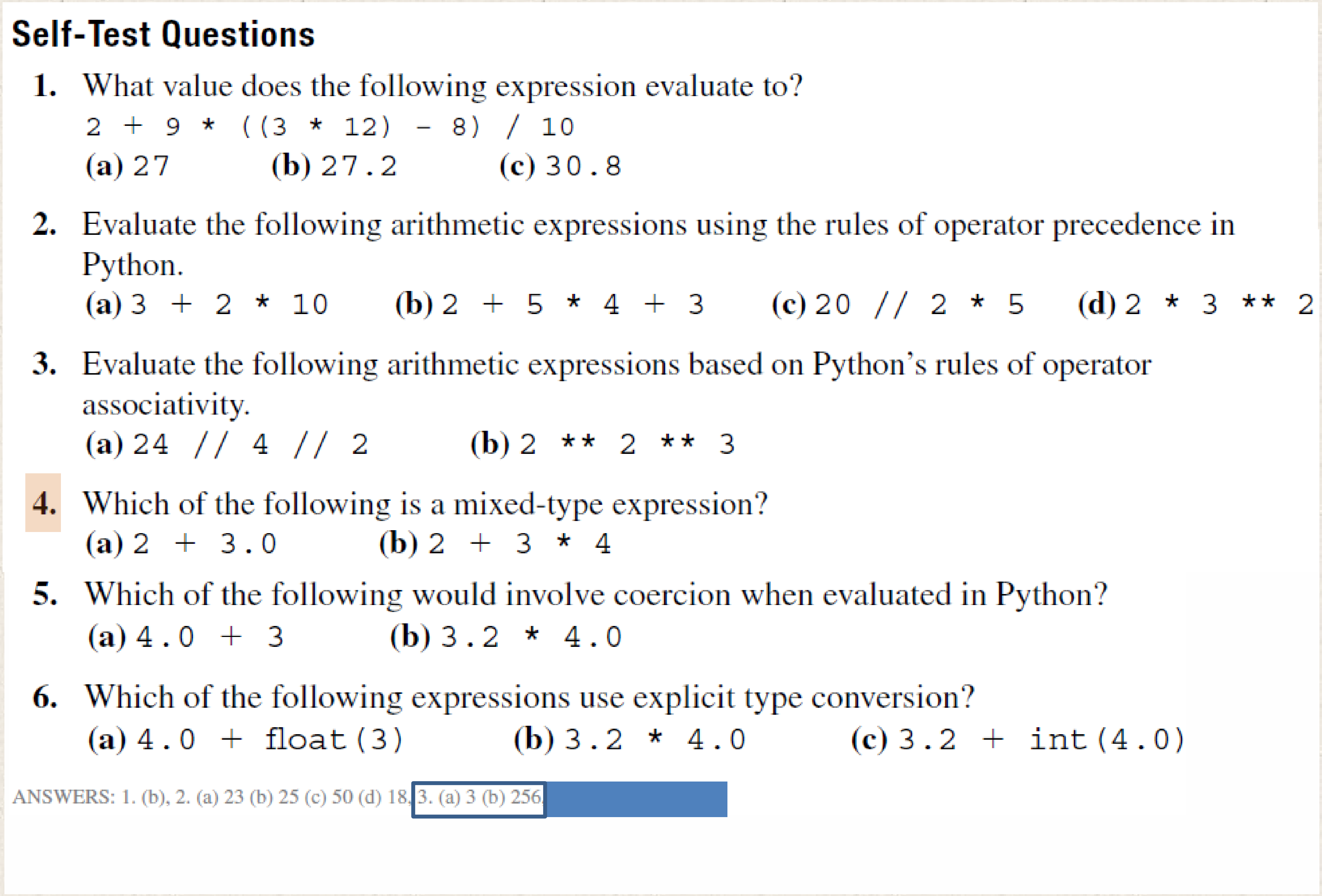
5. Which of the following would involve coercion when evaluated in Python?

(a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?

(a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, [redacted]



Self-Test Questions

1. What value does the following expression evaluate to?

   `2 + 9 * ((3 * 12) - 8) / 10`

   (a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

   (a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

   (a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?

   (a) `2 + 3.0` **(b)** `2 + 3 * 4`

5. Which of the following would involve coercion when evaluated in Python?

   (a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?

   (a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256 [redacted]

Self-Test Questions

1. What value does the following expression evaluate to?

`2 + 9 * ((3 * 12) - 8) / 10`

(a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.

(a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.

(a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?

(a) `2 + 3.0` **(b)** `2 + 3 * 4`

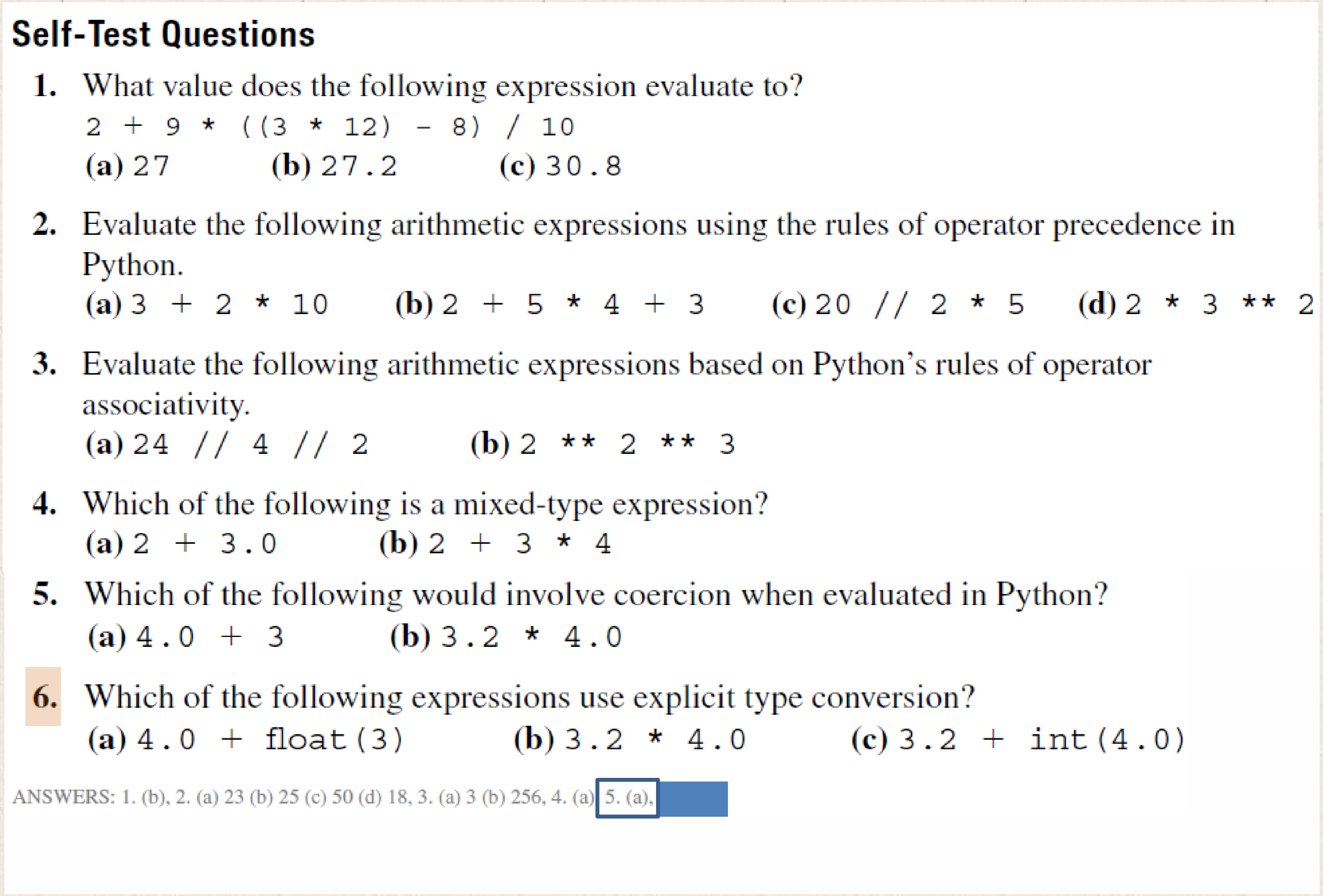
5. Which of the following would involve coercion when evaluated in Python?

(a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?

(a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256 4. (a), [redacted]



Self-Test Questions

1. What value does the following expression evaluate to?
   `2 + 9 * ((3 * 12) - 8) / 10`
   (a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.
   (a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

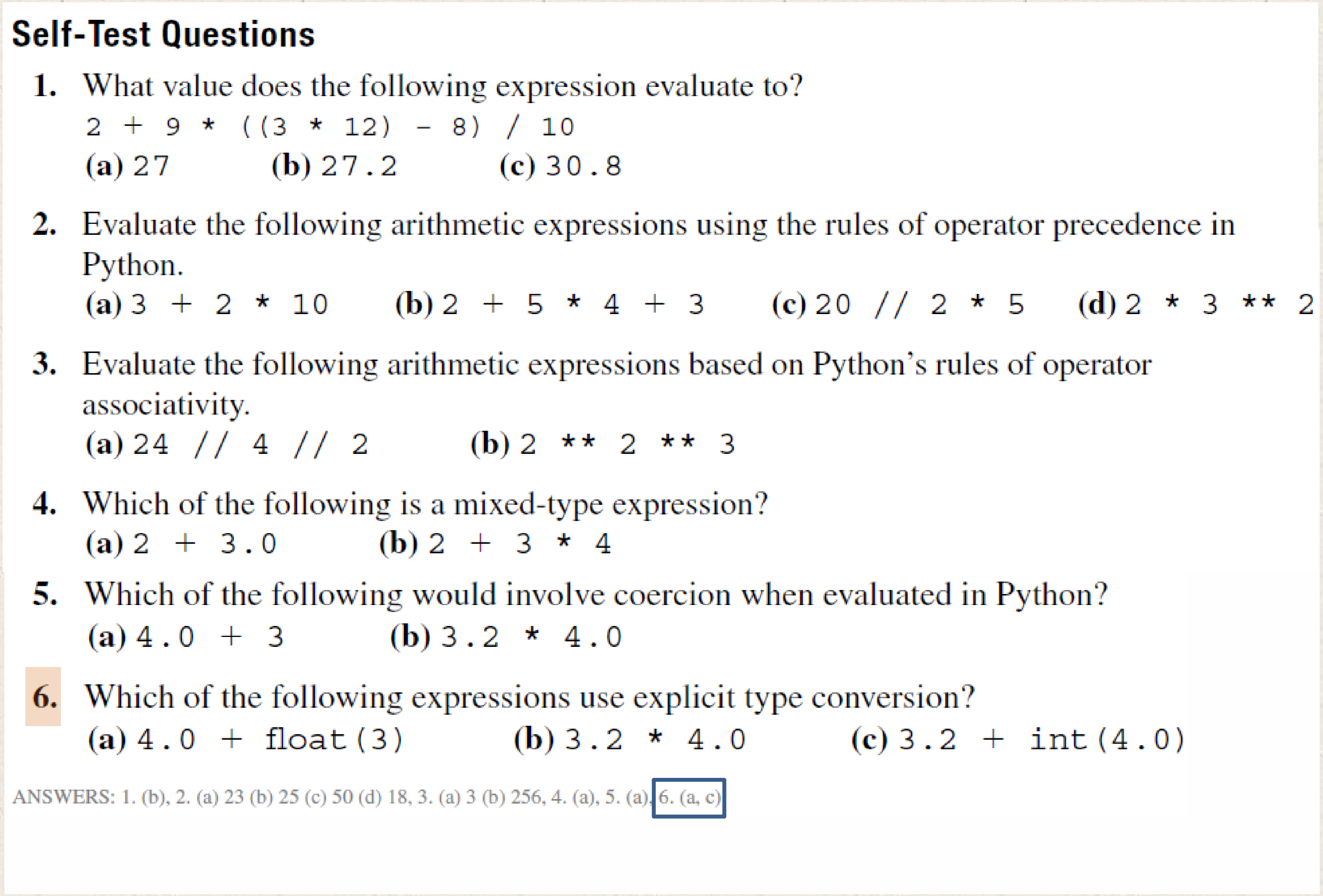
3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.
   (a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

4. Which of the following is a mixed-type expression?
   (a) `2 + 3.0` **(b)** `2 + 3 * 4`

5. Which of the following would involve coercion when evaluated in Python?
   (a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?
   (a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256, 4. (a) 5. (a), [redacted]



Self-Test Questions

1. What value does the following expression evaluate to?
   `2 + 9 * ((3 * 12) - 8) / 10`
   (a) `27` **(b)** `27.2` **(c)** `30.8`

2. Evaluate the following arithmetic expressions using the rules of operator precedence in Python.
   (a) `3 + 2 * 10` **(b)** `2 + 5 * 4 + 3` **(c)** `20 // 2 * 5` **(d)** `2 * 3 ** 2`

3. Evaluate the following arithmetic expressions based on Python's rules of operator associativity.
   (a) `24 // 4 // 2` **(b)** `2 ** 2 ** 3`

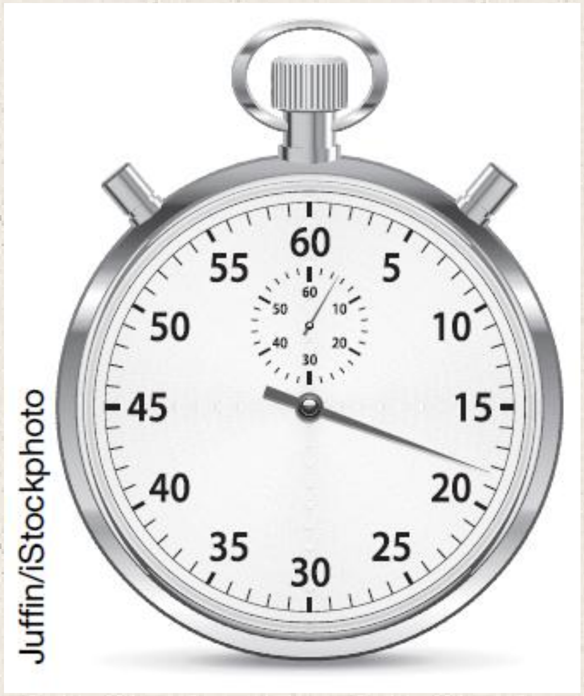
4. Which of the following is a mixed-type expression?
   (a) `2 + 3.0` **(b)** `2 + 3 * 4`

5. Which of the following would involve coercion when evaluated in Python?
   (a) `4.0 + 3` **(b)** `3.2 * 4.0`

6. Which of the following expressions use explicit type conversion?
   (a) `4.0 + float(3)` **(b)** `3.2 * 4.0` **(c)** `3.2 + int(4.0)`

ANSWERS: 1. (b), 2. (a) 23 (b) 25 (c) 50 (d) 18, 3. (a) 3 (b) 256, 4. (a), 5. (a), 6. (a, c)

Age in Seconds Program



We look at the problem of calculating an individual's age in seconds. It is not feasible to determine a given person's age to the exact second. This would require knowing, to the second, when they were born. It would also involve knowing the time zone they were born in, issues of daylight savings time, consideration of leap years, and so forth. Therefore, the problem is to determine an *approximation of age in seconds.* The program will be tested against calculations of age from online resources.

Age in Seconds
The Problem

The problem is to determine the approximate age of an individual in seconds within 99% accuracy of results from online resources. The program must work for dates of birth from January 1, 1900 to the present.

Age in Seconds
Problem Analysis

The fundamental computational issue for this problem is the development of an algorithm incorporating approximations for information that is impractical to utilize (time of birth to the second, daylight savings time, etc.), while producing a result that meets the required degree of accuracy.

Age in Seconds

Program Design

Meeting the Program Requirements

There is no requirement for the form in which the date of birth is to be entered. We will therefore design the program to input the date of birth as integer values. Also, the program will not perform input error checking, since we have not yet covered the programming concepts for this.

Data Description

The program needs to represent two dates, the user's date of birth, and the current date. Since each part of the date must be able to be operated on arithmetically, dates will be represented by three integers. For example, May 15, 1992 would be represented as follows:

```
year = 1992     month = 5      day = 15
```

Algorithmic Approach

Python Standard Library module `datetime` will be used to obtain the current date. We consider how the calculations can be approximated without greatly affecting the accuracy of the results.

We start with the issue of leap years. Since there is a leap year once every four years (with some exceptions), we calculate the average number of seconds in a year over a four-year period that includes a leap year. Since non-leap years have 365 days, and leap years have 366, we need to compute,

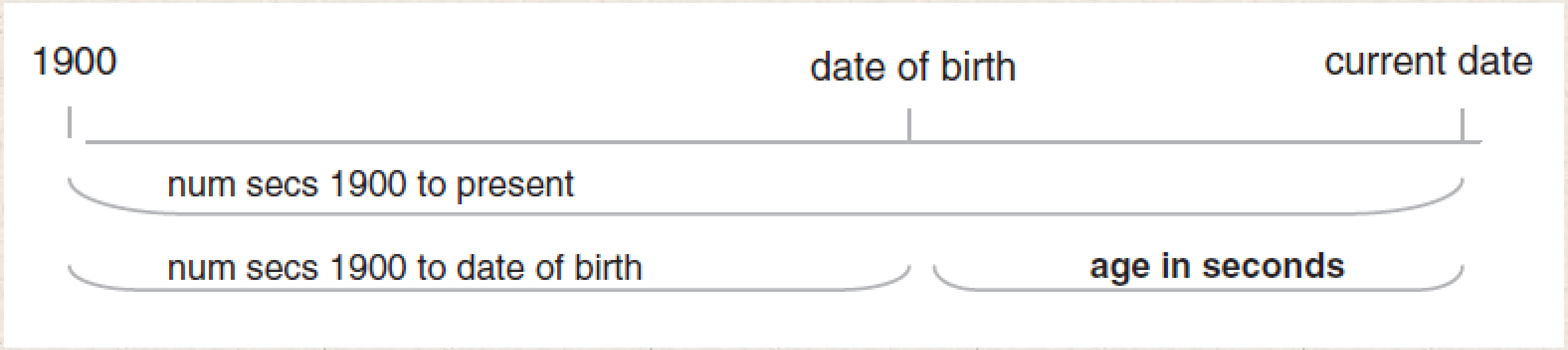
numsecs_day = (**hours per day**) * (**mins per hour**) * (**secs per minute**)

numsecs_year = (**days per year**) * **numsecs_day**

avg_numsecs_year = (4 * **numsecs_year**) + **numsecs_day**) // 4 (one extra day for leap year)

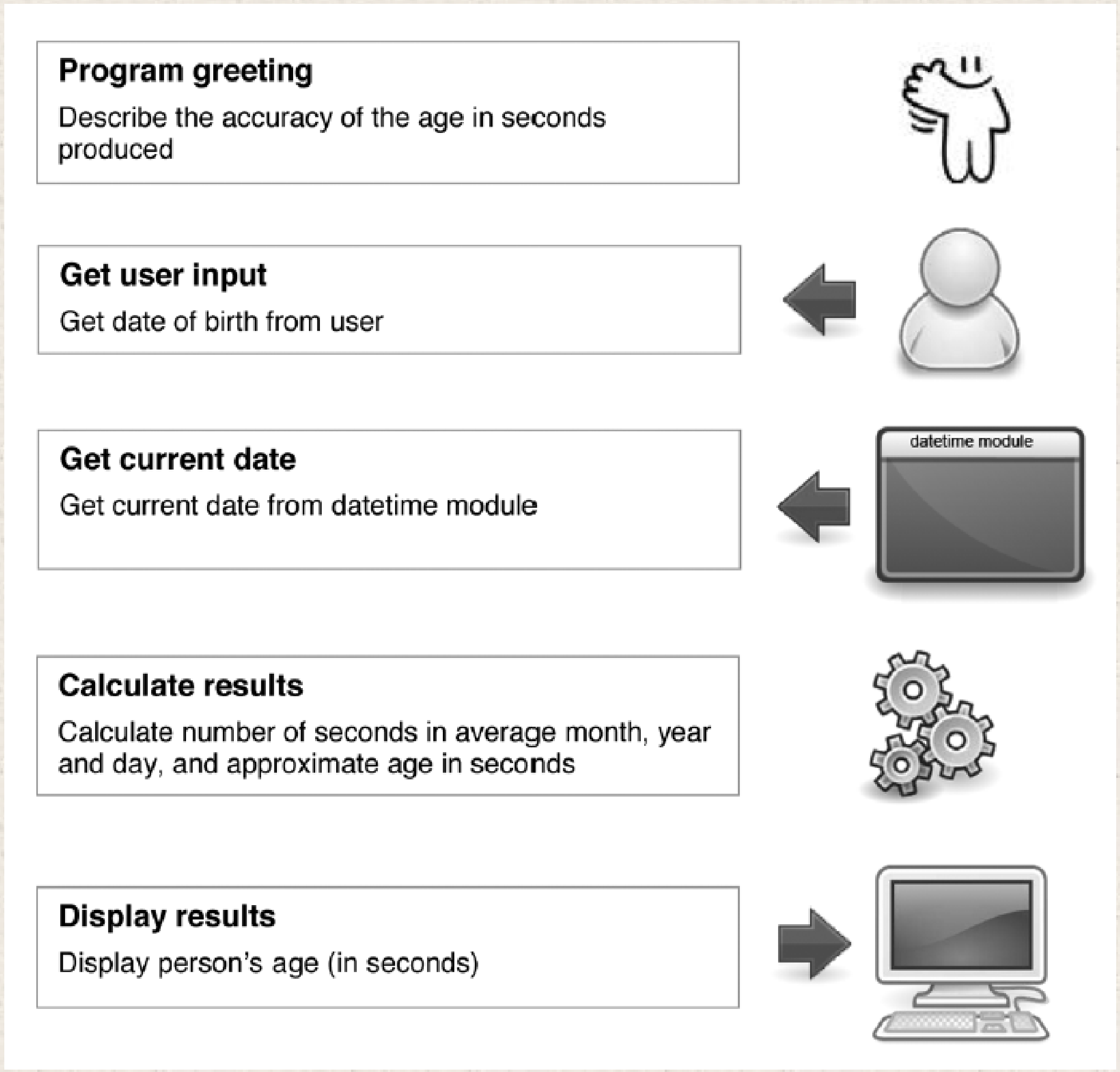
avg_numsecs_month = **avgnumsecs_year** // 12

To calculate someone's age in seconds, we use January 1, 1900 as a basis. Thus, we compute two values—the number of seconds from January 1, 1900 to the given date of birth, and the number of seconds from January 1, 1900 to the current date. Subtracting the former from the latter gives the approximate age,



Note that if we directly determined the number of seconds between the date of birth and current date, the months and days of each would need to be compared to see how many full months and years there were between the two. Using 1900 as a basis avoids these comparisons. Thus, the rest of our algorithm follows.

```
numsecs_1900_to_dob = (year_birth − 1900) * avg_numsecs_year +
                      (month_birth − 1) * avg_numsecs_month +
                      (day_birth * numsecs_day)

numsecs_1900_to_today = (current_year − 1900) * avg_numsecs_year +
                        (current_month − 1) * avg_numsecs_month +
                        (current_day * numsecs_day)

age_in_secs = num_secs_1900_to_today − numsecs_1900_to_dob
```


Program greeting
Describe the accuracy of the age in seconds produced
Get user input
Get date of birth from user
Get current date
Get current date from datetime module
datetime module
Calculate results
Calculate number of seconds in average month, year and day, and approximate age in seconds
Display results
Display person's age (in seconds)

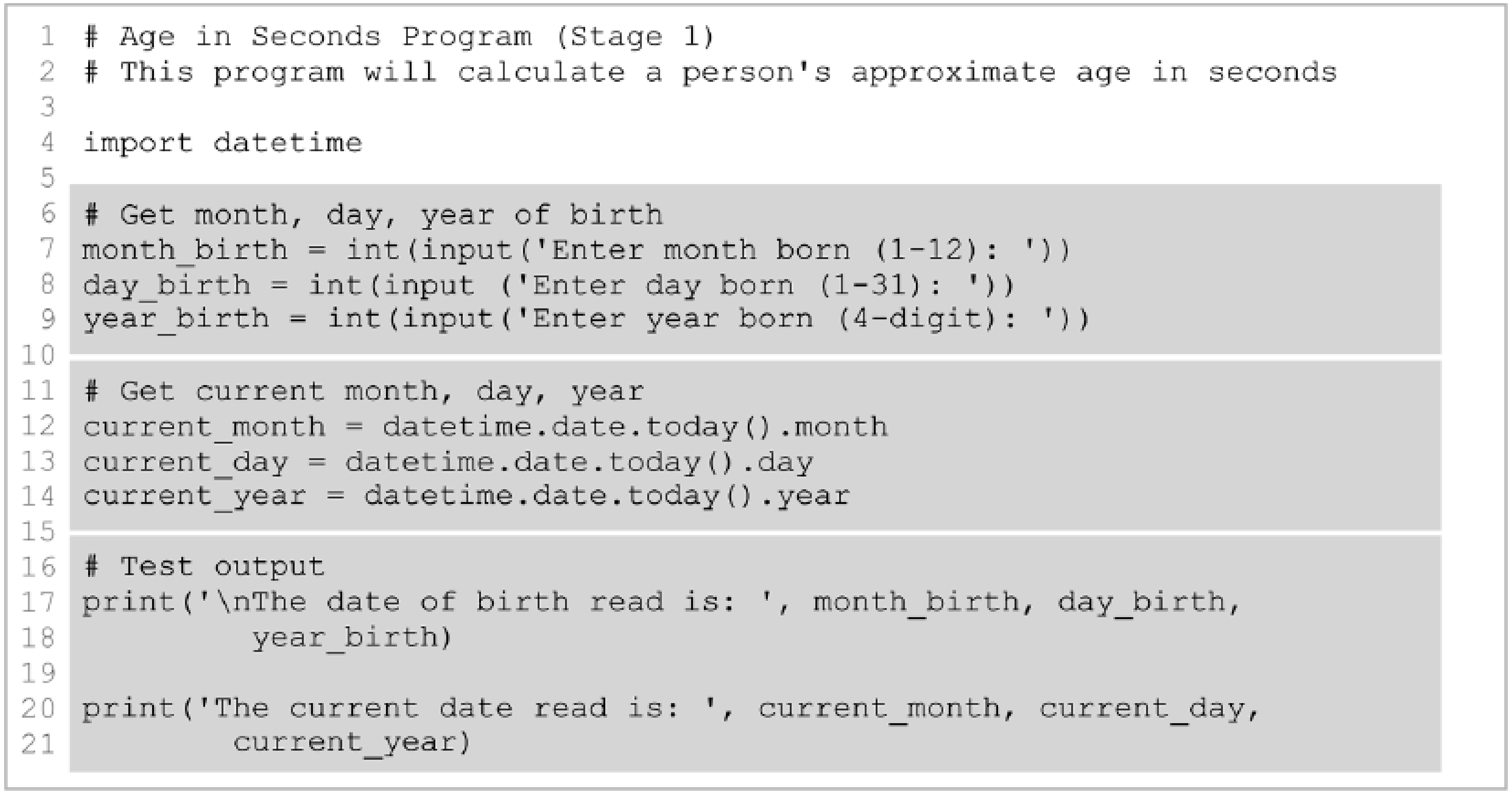
The Overall Steps of the Program

Age in Seconds
Program Implementation

Stage 1—Getting the Date of Birth and Current Date

First, we decide on the variables needed for the program. **For date of birth, we use variables month_birth**, **day_birth**, and **year_birth**.

Similarly, **for the current date we use variables current_month**, **current_day**, and **current_year**. The first stage of the program assigns each of these values.



```python
# Age in Seconds Program (Stage 1)
# This program will calculate a person's approximate age in seconds

import datetime

# Get month, day, year of birth
month_birth = int(input('Enter month born (1-12): '))
day_birth = int(input ('Enter day born (1-31): '))
year_birth = int(input('Enter year born (4-digit): '))

# Get current month, day, year
current_month = datetime.date.today().month
current_day = datetime.date.today().day
current_year = datetime.date.today().year

# Test output
print('\nThe date of birth read is: ', month_birth, day_birth,
         year_birth)

print('The current date read is: ', current_month, current_day,
        current_year)
```

Stage 1 Testing

We add test statements that display the values of the assigned variables. This is to ensure that the dates we are starting with are correct; otherwise, the results will certainly not be correct. The test run below indicates that the input is being correctly read.

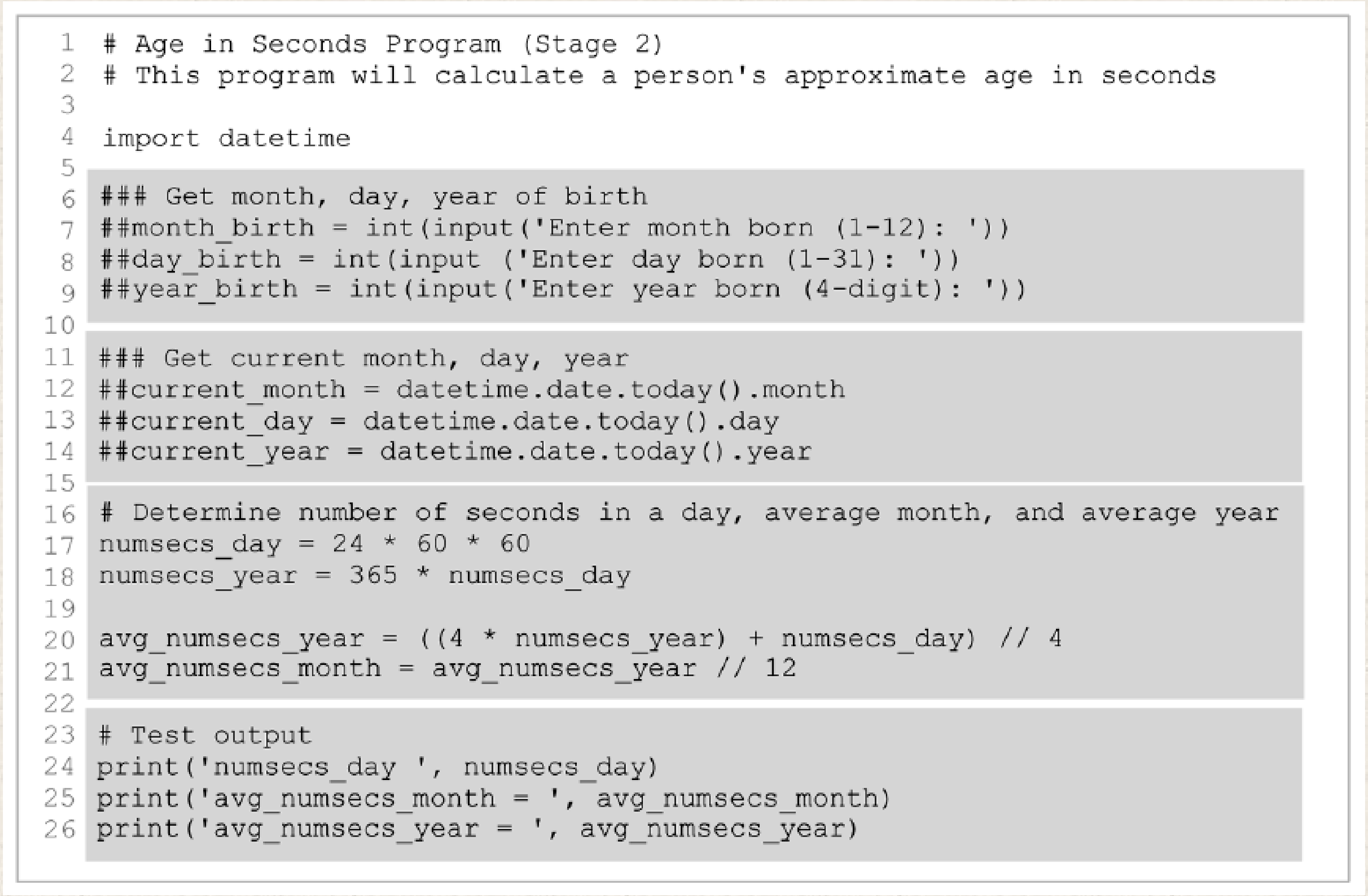
```
Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born (4-digit): 1981
The date of birth read is: 4 12 1981
The current date read is: 1 5 2010
>>>
```

Program Implementation

Stage 2—Approximating the Number of Seconds in a Year/Month/Day

Next we determine the approximate number of seconds in a given year and month, and the exact number of seconds in a day stored in variables **avg_numsecs_year**, **avg_numsecs_month**, and **numsecs_day**, respectively.

The lines of code prompting for input are commented out (lines 6–9 and 11–14). Since it is easy to comment out (and uncomment) blocks of code in IDLE, we do so; the input values are irrelevant to this part of the program testing



```
 1  # Age in Seconds Program (Stage 2)
 2  # This program will calculate a person's approximate age in seconds
 3
 4  import datetime
 5
 6  ### Get month, day, year of birth
 7  ##month_birth = int(input('Enter month born (1-12): '))
 8  ##day_birth = int(input ('Enter day born (1-31): '))
 9  ##year_birth = int(input('Enter year born (4-digit): '))
10
11  ### Get current month, day, year
12  ##current_month = datetime.date.today().month
13  ##current_day = datetime.date.today().day
14  ##current_year = datetime.date.today().year
15
16  # Determine number of seconds in a day, average month, and average year
17  numsecs_day = 24 * 60 * 60
18  numsecs_year = 365 * numsecs_day
19
20  avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
21  avg_numsecs_month = avg_numsecs_year // 12
22
23  # Test output
24  print('numsecs_day ', numsecs_day)
25  print('avg_numsecs_month = ', avg_numsecs_month)
26  print('avg_numsecs_year = ', avg_numsecs_year)
```

Stage 2 Testing

Following is the output of this test run. Checking online sources, we find that the number of seconds in a regular year is 31,536,000 and in a leap year is 31,622,400. Thus, **our approximation of 31,557,600 as the average number of seconds over four years (including a leap year) is reasonable**. The **avg_num_seconds_month** is directly calculated from variable **avg_ numsecs_ year,** and **numsecs_day** is found to be correct.

```
numsecs_day 86400
avg_numsecs_month = 2629800
avg_numsecs_year 5 31557600
>>>
```

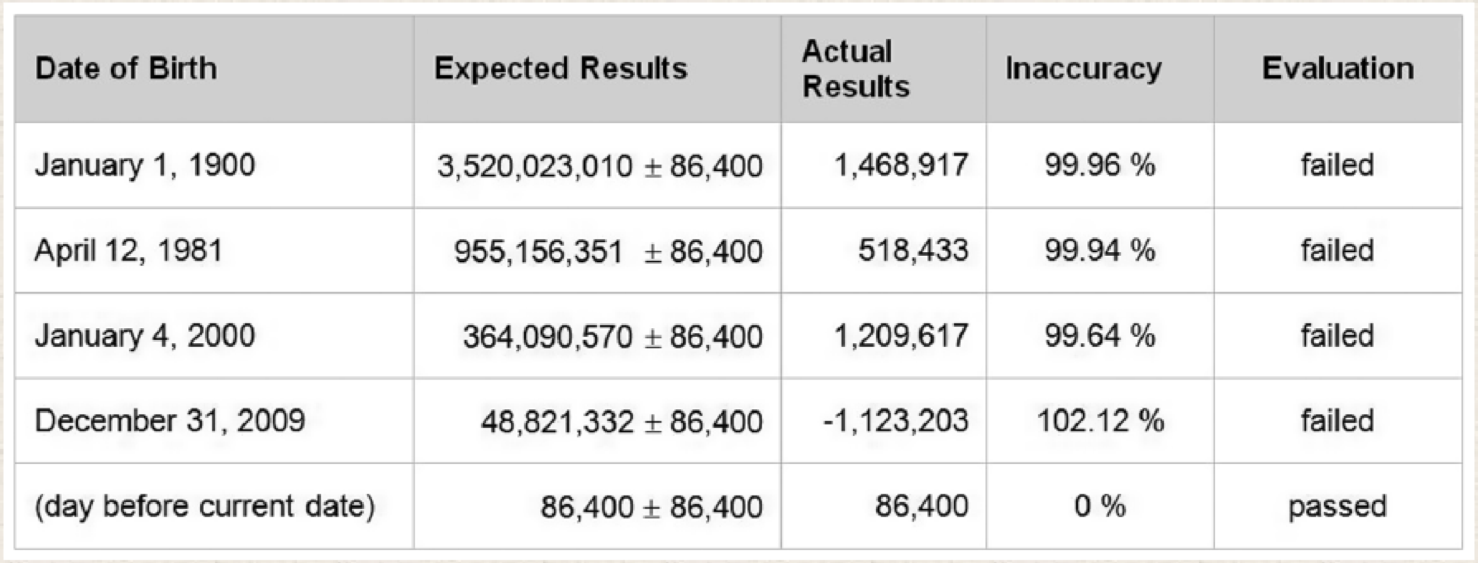
Program Implementation

Final Stage—Calculating the Number of Seconds from 1900

Finally, we complete the program by calculating the approximate number of seconds from 1900 to both the current date and the provided date of birth. The difference of these two values gives the approximate age in seconds.

```python
# Age in Seconds Program
# This program will calculate a person's approximate age in seconds

import datetime

# Program greeting
print('This program computes the approximate age in seconds of an')
print('individual based on a provided date of birth. Only dates of')
print('birth from 1900 and after can be computed\n')

# Get month, day, year of birth
month_birth = int(input('Enter month born (1-12): '))
day_birth = int(input ('Enter day born (1-31): '))
year_birth = int(input('Enter year born (4-digit): '))

# Get month, day, year of birth
current_month = datetime.date.today().month
current_day = datetime.date.today().day
current_year = datetime.date.today().year

# Determine number of seconds in a day, average month, and average year
numsecs_day = 24 * 60 * 60
numsecs_year = 365 * numsecs_day

avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
avg_numsecs_month = avg_numsecs_year // 12

# Calculate approximate age in seconds
numsecs_1900_dob = (year_birth - 1900 * avg_numsecs_year) + \
                    (month_birth - 1 * avg_numsecs_month) + \
                    (day_birth * numsecs_day)

numsecs_1900_today = (current_year - 1900 * avg_numsecs_year) + \
                    (current_month - 1 * avg_numsecs_month) + \
                    (current_day * numsecs_day)

age_in_secs = numsecs_1900_today - numsecs_1900_dob

# output results
print('\nYou are approximately', age_in_secs, 'seconds old')
```

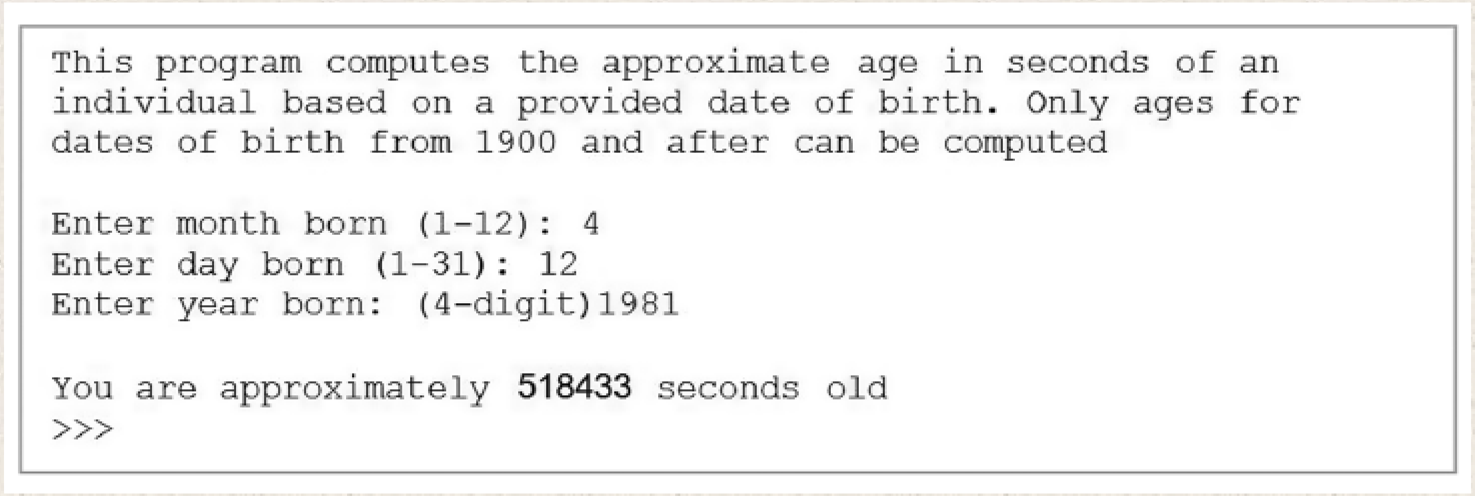
We develop a set of test cases for this program. We follow the testing strategy of including "average" as well as "extreme" or "special case" test cases in the test plan.



Date of Birth	Expected Results	Actual Results	Inaccuracy	Evaluation
January 1, 1900	3,520,023,010 ± 86,400	1,468,917	99.96 %	failed
April 12, 1981	955,156,351 ± 86,400	518,433	99.94 %	failed
January 4, 2000	364,090,570 ± 86,400	1,209,617	99.64 %	failed
December 31, 2009	48,821,332 ± 86,400	-1,123,203	102.12 %	failed
(day before current date)	86,400 ± 86,400	86,400	0 %	passed

The "correct" age in seconds for each was obtained from an online source. **January 1, 1900 was included** since it is the earliest date that the program is required to work for. **April 12, 1981 was included** as an average case in the 1900s, and **January 4, 2000** as an average case in the 2000s. **December 31, 2009** was included since it is the last day of the last month of the year, and **a birthday on the day before the current date** as special cases. Since these values are continuously changing by the second, we consider any result within one day's worth of seconds (± 84,000) to be an exact result.

Example output of results for a birthday of April 12, 1981.



```
This program computes the approximate age in seconds of an
individual based on a provided date of birth. Only ages for
dates of birth from 1900 and after can be computed

Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981

You are approximately 518433 seconds old
>>>
```

The program results are obviously incorrect, since the result is approximately equal to the average number of seconds in a month. The only correct result is for the day before the current date. The inaccuracy of each result was calculated as follows for April 12, 1981,

((abs(expected_results – actual_results) – 86,400) / expected_results) * 100 = ((917,110,352 – 518,433) – 86400) / 917,110,352) * 100 = 99.93 %

Either our algorithmic approach is flawed, or it is not correctly implemented. Since we didn't find any errors in the development of the first and second stages of the program, the problem must be in the calculation of the approximate age in lines 29–37. These lines define three variables: **numsecs_1900_dob**, **numsecs_1900_today**, and **age_in_secs**. We can inspect the values of these variables after execution of the program to see if anything irregular pops out at us.

```
This program computes the approximate age in seconds of an
individual based on a provided date of birth. Only ages for
dates of birth from 1900 and after can be computed
Enter month born (1-12): 4
Enter day born (1-31): 12
Enter year born: (4-digit)1981
You are approximately 604833 seconds old
>>>
>>> numsecs_1900_dob
-59961031015
>>> numsecs_1900_today
-59960426182
>>>
```

Clearly, this is where the problem is, since we are getting negative values for the times between 1900 and date of birth, and from 1900 to today. **We "work backwards" and consider how the expressions could give negative results.** This would be explained if, for some reason, the second operand of the subtraction were greater than the first. That would happen if the expression were evaluated, for example, as

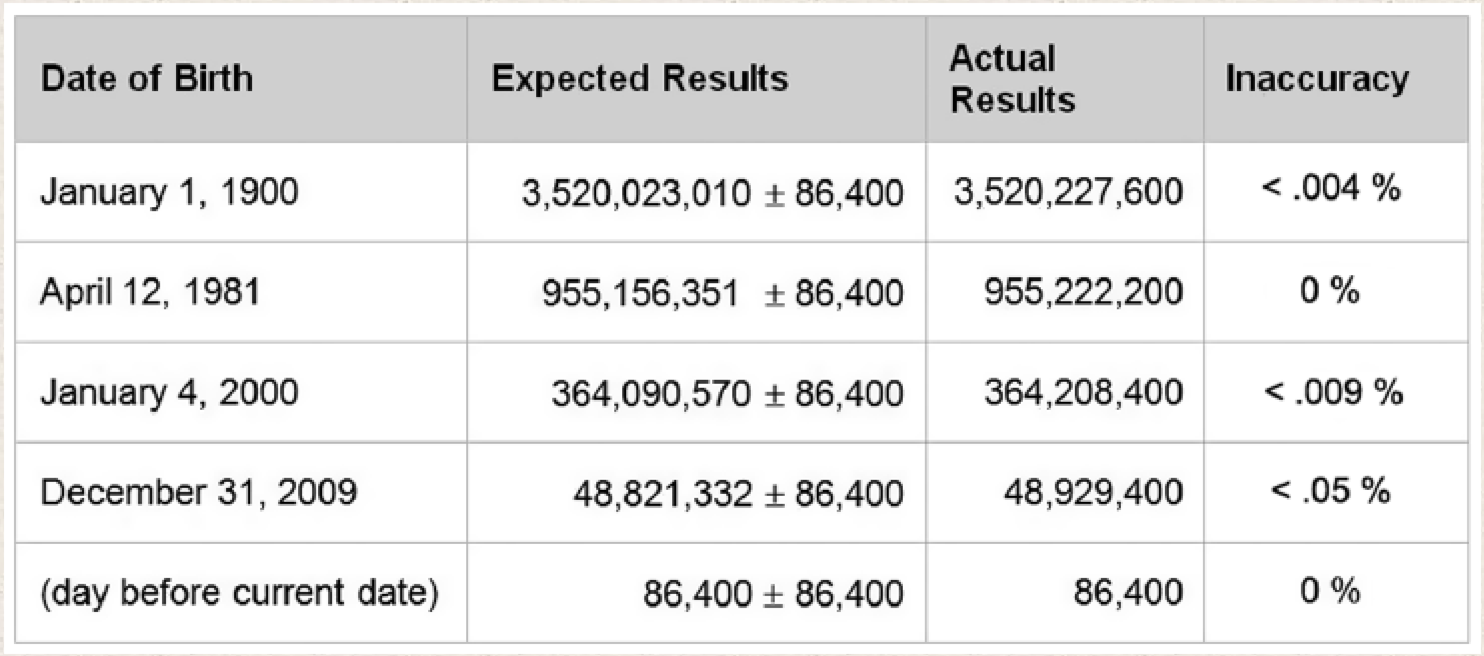
```
numsecs_1900_dob = (year_birth - (1900 * avg_numsecs_year) ) + \
                   (month_birth - (1 * avg_numsecs_month) ) + \
                   (day_birth * numsecs_day)
```

rather than the following intended means of evaluation,

```
numsecs_1900_dob = ( (year_birth - 1900) * avg_numsecs_year) + \
                   ( (month_birth - 1) * avg_numsecs_month) + \
                   (day_birth * numsecs_day)
```

Now we realize! Because we did not use parentheses to explicitly indicate the proper order of operators, **by the rules of operator precedence Python evaluated the expression as the first way above, not the second as it should be**. This would also explain why the program gave the correct result for a date of birth one day before the current date.

Once we make the corrections and re-run the test plan, we get the following results,



Date of Birth	Expected Results	Actual Results	Inaccuracy
January 1, 1900	3,520,023,010 ± 86,400	3,520,227,600	< .004 %
April 12, 1981	955,156,351 ± 86,400	955,222,200	0 %
January 4, 2000	364,090,570 ± 86,400	364,208,400	< .009 %
December 31, 2009	48,821,332 ± 86,400	48,929,400	< .05 %
(day before current date)	86,400 ± 86,400	86,400	0 %

These results demonstrate that our approximation of the number of seconds in a year was sufficient to get well within the 99% degree of accuracy required for this program. We would expect more recent dates of birth to give less accurate results given that there is less time that is approximated. Still, for test case December 31, 2009 the inaccuracy is less than .05 percent. Therefore, we were able to develop a program that gave very accurate results without involving all the program logic that would be needed to consider all the details required to give an exact result.