Part I
Basic Concepts

Computer programs always operate on data. That data can take many different forms. Examples are numerical data operated on by a software system to compute taxes, textual data operated on by a word processor, and graphical data operated on by an application for correcting images. In this part, we learn how computer programs can operate on integer numbers, on broken numbers referred to as floating-point numbers, and on sequences of symbols referred to as strings of characters. Programming languages typically offer built-in support for these basic types of data.

In this part, we learn how to write simple programs that consist of basic instructions to read input data, to store data in memory, and to print computed results. We also learn to use conditional statements and iterative statements to steer the flow of control, i.e. the actual sequence of instructions to be executed by the machine. Conditional statements serve to choose between several alternative sequences of instructions based on the current data. Iterative statements make it possible to execute the same sequence of instructions over and over again on slightly changing data.

Computers are not able to execute programs written in Python, C++, Java, C# or any other programming language. Some programming languages such as C++ come with a so-called *compiler* that translates programs into machine code. Other programming languages such as Python, Java and C# offer a so-called interpreter to execute their programs. An *interpreter* is another computer program specifically developed to execute programs written in a certain programming language.

Chapter 1
Integer Arithmetic

Learning outcomes

Write documentation in a structured way.

Read data input from end-users.

Store intermediate results of computations in variables.

Compute with integer numbers.

Influence the flow of control by means of selection statements.

Communicate final results of computations to end-users.

1.1 Example Program

In this chapter, we discuss Python concepts for computing with integer numbers. Our illustration is a program that reads a year and determines whether or not that year is a *leap year*. A year is a leap year if it is a multiple of 4 and not a multiple of 100, or if it is a multiple of 400. For example, 2012 is a leap year, whereas 2011 is not. The year 2000 is also a leap year, because it is a multiple of 400. The year 2100, on the other hand, is not a leap year. It is a multiple of 100, and not a multiple of 400. Example 1 shows the entire Python program for this problem. We explain it in detail in the course of this chapter:

```python
# Check whether a given year is a leap year.
#   A year is a leap year if it is a multiple of 4 and
#   not a multiple of 100, or it is a mutiple of 400.
#
# Author: Eric Steegmans
# Date: September 2016

year = int(input("Enter the year to examine: "))

is_4_multiple   = (year % 4 == 0)
is_100_multiple = (year % 100 == 0)
is_400_multiple = (year % 400 == 0)

if is_4_multiple and \
     ( (not is_100_multiple) or is_400_multiple):
  print("The year", year, "is a leap year!")
else:
  print("The year", year, "is not a leap year!")
```

Example 1: Program to determine whether a year is a leap year.

1.2 Computer Program

A *computer program* is a series of instructions that can be executed by a computer. A program is written to perform a specific task. A simple example is a computer program to calculate the greatest common divisor of two integer numbers. A more complex example is a program to compute the shortest route from one location to another. When things get even bigger, we talk about software systems that consist of a collection of computer programs.

A computer program typically involves input and output. The keyboard is the default instrument to supply data to a computer program. We say that the *standard input stream* for a computer program is connected to the keyboard. In the simple example of the program to compute the greatest common divisor, the end-user enters two integer numbers using the keyboard. The computer program reads these numbers, and computes their greatest common divisor. The keyboard is just one device used to provide input data. Computer programs can get data from many other devices, such as a network connection, a hard disk, a computer mouse or a joystick.

The computer screen is the default instrument to present results that have been computed by a program. We say that the *standard output stream* for a computer program is connected to the computer screen. In the example of the greatest common divisor, the program displays the greatest common divisor of the given integer numbers on the computer screen. Again, there are many other devices that computer programs can use to communicate their results. Examples include network connections, hard disks and printers.

Computer programs are written in a programming language. In this book, we use Python as the language in which we write all our programs. Examples of other popular programming languages are Java, C#, C++ and Scala. Computers themselves have a very primitive set of machine instructions that they can execute. In the early days of computing, programs were written directly in machine code. However, high-level programming languages offering higher-level concepts were soon introduced to simplify the process of writing programs. High-level programming languages also made it possible to write more complex computer programs dealing with more complex problems. In his famous book *The Mythical Man-Month*, Fred Books states that a software engineer is able to write 10 lines of (correct) code per day, regardless of the level of the programming language.

Programs written in high-level programming languages must either be translated into machine code, or there must be some other instrument that is able to execute them. Translating computer programs written in a high-level programming language into machine code is known as *compilation*. A compiler not only translates the program in question. It also checks whether the program has been written according to the rules of the programming language. Nowadays, most computer programs are interpreted. An *interpreter* is an application (a computer program) specifically developed to execute computer programs written in some programming language. Prior to the actual execution of the program, the interpreter may check its correctness. Python, Java and C# all use an interpreter to execute their programs. C++ still sticks to a compiler.

1.3 Documentation

Readability is a basic requirement for programs written in any programming language. Programs we write must be easy to understand for others. Proper naming of all the ingredients of a program makes programs a lot easier to read. Functions and variables are such ingredients, and we discuss them in the initial chapters of this book. Suppose we must write a program that needs to store the amount of money held on a bank account. If we name the variable that we introduce for that purpose `balance`, the reader can guess the intention of that variable from the name. If we were to use names such as `b` or `bl`, it would be a lot harder to guess their role in the program. Another example: suppose we want to devise a function to compute the total cost of some financial transaction. Naming that function `total_transaction_cost` immediately reveals its purpose. Shorter names such as `ttc` or `tr_c` would be mysterious for the reader.

Proper naming of variables, functions and other ingredients is not enough to make programs easy to read. At specific points, we must add comments concerning the program. These might be a description of the overall goal of some ingredient. They might also clarify the way the program computes certain things. Programming languages offer programmers *comments* to add additional information wherever needed. Those comments are written in some natural language such as English or Dutch. In Python, comments start with the symbol # and proceed up to the end of the line. The symbol # is called the number sign or the hash. Comments are just there to give information to the readers of programs. They have no impact at all on the ultimate results that programs produce. In executing programs, the Python interpreter simply skips all the comments it encounters.

A program typically starts with a brief description of its overall purpose. The brief description at the start of Example 1 reveals that the program is intended to check whether a given year is a leap year. Subsequent lines may give some more details concerning the program. In that example, the next two lines describe the conditions for a given year to be a leap year. The general information concerning a program typically finishes with some administrative stuff. In the example, the comment in front of the leap year program lists the author of the program and the date it was last modified. We will not include that type of information in examples in the rest of this book.

1.4 Input

Python provides a large collection of predefined functions that we can use in our programs. Some of these functions are part of the language itself. These so-called *built-in functions* provide functionality that we need in lots of programs. Examples are functions to read data, to print data and to compute the absolute value of some number. Other functions are offered as part of modules that we can import into our programs whenever we need them. The *Python standard library* provides a large collection of library functions organized in modules. Examples are the module `math`, which includes mathematical functions such as `factorial(n)`, `sin(x)` and `log(x)`, and the module `datetime`, which comprises date and time functions such as `today()`, `time()` and `date(year,month,day)`. In Chapter 5, we discuss how we can define our own functions. They cover needs that are specific to the program in question.

Python provides the built-in function `input(prompt)` to read data from the standard input stream. This function takes a string as its argument. We discuss the semantics of strings in much more detail in Chapter 4. For the time being, it is sufficient to know that a string in Python is a sequence of characters enclosed in single quotes (`'`) or in double quotes (`"`). For example, `'Leuven'` is a string. Other examples of strings are `"Celestijnenlaan 200"`, `'John.Carter@gmail.com'` and `'#k&!]'`.

The function `input(prompt)` displays the prompt. This tells users what kind of data they need to supply. The function then reads the characters that the user has entered in response to the prompt, and returns them in the form of a string. Experiment 1 below shows some experiments with the built-in function `input`. The first experiment shows that the function returns a string that contains all the characters that have been entered by the user. The second experiment illustrates that the function itself does not try to transform the sequence of characters into something else. In that experiment, the function does not return the integer number 23. It returns a string consisting of the characters `'2'` and `'3'`. The third experiment shows that users should not enclose their response in quotes. In the experiment, the user enters his name enclosed in double quotes. Those quotes are part of the string the built-in function `input` returns as its result:

```
>>> input("Enter something: ")
Enter something: hello 42 out %there%
'hello 42 out %there%'

>>> input("Enter an integer: ")
Enter an integer: 23
'23'

>>> input("Enter your name: ")
Enter your name: "Steegmans"
'"Steegmans"'
```

Experiment 1: The built-in function `input`.

Example 1 shows how to read the year under examination. The program prompts the user to enter the year. It then waits for the user to enter an integer number. The string returned by the built-in function `input` is then transformed into an integer number by means of the built-in function `int`. That number is then assigned to the variable `year`. The next section discusses the semantics of assignment statements in Python. The semantics of the built-in function `int` are discussed in more detail in section 1.7. The program assumes that the user will enter values of the correct type. It will probably crash at some point if the user supplies unsuitable inputs. For example, if the user supplies `two thousand seventeen` as the year for testing, the program will crash giving a rather curious message (try it out!).

1.5 Assignment Statement

The programs that we work through in the initial chapters of this book are very simple. They easily fit onto a single page. Later chapters use more complex programs to illustrate more advanced concepts. In practice, software systems are much more complex and may require hundreds of man-years of work. A man-year is the amount of work one person can do in a single year composed of a standard number of working days. Obviously, complex software projects are developed by large teams that may consist of dozens or even hundreds of software engineers.

Regardless of their complexity, programs must be able to store intermediate results. Programming languages offer variables to which values can be assigned during the execution of programs. It is handy to think of *variables* as named boxes in which we can store a single piece of information, such as an integer number, a floating-point number, or a string. We can inspect the contents of a variable via its name, and we can change its contents by storing other information in the box.

1.5.1 Basics of assignment statements

In its most simple form, an *assignment statement* in Python involves the name of a variable on the left and an expression on the right. The sides are separated from one another by means of the assignment operator (=). Basically, the value of the expression on the right is evaluated and then assigned to the variable on the left. Using the box metaphor, the value resulting from the evaluation of the expression is stored in the box associated with the variable in question. If the variable does not yet exist, it comes into existence as a result of the execution of the assignment statement. In other words, if the variable does not already exist, a new box named after the variable is created. If, on the other hand, a value has already been assigned to the variable, the new value replaces the old value. In other words, the old content is removed from the box associated with the variable, and the new value is stored in that box.

For the moment, we are only going to use simple expressions involving a single variable, constant or function. In section 1.6, however, we learn how to write more complex expressions involving operators that are applied to operands. Experiment 2 below shows examples of simple assignment statements. In the first assignment, the value `42` is assigned to a variable named `magic`. Assuming no variable named `magic` has been assigned before, the variable comes into play and its contents can be inspected from this point on. In Experiment 2 below, we ask for the value of the variable `magic` by simply typing its name after the assignment statement has been executed. Python evaluates this simple expression and returns its value, `42` in this case. The experiment continues with similar assignments involving floating-point numbers and strings:

```
>>> magic = 42
>>> magic
42

>>> price = 79.95
>>> price
79.95

>>> message = "Hello from Flanders!"
>>> message
'Hello from Flanders!'
```

Experiment 2: Assignment statements.

1.5.2 Classification of errors

The programs that we write will not always be immediately correct. We distinguish between syntax errors, runtime errors and semantic errors in programs. Python defines a particular syntax for each of its constructs. Whenever we ask to execute a program, Python checks whether the program is syntacticly correct. If it is not, Python signals the *syntax error* and refuses to execute the program. In Experiment 3 below, the first line of code illustrates the notion of a syntax error. We have typed a colon instead of an equal sign to separate the variable on the left of the assignment statement from the expression on the right. Python signals the syntax error by means of an arrow pointing to the unexpected character.

Programs that are syntacticly correct may fail during their execution. Such errors are typically called *runtime errors*. Experiment 3 below also shows a runtime error. Here, we try to assign the value of a variable `b` to another variable `a`. However, the variable `b` has not been assigned yet. The Python interpreter starts to execute the program, but stops as soon as it encounters the faulty assignment statement. Note that Python classifies errors in various categories, such as `SyntaxError`, `ZeroDivisionError` and `NameError`.

The final types of errors are *semantic errors*. They are by far the hardest to find. Here the program in question is syntacticly correct. Moreover, the execution of the program does not terminate with a runtime error. However, the results produced by the program in question are not as expected. For example, suppose we have written a program to compute the greatest common divisor of two natural numbers. If we execute the program to compute the greatest common divisor of 12 and 81, an incorrect program might return 6, which is obviously incorrect. Another example: we write a program to calculate the square root of some number, but when we execute it, it never stops. Such programs are said to be in an infinite loop:

```
>>> a : 42
      ^
SyntaxError: invalid syntax

>> a = b
NameError: name 'b' is not defined
```

Experiment 3: Syntax errors and semantic errors.

1.5.3 Semantics of assignment statements

The semantics of assignment in Python are a bit more complicated than simple boxes in which we store values. To get a deeper understanding of the semantics of assignment in Python, we need to know that Python programs have access to the computer's central memory for storing pieces of information. The central memory is a large collection of memory cells. Each cell has a unique address. Python refers to consecutive memory cells storing some piece of information as *objects*. Objects can store simple values such as integer numbers, floating-point numbers or strings. However, objects can also store more complex things such as lists, tuples, sets and dictionaries, which we discuss in the following chapters. The notion of an object stems from the paradigm of object-oriented programming that we discuss in Chapter 18. That paradigm introduces classes from which objects can be created. A typical example is a class of bank accounts. Objects belonging to that class may store the balance, the credit limit and the holder. Whatever its actual type, the evaluation of an expression in Python always results in an object in which the value of the expression in question is stored.

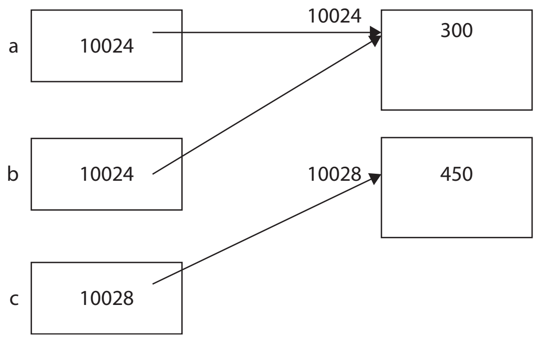

Figure 1: Semantics of simple assignments.

In Python, a single memory cell is assigned to each of the variables used in the program under execution. That cell does not directly store the value assigned to that variable. Instead, it stores the address of the object in which the actual value is stored. Figure 1 illustrates the bindings of variables for the code snippet shown below. In the first assignment statement, the value -300 is assigned to variable a. If this statement is executed, the Python interpreter allocates a memory cell for variable a. In the example, we assume that the evaluation of the right-hand side of the assignment yields a new object at address 10024 in which the value -300 is stored. The assignment in question then stores the address 10024 in the cell assigned to variable a.

We will not consistently provide addresses of memory cells in graphical illustrations in this text. Instead, we use arrows that point to the memory cells storing the value. In Figure 1, both addresses and arrows are shown to emphasize their equivalence. We say that the variable in question references (or points to) the memory cells (the object) storing its value. In Python terminology, variable a references an object that stores the integer value -300.

The next assignment in the code snippet below assigns the contents of variable a to another variable, b. The execution of the statement allocates another memory cell to variable b. This time, no new object needs to be created to store the result of the expression on the right of the assignment statement. Evaluation of the expression simply yields the address 10024 of the object referenced by variable a. By storing that address in the memory cell allocated to variable b, both variables a and b reference the same object. We say that these variables share the same object.

In the last assignment, variable c is assigned the value 450. The execution of this statement assigns yet another memory cell to variable c. That cell stores the address 10028 of a newly created object in which the value 450 is stored:

```
a = -300
b = a
c = 450
```

Python provides the operator **is** to check whether or not different expressions reference the same object. The operator relates an expression on the left to an expression on the right. It checks whether the objects resulting from the evaluation of the expressions are the same on both sides. In other words, the operator checks whether the addresses at which the resulting objects are stored are identical. The operator **is** either returns True or False. These values are part of Python's built-in type bool that we discuss in subsection 1.7.1.

Experiment 4 below shows some examples of expressions involving the operator **is**. From Figure 1 it is clear that the variables a and b reference the same object, whereas the variables a and c clearly do not. The variables x and y near the end of Experiment 4 below do not reference the same object. The value 10000 resulting from the expression on the right of the assignment to the variable x is stored in a new object. The value 10000 resulting from the expression on the right of the assignment to the variable y is stored in another new object. We say that the variables x and y reference different objects with the same content. The first part of Figure 2 illustrates this.

For small integer values, Python has pre-allocated objects storing their values. In the current version of Python, this is the case for all integer values in the range -5 .. 256. Expressions yielding such small integer values do not store their value in a newly allocated object. Instead, they re-use one of the pre-allocated objects. In Experiment 4 below, the value resulting from the evaluation of the right-hand side of the assignment to the variable i1 is a small integer value. The variable i1 therefore references the pre-allocated object storing the value 10. The evaluation of the expression on the right of the assignment to the variable i2 yields the same small integer value. The variable i2 therefore also references the pre-allocated object storing the value 10. Both variables therefore reference the same object, as illustrated in Figure 2, and as evidenced by the result of the expression i1 **is** i2:

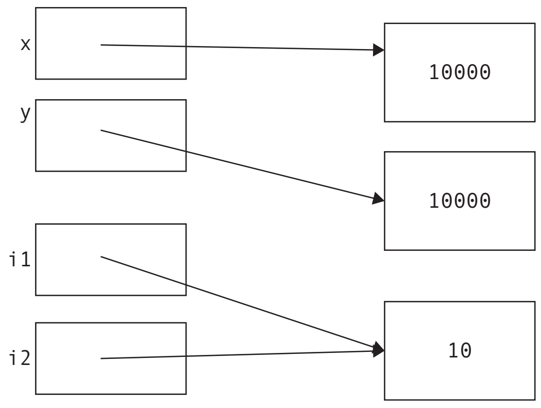

Figure 2: Semantics of the operator is.

```
>>> a = -300
>>> b = a
>>> a is b
True

>>> c = 450
>>> a is c
False

>>> x = 10000
>>> y = 10000
>>> x is y
False

>>> i1 = 10
>>> i2 = 10
>>> i1 is i2
True
```

Experiment 4: The operator **is**.

1.5.4 Style guide for naming variables

We have already mentioned that programs must pay proper attention to the names of variables they use. The name of each variable should reflect as well as possible the value it is intended to store. Programming languages also use style guides in spelling various kinds of ingredients. In this course, we follow the Google style guide for Python. For naming variables, the Google style guide suggests using only lower-case letters. If a name consists of several words, an underscore (_) separates successive words. Examples of variable names that satisfy the Google style guide are `balance`, `savings_account` and `library_members`.

The program in Example 1 to check whether a given year is a leap year stores the year to be examined in the variable `year`. An object storing that value is returned as the result of the invocation of the built-in function `int` on the string returned by the built-in function `input`.

1.6 Built-in Types

Programs must be able to compute with different types of data. Some types of data are needed in almost all programs. Examples are integer numbers, broken numbers and strings. Programming languages offer *built-in types* to support computations with such frequently occurring types of data. For other types of data, programmers may define classes, as we discuss in Chapter 18. A built-in type defines a collection of values complemented with functions and operators to compute with elements of that collection. A function takes some values of that type and delivers a result. We have already studied the built-in function `input` to read input. Another example is the built-in function `abs(x)` that takes a numerical value `x` and returns its absolute value. An example of an operator is the operator `+` that returns the sum of the value of its left-hand operand and the value of its right-hand operand.

1.7 Integer types

Python provides the built-in type `int` to compute with integer numbers. This type defines an unlimited collection of integer numbers and provides both positive and negative values of unlimited size. Apart from writing integers in the decimal representation that we are used to, Python provides the ability to write integer constants in binary, octal and hexadecimal representation. Computers use binary representation for integer numbers, because hardware switches inside computers only have two states. We discuss the various representations for integer numbers below. Python further provides an extended set of operators to compute with integer numbers. We also discuss all of them in this subsection.

Integer literals

Integer constants that we directly write in programs are called *integer literals*. We literally write the value of an integer into the program text, hence the term "integer literal". We customarily write integer values with a *decimal base*, i.e. with base `10`. An integer constant in that base consists of a sequence of digits ranging from `0` to `9`. The value of the integer literal $d_n d_{n-1} \ldots d_2 d_1 d_0$ in decimal is obtained as $d_n*10^n + d_{n-1}*10^{n-1} + \ldots + d_2*10^2 + d_1*10^1 + d_0*10^0$. For example, the value of the integer `157` is obtained as the sum `1*100+5*10+7`.

Experiment 5 below shows some integer literals in decimal. Note that integer literals may start with a sign. Except for small integer values, the evaluation of each integer literal that occurs in a Python program results in a new object that stores the value of the literal. That object can then be assigned to a variable, as illustrated in section 1.5. Objects that store integer values can also be used as operands of an arithmetic operator or as arguments of a function. We discuss this functionality at the end of this subsection.

Python also supports writing integer literals as binary, octal and hexadecimal numbers. A *binary number* uses `2` as its base. A binary literal starts with an optional sign followed by one of the markers `0B` or `0b`. It further consists only of the digits `0` and `1`. In computer science, binary digits are called *bits*. In fact, a bit is the smallest possible carrier of information. It can only have two values, represented as 0 and 1, as low and high, as on and off, etc. The value of the binary literal 0B$d_n d_{n-1}…d_2 d_1 d_0$ is obtained as d_n*2^n + d_{n-1}*2^{n-1} + … + d_2*2^2 + d_1*2^1 + d_0*2^0. For example, the value of the binary literal 0B10101 is obtained as the sum 1*16+0*8+1*4+*0*2+1.

Experiment 5 below shows some binary literals. Note that the Python shell always shows the value of integer literals in decimal, regardless of the base in which they have been written.

An *octal number* uses 8 as its base and the digits 0, 1, …, 7. In Python, an octal literal starts with an optional sign followed by one of the markers 0O or 0o, followed by a sequence of octal digits. The value of the octal literal 0O$d_n d_{n-1}…d_2 d_1 d_0$ is obtained as d_n*8^n + d_{n-1}*8^{n-1} + … + d_2*8^2 + d_1*8^1 + d_0*8^0. For example, the value of the octal literal 0O371 is obtained as the sum 3*64+7*8+1.

Experiment 5 below shows some octal literals in Python. In version 2 of the language, octal literals just started with a zero. For example, the literal 047 is an octal literal in Python 2 with value 4*8+7. This is rather confusing because it is easy to misunderstand the literal as the decimal literal 47. Python 3 therefore changed the way octal literals must be spelled. In order to avoid any further confusion, decimal literals involving more than 1 digit may not start with a zero. Experiment 5 below illustrates that the decimal literal 047 is no longer valid. The literal 0, on the other hand, is still a valid decimal literal.

A *hexadecimal number* uses 16 as its base. A hexadecimal literal starts with an optional sign followed by one of the markers 0X or 0x. Because we need 16 digits, hexadecimal literals use the digits 0, 1, …, 9 extended with the letters A or a (value 10), B or b (value 11), …, F or f (value 15). The value of a hexadecimal literal 0X$d_n d_{n-1}…d_2 d_1 d_0$ is obtained as d_n*16^n + d_{n-1}*16^{n-1} + … + d_2*16^2 + d_1*16^1 + d_0*16^0. For example, the value of the hexadecimal literal 0XA1 is obtained as the sum 10*16+1. Experiment 5 below shows some hexadecimal literals.

Python also provides the built-in function `int` to convert a given value to an integer value. In its simplest form, the function just takes an integer number, a floating-point number or a string as its argument. If a floating-point number is supplied, the conversion truncates towards zero. We discuss floating-point numbers in more detail in Chapter 2. If a string is supplied, it must be an integer literal, as they can occur directly in Python code. In a more extended form, the function `int` may involve a base that follows the string to be converted. In that case, the string must be an integer literal in the given base. The base can be any value in the range 2 to 36. The letters a to z represent digits with values 10 to 35. Literals in base 2, 8 or 16 may be, but are not obliged to be, prefixed respectively with 0b/0B, with 0O/0o, or with 0x/0X. If the function `int` cannot convert its input to an integer value, it raises a runtime error of type `ValueError`. Experiment 5 below shows some experiments with the built-in function `int`.

We can check whether a particular value is of a particular type by means of the built-in function `isinstance`. This function takes the value to be checked as its first argument, and the type as its second argument. It always returns a Boolean value. Experiment 5 below shows some experiments with the built-in function `isinstance`. In Python, `int` stands for the integer type; `str` stands for the string type:

```
# Decimal literals
>>> 423
423
>>> -78945
-78945
>>> +5
5
>>> 0
0

# Binary literals
>>> 0B101101
45
>>> -0b1100
-12

# Octal literals
>>> 0O47
39
>>> -0o14
-12
>>> 047
    ^
SyntaxError: invalid token

# Hexadecimal literals
>>> 0X3A
58
>>> -0xf0
-240

# Built-in function int
>>> int(24.56)
24
>>> int("251")
251
>>> int("13",8)
11
>>> int("1g",20)
36

# Built-in function isinstance
>>> isinstance(100,int)
True
>>> isinstance("13",int)
False
>>> isinstance(100,str)
False
```

Experiment 5: Integer literals.

Integer operators

Python provides the binary additive operators + and -. When both operands of an additive operator are integer values, the resulting value is also an integer value. The *addition operator* (+) yields the sum of its left- and right-hand operands. The *subtraction operator* (-) yields the difference between its left- and right-hand operands. The additive operators associate from left to right. This means that `a+b+c` is equivalent to `(a+b)+c`. Experiment 6 below starts with an expression involving additive operators.

Python also provides the binary multiplicative operators *, /, // and %. The *multiplication operator* (*) yields the product of its left- and right-hand operands. The multiplication operator yields an integer if both its operands are integer numbers. Python provides the *division operator* (/) and the *floor division operator* (//). The division operator always yields a non-rounded quotient represented as a floating-point number, regardless of whether or not its operands are integer numbers. For example, `10/4` yields `2.5` and `10/-4` yields `-2.5`. The floor division operator yields the quotient of the division of its left-hand integer operand by its right-hand integer operand rounded down to the nearest integer value. If both the left- and right-hand operand of the floor division operator are integer numbers, the resulting value is also an integer number. For example, the expressions `10//3` and `-10//-3` both yield `3`; the expressions `-10//3` and `10//-3` both yield `-4`. Division by zero results in a runtime error of type `ZeroDivisionError`.

The *modulo operator* (%) yields the remainder of the division of its left-hand operand by its right-hand operand. If the right-hand operand is zero, a runtime error of type `ZeroDivisionError` is raised. The modulo operator always yields a result with the same sign as its right-hand operand (or zero). Moreover, the absolute value of its result is smaller than the absolute value of its left-hand operand. The integer floor division and modulo operators are connected by the relationship `x == (x//y)*y +(x%y)`. For example, `20%7` yields `6`, `-20%-7` yields `-6`, `-20%7` yields `+1` and `20%-7` yields `-1`. The multiplicative operators associate from left to right. This means that `a*b/c` is equivalent to `(a*b)/c`. Multiplicative operators have a higher priority than the additive operators. This means that `a+b*c` is equivalent to `a+(b*c)`. Experiment 6 below shows some expressions involving multiplicative operators.

Python also provides the unary operators + and -. Unlike binary operators, unary operators only involve a single operand. The *unary operator* + has no effect. When applied to a value, it yields the same value. The unary operator – yields the negation of the value to which it is applied. The unary operators + and – have a higher priority than binary multiplicative operators. This means that `–a*b` is equivalent to `(-a)*b`. Experiment 6 below shows some expressions involving the unary operators + and -.

The *power operator* ** yields its left-hand operand raised to the power specified by its right-hand operand. If the left-hand operand is an integer value, and the right-hand operand is a non-negative integer value, the resulting value is an integer. For example, `3**2` yields `9` and

(-3)**3 yields -27. If the left-hand operand is an integer value, and the right-hand operand is a negative integer value, the resulting value is a floating-point value. The value of a**-b is always equal to 1.0/(a**b). For example, 2**-3 yields 0.125 and (-2)**-2 yields 0.25. Raising 0 to a negative power results in a runtime error of type ZeroDivisionError. The power operator has a higher priority than the unary operators – and + to the left of the power operator. This means that –a**b is equivalent to –(a**b). The power operator has a lower priority than the unary operators to its right. This means that a**-b is equivalent to a**(-b). Experiment 6 below shows some expressions involving the power operator.

Comparison operators serve to compare two integer values. Python provides equality (==) to check whether its left operand has the same value as its right operand, inequality (!=) to check whether its left operand has a different value than its right operand, less than (<) to check whether its left operand has a smaller value than its right operand, less than or equal (<=) to check whether its left operand has a smaller or equal value than its right operand, greater than (>) to check whether its left operand has a greater value than its right operand, and greater than or equal to (>=) to check whether the value of its left operand is greater than, or equal to, that of its right operand. The result of a comparison operator is one of the Boolean values True or False. Comparison operators have a lower priority than additive operators. This means that 3+4<6+7 is equivalent to (3+4)<(6+7). Experiment 6 below shows some expressions involving comparison operators.

Python provides augmented versions of assignment. They combine the semantics of operators applicable to integer values with the semantics of assignment. In particular, an *augmented assignment* consists of the name of a variable on the left, an expression on the right, and one of the augmented assignment operators +=, -=, *=, /=, //=, %= or **= in between. At this point, we assume that both the current value of the variable on the left and the value resulting from the evaluation of the expression on the right are integer values. The current value of the variable on the left then acts as the left-hand operand of the operator involved in the augmented assignment operator. The expression on the right acts as the right-hand operand. The resulting value is then registered as the new contents of the variable on the left. In other words, a statement in the form x q= y, where q denotes some operator, is equivalent to the more expanded statement x = x q y. Experiment 6 below ends with some augmented assignments:

```
# Additive operators
>>> 24+30-7
47

# Multiplicative operators
>>> 20+3*7*2
62
>>> -20//6-3*4//5
-6
>>> -20//6*6+-20%6
-20
```

```
# Unary operators
>>>+20*---5
-100

# Power operator
>>> -4**2 + (-4)**(-2) + 0**0
-14.9375

# Comparison operators
>>> 3 < 4-3
False
>>> 4 != 8/2
False

# Augmented assignment
>>> x = 20
>>> x *= 4
>>> x
80
>>> x %= 11
>>> x
3
```

Experiment 6: Integer operators.

The example program to determine whether a given year is a leap year computes with integer values. Once the year in question has been read in, the program examines whether the given integer value is a multiple of 4, a multiple of 100 and a multiple of 400. As illustrated in Example 1, the program uses the modulo operator (`%`) for that purpose. In particular, the program checks whether the remainder obtained by dividing the given year by 4 is equal to 0. It then does the same for divisions of the given year by 100 and by 400.

The result of each of these comparisons is stored respectively in the variables `is_4_multiple`, `is_100_multiple` and `is_400_multiple`. Each of the variables `is_4_multiple`, `is_100_multiple` and `is_400_multiple` stores one of the Boolean values `True` or `False`. Note once more how well chosen names for variables improve the readability of our programs. Imagine for a moment how the program would look if we had chosen names such as `c1`, `c2` and `c3` for these variables. Readers would then have to figure out themselves what kind of value is stored in each of these variables.

1.7.1 Boolean type

The type `bool` has just two possible values: `True` and `False`. It is named after the English mathematician George Boole, who worked out the basic ingredients of what is now called the Boolean logic. In retrospect, George Boole is regarded as the founder of the field of digital electronics. Expressions involving Boolean values are used to steer the order in which statements are executed based on logical expressions or predicates. The comparison operators discussed in section 1.7 yield Boolean values. However, Boolean values can also be assigned to variables, as illustrated in Experiment 7 below.

The *conjunction* of two Boolean expressions is denoted by the binary operator **and**. A conjunction yields `True` if and only if both the expression on the left and the expression on the right evaluate to `True`. The *disjunction* of two Boolean expressions is denoted by the binary operator **or**. A disjunction yields `False` if and only if both the expression on the left and the expression on the right evaluate to `False`. Finally, the unary operator **not** denotes the *negation* of a Boolean expression. A negation yields `False` if the expression to which it is applied evaluates to `True`, and vice versa.

Boolean operators have a lower priority than comparison operators. This means that `3<6 and 5>7` is equivalent to `(3<6) and (5>7)`. The Boolean operator **not** has a higher priority than the Boolean operator **and**. This means that `not a and b` is equivalent to `(not a) and b`. The Boolean operator **and** has a higher priority than the Boolean operator **or**. This means that `a or b and c` is equivalent to `a or (b and c)`. Experiment 7 below shows some expressions involving Boolean operators.

Python uses the technique of *short-circuit evaluation* to optimize the evaluation of conjunctions and disjunctions. Whenever the expression on the left of a conjunction evaluates to `False`, Python does not evaluate the expression on the right. The value of the latter expression does not change the end result of the conjunction, which by definition is `False` in this case. Similarly, Python does not evaluate the right-hand side of a disjunction if the evaluation of its left-hand side has yielded `True`. In that case, the end result of the disjunction is `True`. Apart from being more efficient, short-circuit evaluation also makes life easier for the Python programmer. It can avoid runtime errors, as illustrated by the expression `(x == 0) or (y//x >= 0)` in Experiment 7 below. If the variable `x` were `0`, evaluation of the right-hand side would result in division by `0`. Because of short-circuit evaluation, the right-hand side is not evaluated in this case.

Lots of programs need to check whether a value is in a particular range. We can write such conditions using the comparison operators that we discussed in the previous section combined with a conjunction. In general, the expression `(a <= x) and (x <= b)` checks whether the value of the variable `x` is in the range `a..b`. We can write such expressions in a more compact way by *chaining comparison operators*. Specifically, the expression `(a <= x <= b)` checks whether the variable `x` is in the range `a..b`. It is completely equivalent to the more verbose expression involving the operator **and**. Experiment 7 below shows a more concrete example.

Integer values and Boolean values can be mixed. The Boolean values `True` and `False` are interpreted respectively as the integer values `1 and 0` in any context in which an integer value is expected. Experiment 7 below illustrates this with an integer addition that takes `True` on the left and a multiplication involving `False` on the right. The reverse is also true. Any non-zero integer value is interpreted as `True` in any context in which a Boolean value is expected. Similarly, the integer value 0 is interpreted as `False` in any context in which a Boolean value is expected. Experiment 7 below illustrates this with a conjunction involving the integer 42 on the left, and with a disjunction involving 0 on the left. If we had supplied the integer values on the right of the Boolean operators, they would have returned them as a result. Indeed, the Boolean operators **and** and **or** always return the value of the operand that has been evaluated last. For example, the expression `True and 42` returns 42:

```
# Boolean literals
>>> x = True
>>> x
True

# Conjunction
>>> x = True
>>> x and (5<6)
True

# Disjunction
>>> y = False
>>> y or (3!=4)
True

# Negation
>>> x = True
>>> not x
False

# Short-circuit evaluation
>>> x = 0
>>> y =24
>>> (x==0) or (y/x >= 0)
True

# Chaining comparison operators
>>> x = 20
>>> -50 <= x <= 50
True

# Equivalence with integer numbers
>>> True + 12
13
>>> 42*False
0
>>> 42 and True
True
>>> 0 or False
False
```

Experiment 7: Boolean operators.

As noted above, the program shown in Example 1 to check whether a given year is a leap year stores the results of examining respectively whether the given year is a multiple of 4, a multiple of 100 and a multiple of 400 in Boolean variables. The expressions on the right of each of these assignments are comparisons checking whether the remainder in question is equal to 0. The if statement following those assignments also uses a Boolean expression. We discuss the full semantics of if statements in section 1.8.2.

The Boolean expression following the keyword **if** first of all uses the operator **and** to check that the year in question is both a multiple of 4 and not a multiple of 100 or a multiple of 400. The right-hand side of that rather complex condition is expressed using the Boolean operators **not** and **or**. Because of short-circuit evaluation, the right-hand side of the operator **and** is not evaluated if the year in question is not a multiple of 4. For the same reason, the right-hand side of the operator **or** is not evaluated if the year in question is not a multiple of 100.

Note that the entire Boolean expression has not been written on a single line. There is no hard limit on the length of a line in a Python program. However, common sense dictates a practical limit of 80 characters, because lines with more characters do not always print properly on paper. As illustrated by the condition, we can use the so-called *continuation character* (\) to spread expressions and other ingredients of Python programs over several lines. Python further allows parenthesized expressions to be spread over several lines, provided the line break is in between the parentheses. Experiment 8 below illustrates this functionality with a simple addition. The addition itself is between parentheses, and its right-hand side starts on a new line. The bottom part of the experiment shows that line breaks are only allowed inside parenthesized expressions. The idea of the example was to continue the expression on the next line to give the right-hand side of the addition. However, the Python interpreter signals an error after having processed the first line. Because the expression is not between parentheses, it must all be given on a single line:

```
>>> (3+
4)
7
>>> 3+
SyntaxError: invalid syntax
```

Experiment 8: Spreading expressions over several lines.

1.8 Selection

The programs we have been writing so far merely consist of a sequence of instructions. The instructions of such programs are executed one by one in the order in which they occur.

We often want to influence the sequence of instructions that our programs execute. The order in which the instructions of a program are executed is called the *control flow*. Programming languages offer constructs to influence that flow. This section discusses the if statement as one of Python's statements to influence the control flow. Programming languages often use the so-called *Backus-Naur Form* (BNF) or some extended form of it to clarify the syntax – the spelling – of their constructs. We therefore start this section with a short introduction to the Backus-Naur Form and illustrate it with the syntax of assignment statements and expressions that we have discussed in previous sections.

1.8.1 Backus-Naur Form

John Backus was one of the first to propose a notation to describe the syntax of programming languages. He introduced it in 1959 to clarify the syntax of the programming language Algol 58. In the early sixties, Peter Naur simplified it to a notation that is now commonly known as the Backus-Naur Form. A BNF specification consists of a number of rules. Each BNF rule describes a particular construct of the programming language in question. Such a rule consists of a nonterminal symbol on the left and a BNF expression on the right. The parts are separated from each other by means of the *BNF define operator* (`::=`) Because we do not always discuss the full semantics of Python constructs all at once, we often need to extend the rule defining some nonterminal symbol in later chapters. In order to distinguish the final definition of a nonterminal symbol from a temporary definition, we use the *BNF temporary define operator* (~~=) for temporary definitions of nonterminal symbols that are completed later in the book.

In their simplest form, BNF Expressions are sequences of terminal symbols and nonterminal symbols. *Nonterminal symbols* are always enclosed in angular brackets (<...>). *Terminal symbols* are usually written as such. However, from time to time, the spelling of a terminal symbol can be confused with with BNF symbols. For that reason, terminal symbols may also be enclosed in double quotes. Syntax rule 1 below describes the syntax of a simple assignment statement in Python as discussed in section 1.5. The BNF rule temporarily defines the nonterminal `<Assignment Statement>` as a construct that starts with the name of a variable, followed by the assignment operator =, and terminated with an expression. The symbols `variable_name` and = are terminal symbols. The spelling of the expression on the right of an assignment must satisfy the BNF rule defining the nonterminal symbol `<Expression>`. Because we have used the temporary operator ~~=, a BNF rule defining the syntax of more extended forms of assignment statements will follow in later chapters:

```
<Assignment Statement> ~~=
    variable_name = <Expression>
```

Syntax rule 1: Assignment statement.

BNF expressions can use operators to describe more complex constructs. The *BNF choice operator* | is available to describe alternatives. A BNF expression in the form e_1 | e_2 | … | e_n specifies that the spelling must be according to one of the BNF expressions e_1, e_2, …, e_n. Syntax rule 2 below shows the final definition of the nonterminal symbol `<Statement>`. It specifies that a statement in Python is either a simple statement or a compound statement. A simple statement is temporarily defined as either an assignment statement or an augmented assignment statement. We introduce other kinds of simple statements later on. The BNF choice operator may also be used in subexpressions as illustrated in the definition of the nonterminal `<Augmented Assignment Statement>`. It enumerates all possible augmented assignment operators in a BNF subexpression that involves the choice operator. BNF subexpressions are typically enclosed between parentheses:

```
<Statement> ::=
    <Simple Statement>
  | <Compound Statement>

<Simple Statement> ~~=
    <Assignment Statement>
  | <Augmented Assignment Statement>

<Augmented Assignment Statement> ~~=
  variable_name
    ( += | -= | *= | /= | //= | %= | **= ) <Expression>
```

Syntax rule 2: Statement, Simple Statement and Augmented Assignment Statement.

The *BNF iterative operators* `*` and `+` are used to specify that a certain component may appear a number of times in succession. The BNF expression `e*` specifies that the component described by the subexpression `e` can occur an arbitrary number of times in succession. The BNF expression `e+` specifies that the component described by the subexpression `e` must occur at least once and may be repeated an arbitrary number of times immediately afterwards. Syntax rule 3 below describes the syntax of expressions. A numeric expression consists of a multiplicative term followed by zero or more sequences of an additive operator and another multiplicative term. In the same way, a multiplicative term consists of a unary term followed by zero or more sequences of a unary operator and another unary term. Note that the BNF rules for numeric expressions fully describe the priority rules that Python imposes on numeric operators. For example, the expression `a*b+c/d` can only be interpreted as the multiplicative term `a*b` followed by the operator `+` followed by the multiplicative term `c/d`.

BNF Expressions may also enclose subexpressions in square brackets. A squared BNF subexpression `[e]` denotes an *optional component*. In Syntax rule 3 below, the BNF rule describing power terms uses this facility to specify that a power term consists of a basic numeric term optionally followed by the power operator and a unary term. In this way, the definition reflects the right-to-left association of the power operator `**`. Indeed, an expression in the form `a**-b**c` must be interpreted as the basic numeric expression `a` followed by the power operator `**` followed by the unary term `-b**c`.

Finally, the BNF rule defining basic numeric terms in Syntax rule 3 below needs some further explanation. Specifically, it illustrates the need to enclose terminal symbols in double quotes. Indeed, one of the alternatives for a basic numeric term is a numeric expression between parentheses. If we were to have written this alternative simply as `( <Numeric Expression> )` instead of `"(" <Numeric Expression> ")"`, the parentheses would be interpreted as BNF parentheses enclosing the BNF subexpression `<Numeric Expression>`, rather than as terminal symbols enclosing a numeric expression in Python. For similar reasons, the terminal symbols `+` and `*` are enclosed in double quotes in the BNF rules describing respectively numeric expressions and multiplicative terms:

```
<Expression> ~~=
    <Numeric Expression>
  | <Boolean Expression>

<Numeric Expression> ::=
  <Multiplicative Term>
    ( ( "+" | - ) <Multiplicative Term> )*

<Multiplicative Term> ::=
  <Unary Term> ( ( "*" | / | // | % ) <Unary Term> )*

<Unary Term> ::=
  ( + | - )* <Power Term>

<Power Term> ::=
  <Basic Numeric Term> [ ** <Unary Term> ]

<Basic Numeric Term> ~~=
    integer_literal
  | floating_point_literal
  | variable_name
  | "(" <Numeric Expression> ")"

<Boolean Expression> ::=
  <Conjunctive Term> ( or <Conjunctive Term> )*

<Conjunctive Term > ::=
  <Negated Term> ( and <Negated Term> )*

<Negated Term > ::=
  [ not ] <Basic Boolean Term>
```

```
<Basic Boolean Term> ~~=
  | True
  | False
  | variable_name
  | "(" <Boolean Expression> ")"
  | <Comparison Expression>

<Comparison Expression> ::=
  <Expression> ( <Comparison Operator> <Expression> )+

<Comparison Operator> ~~=
  == |!= | <> | < | <= | > | >=
```

Syntax rule 3: Expression, Numeric Expression and Boolean Expression.

1.8.2 The if statement

An *if statement* in Python serves to state that a group of statements may only be executed if a particular, stated condition is satisfied. Syntax rule 4 below states that an if statement always starts with the keyword **if**, followed by a Boolean expression, a colon and a statement group. That part of an if statement is often called its if part or its then part. The execution of an if statement starts with the evaluation of the Boolean expression of its *if part*. If the resulting value is `True`, the statement group of that then part is executed. Hereafter, execution proceeds with the statement following the entire if statement.

Syntax rule 4 below further specifies that the if part of an if statement may be followed by an arbitrary number of *elif parts*, short for else-if parts. An elif part consists of the keyword **elif** followed by a Boolean expression, a colon and a statement group. If the evaluation of the Boolean expression following the keyword **if** has yielded `False`, execution of the if statement proceeds with the evaluation of the Boolean expression that is part of the first elif part, if available. If the resulting value is `True`, the statement group following that Boolean expression is executed. Execution then proceeds with the statement following the entire if statement.

If the evaluation of the Boolean expression following the first occurrence of the keyword **elif** has yielded `False`, execution of the if statement proceeds with the evaluation of the Boolean expression that is part of the next elif part. The statement group associated with that elif part is executed if the evaluation of its Boolean expression yields `True`. In this way, execution of the if statement steps from one elif part to the next elif part until either the statement group of an elif part is selected for execution, or until the evaluation of all Boolean expressions of all elif parts has yielded `False`.

The final component of an if statement is an optional *else part* at the end. According to Syntax rule 4 below, that part consists of the keyword **else** immediately followed by a colon and a statement group. No Boolean expression is involved in this part. If the evaluation of the Boolean expressions of the then part and of all the elif parts of an if statement has yielded `False`, the statement group of its else part, if any, is executed. If the if statement has no else part, execution immediately proceeds with the statement following the entire if statement:

```
<If Statement> ::=
    if <Boolean Expression> :
      <Statement Group>
  ( elif <Boolean Expression> :
      <Statement Group> ) *
  [ else:
      <Statement Group> ]

<Statement Group> ::=
  <Statement> +
```

Syntax rule 4: If Statement and Statement Group

The program shown in Example 1 to check whether a given year is a leap year uses an if statement. That if statement only consists of a then part and of an else part. If the condition of the then part evaluates to `True`, the program prints out that the given year is a leap year. Otherwise, the else part is executed. In that case, the program prints out that the given year is not a leap year. The example that we develop in Example 2 involves a more extended if statement. It also involves some elif parts.

So far, we have been rather vague about the notion of a statement group. According to Syntax rule 4 above, a *statement group* is a non-empty sequence of statements. Python uses indentation to delineate statement groups. In computer science, *indentation* is defined as the rightward displacement of programming text to separate it clearly from surrounding text. Programming languages have been using indentation for a long time to improve the readability of their programs. Indentation makes the structure of the program stand out. In Python, we are free to use spaces or tabs to indent code. However, we must be consistent in our choice throughout the program text. Because tabs may lead to problems when we send our program to a printer or to some other machine, we recommend to use spaces to indent Python code. In Example 1, the statement group that belongs to the then part and the else part of the if statement just consists of a single statement. They are both shifted to the right by means of two spaces. In this way, the different alternatives belonging to the if statement clearly stand out.

In most programming languages, indentation serves no other purpose than to improve the readability of their programs. These languages use brackets such as {…} or keywords such as **begin** and **end**, or **do** and **od** to delineate statement groups. Python goes one step further than most programming languages. It uses indentation to delineate statement groups. The language demands that statement groups such as those belonging to the different parts of an if statement are indented to the right. Moreover, each singular statement in a statement group must be indented to the right over the same distance. We are free to choose the number of spaces we use for each level of indentation. Common values are 2 or 4 spaces. We use two spaces to indent all code fragments in this book. In Example 1, each invocation of the built-in function print is indented to the right. Those statements are therefore the first statement of a statement group. Because none of them is followed by another statement indented to the right over the same distance, both statement groups consist of only one statement.

1.9 Output

In section 1.4, we discussed the semantics of the built-in function `input` to read data that is supplied by the end-user. That data is read from the standard input stream. Similarly, Python provides the built-in function `print` to display results to end-users. The resulting data is written to the standard output stream. The function involves an arbitrary sequence of expressions separated by commas. It evaluates all expressions in succession, and prints their values to the standard output stream. By default, successive values displayed to the standard output stream are separated from each other by a space. After all values have been written, the standard output stream turns to a new line unless specified otherwise. Experiment 9 below starts with a simple example of an invocation of the built-in function `print`.

We may want to use other symbols than a simple space to separate successive values printed to the standard output stream. In that case, we must use a so-called keyword argument named `sep` to which we assign the string that must be used as a separator between all the values printed out as a result of executing the built-in function `print`. We explain the semantics of keyword arguments in more detail in 5.3. The second example in Experiment 9 below illustrates the use of a self-supplied separator. This time the string `"John's score"` is separated from the integer number `42` by means of a colon followed by a space.

After the invocation of the built-in function print has written all the values, it turns to a new line unless we supply another symbol to follow the output. This time, we must use a keyword argument named `end` to which we assign the string that must follow the last printed value. By default, that string is equal to `"\n"`, which stands for a new line. The last example in Experiment 9 below illustrates how to supply another sequence of characters following the last printed value. This time, we use two successive invocations of the built-in function `print` to print the string `"John's score: 42"` on a single line. The first invocation specifies that the string `": "` must follow `"John's score"`. The second invocation continues printing the integer number `42` on the same line. Because that invocation does not involve a self-supplied terminating string, later invocations of the built-in function `print` will start printing on the next line:

```
>>> print("John's score:",42)
John's score: 42

>>> print("John's score",42,sep=": ")
John's score: 42

print("John's score",end=": ")
print(42)
John's score: 42
```

Experiment 9: The built-in function `print`.

The program to check whether a given year is a leap year terminates with some invocations of the built-in function `print`. As illustrated in Example 1, both invocations involve two strings that are printed as such to the standard output stream. The given year is printed between the strings. Whenever something other than a string is to be printed to the standard output stream, a textual representation of that thing is written out. That textual representation is obtained from the built-in function `str`. In the example, the latter function is invoked with the given year, and returns a textual representation of the given integer number. We can influence the way all these strings are eventually written out by supplying formatting information. For example, we could say that the given year must always be six characters long. Spaces (or other symbols that we specify) would then be added in front of the textual representation of the given year. However, we do not further discuss the formatting of output in this book.

The code snippet below shows an alternative for the last part of the program in Example 1. This time, the common parts of both invocations of the built-in function `print` have been factored out in a single invocation that precedes the if statement. The example illustrates the use of a self-defined ending symbol after the last expression in an invocation of the built-in function print. Because of the comma after the year in the invocation of the built-in function print preceding the if statement, the invocation of the built-in function prints in the then part and in the else part of the if statement continues printing on the same line:

```
print("The year", year, end =" ")
if is_4_multiple and \
     ( (not is_100_multiple) or is_400_multiple):
  print("is a leap year!")
else:
  print("is not a leap year!")
```