Textbook

https://jrembold.github.io/Website_Backup/class_files/cs151/CS151-Reader-2023.pdf

Arithmetic/PEMDAS:

https://cd-public.github.io/courses/cs1f24/slides/w3d1_py2.html#/2

https://www.geeksforgeeks.org/precedence-and-associativity-of-operators-in-python/

https://www.w3schools.com/python/python_operators.asp

Booleans:

https://cd-public.github.io/courses/cs1f24/slides/w3d2_range.html#/13

https://www.geeksforgeeks.org/boolean-data-type-in-python/

https://www.w3schools.com/python/python_booleans.asp

Strings:

https://cd-public.github.io/courses/cs1f24/slides/w4d2_str1.html
https://cd-public.github.io/courses/cs1f24/slides/w4d3_str2.html

https://www.geeksforgeeks.org/python-string/

https://www.w3schools.com/python/python_strings.asp

if/def/for/while:

https://cd-public.github.io/courses/cs1f24/slides/w2d2_karel4.html#/31

https://www.geeksforgeeks.org/python-if-else/
https://www.geeksforgeeks.org/python-for-loops/
https://www.geeksforgeeks.org/python-functions/

https://www.w3schools.com/python/python_conditions.asp
https://www.w3schools.com/python/python_for_loops.asp
https://www.w3schools.com/python/python_functions.asp

For string data, the domain comprises sequences of characters that appear on the keyboard or that can be displayed on the screen. The ***set of operations*** is the toolbox that allows you to manipulate values of that type. For numeric data, the set of operations includes addition, subtraction, multiplication, and division, along with a variety of more sophisticated functions. For string data, however, it is hard to imagine what an operation like subtraction might mean. Using string data requires a different set of operations, such as combining two strings to form a longer one or comparing two strings to see if they are in alphabetic order. The general rule is that the set of operations must be appropriate to the elements of the domain. The two components together—the domain and the operations—define a ***data type.***

1.2 Numeric data

Computers today store data in so many exciting forms that numbers may seem a bit boring. Even so, numbers are a good starting point for talking about data, mostly because they are both simple and familiar. You've been using numbers, after all, ever since you learned to count. Moreover, as you'll discover in Chapter 7, all information is represented inside the computer in numeric form.

Representing numbers in Python

One of the important design principles of modern programming languages is that concepts that are familiar to human readers should be expressed in an easily recognizable form. Like most languages, Python adopts that principle for numeric representation, which means that you can write numbers in a Python program in much the same way you would write them anywhere else.

In their most common form, numbers consist of a sequence of digits, optionally containing a decimal point. Negative numbers are preceded by a minus sign. For example, the following are all legal numbers in Python:

```
0    42    -273    3.14159265    -0.5    1000000
```

Note that large numbers, such as the value of one million shown in the last example, are written without using commas to separate the digits into groups of three.

Numbers can also be written in a variant of scientific notation, in which the value is represented as a number multiplied by a power of 10. To express a value in scientific notation, you write a number in standard decimal notation, followed immediately by the letter `E` and an integer exponent, optionally preceded by a `+` or `-` sign. For example, the speed of light is approximately 2.9979×10^8 meters per second, which can be written in Python as

```
2.9979E+8
```

In Python's scientific notation, the letter `E` is shorthand for *times 10 to the power.*

Like most languages, Python separates numbers into two classes: ***integers,*** which represent whole numbers, and ***floating-point*** numbers, which contain a decimal point. Integers have the advantage of being exact. Floating-point numbers, by contrast, are approximations whose accuracy is determined by hardware limitations. Fortunately, Python also defines its mathematical operators in a way that makes it less important than it is in most languages to pay attention to the distinction between these two types of numbers.

In addition to integers and floating-point numbers, Python defines a third type of numeric data used to represent ***complex numbers,*** which combine a real component and an imaginary component corresponding to the square root of –1. Although complex numbers are beyond the scope of this text, the fact that Python includes complex numbers as a fully supported, built-in type makes Python especially attractive for scientific and mathematical applications in which complex numbers play an important role.

Arithmetic expressions

The real power of numeric data comes from the fact that Python allows you to perform computation by applying mathematical operations, ranging in complexity from addition and subtraction up to highly sophisticated mathematical functions. As in mathematics, Python allows you to express those calculations through the use of operators, such as `+` and `-` for addition and subtraction.

If you want to understand how Python works, the best approach is to use the Python interpreter, which is called IDLE. (Van Rossum claims that the name is an acronym of Integrated DeveLopment Environment, but the common assumption is that the name honors Monty Python's Eric Idle.) IDLE allows you to enter Python expressions and see what values they produce

Tom Lehrer

To get a sense of how interactions with IDLE work, suppose that you want to solve the following problem, which the singer-songwriter, political satirist, and mathematician Tom Lehrer proposed in his song "New Math" in 1965:

$$\begin{array}{r} 3\,4\,2 \\ -\,1\,7\,3 \\ \hline \end{array}$$

To find the answer, all you have to do is enter the subtraction into IDLE, as follows:

IDLE

```
>>> 342 - 173
169
>>>
```

This computation is an example of an ***arithmetic expression,*** which consists of a sequence of values called ***terms*** combined using symbols called ***operators,*** most of which are familiar from elementary-school arithmetic. The arithmetic operators in Python include the following:

`- a`	Negation (multiply a by –1 to reverse its sign)
`a + b`	Addition (add a and b)
`a - b`	Subtraction (subtract b from a)
`a * b`	Multiplication (multiply a and b)
`a / b`	True division (divide a by b)
`a // b`	Floor division (`a / b` rounded down to the next integer)
`a % b`	Remainder (compute the mathematical result of a mod b)
`a ** b`	Exponentiation (raise a to the b power)

Although most of these operators should be familiar from basic arithmetic, the `//` and `%` operators require additional explanation. Intuitively, these operators compute the quotient and remainder, respectively, when one value divided by another. For example, `7 // 3` has the value 2, because 7 divided by 3 leaves a whole number quotient of 2. Similarly, `7 % 3` has the value 1, because 7 divided by 3 leaves a remainder of 1. If one number is evenly divisible by another, there is no remainder, so that, for example, `12 % 4` has the value 0.

Unlike almost every other programming language, Python defines `//` and `%` for negative operands so that the result is consistent with mathematical convention. The `//` operator computes the result by performing an exact division and then rounding the result down to the next smaller integer. In mathematics, rounding a number down to the closest integer is called computing its ***floor.*** For example, the expression `-9 // 5` has the value –2, because exact division produces –1.8, and the floor of –1.8 is –2. In computing the remainder, the `%` operator applies what mathematicians call the ***mod*** operator, which always has the same sign as the divisor. The `//` and `%` operators are related by the following equivalence:

$$x \equiv (x \,//\, y) \times y + x \,\%\, y$$

Even though Python's definition of these operators makes mathematicians happy, the programs in this text use the `//` and `%` operators only with positive integers, where the result corresponds to the notions of quotient and remainder that you learned in elementary school. In part, the reason for this design decision is to avoid making programming seem more mathematical than it in fact is. In addition, it is dangerous to rely on how these operators behave with negative numbers because Python's definition—although it is clearly correct in mathematical terms—differs from how remainders are defined in other languages. If you write a Python program that relies on this behavior, it will be hard to translate that program into a language that uses a different interpretation.

Mixing types in an expression

Python allows you to mix integers and floating-point numbers freely in an expression. If you do so, the type of the result depends both on the operator and the types of the values to which it applies, which are called its ***operands.*** For almost all of Python's operators, the result is an integer if both operands are integers and a floating-point number if either or both of its operands is floating-point. Thus, evaluating the expression

```
17 + 25
```

produces the integer 42. By contrast, the expression

```
7.5 - 4.5
```

produces the floating-point value 3.0, even though the result is a whole number.

There are two exceptions to Python's standard rule for combining types. The `/` operator, which performs exact division, always returns a floating-point result, even if both operands are integers. The `**` operator is a bit more complicated. The result is an integer if the left operand is an integer and the right operand is a nonnegative integer. In any other case, the result is a floating-point value. For example, the expression

```
2 ** 10
```

calculates 2^{10} and therefore produces the integer 1024. The expression

```
2 ** -1
```

calculates 2^{-1}, which is the floating-point number 0.5.

Precedence

Following the conventions of standard mathematics, multiplication, division, and remainder are performed before addition and subtraction, although you can use parentheses to change the evaluation order. For example, if you want to average the numbers 4 and 7, you can enter the following expression into IDLE:

If you leave out the parentheses, Python first divides 7 by 2 and then adds 4 and 3.5 to produce the value 7.5, as follows:

```
IDLE
>>> 4 + 7 / 2
7.5
>>>
```

The order in which Python evaluates the operators in an expression is governed by their ***precedence,*** which is a measure of how tightly each operator binds to its operands. If two operators compete for the same operand, the one with higher precedence is applied first. If two operators have the same precedence, they are applied from left to right. The only exception is the exponentiation operator `**`, which is applied from right to left. Computer scientists use the term ***associativity*** to indicate whether an operator groups to the left or to the right. Most operators in Python are ***left-associative,*** which means that the leftmost operator is evaluated first. In Python, the only exception to this rule is the `**` operator, which is ***right-associative*** and groups from right to left.

Figure 1-1 shows a complete precedence table for the Python operators, many of which you will have little or no occasion to use. As additional operators are introduced in this book, you can look them up in this table to see where they fit in the precedence hierarchy. Since the purpose of the precedence rules is to ensure that Python expressions obey the same rules as their mathematical counterparts, you can usually rely on your intuition. Moreover, if you are ever in any doubt, you can always include parentheses to make the order of operations explicit.

FIGURE 1-1 Complete precedence table for the Python operators

Operators in decreasing order of precedence
**
unary operators: + − ~
* / // %
+ −
<< >>
&
^
|
== != > >= < <= is is not in not in
not
and
or

2.1 Boolean data

George Boole

In Python, you express conditions by constructing expressions whose values are either true or false. Such expressions are called ***Boolean expressions,*** after the English mathematician George Boole, who developed an algebraic approach for working with data of this type. Boolean values are represented in Python using a built-in type whose domain consists of exactly two values: `True` and `False`.

Python defines several operators that work with Boolean values. These operators fall into two classes—relational operators and logical operators—as described in the next two sections.

Relational operators

The simplest questions you can ask in Python are those that compare two data values. You might want, for example, to determine whether two values are equal or whether one is greater than or smaller than another. Traditional mathematics uses the operators $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$ to signify the relationships *equal to, not equal to, less than, greater than, less than or equal to,* and *greater than or equal to,* respectively. Because several of these symbols don't appear on a standard keyboard, Python represents these operators in a slightly different form, which uses the following character combinations in place of the usual mathematical symbols:

`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

Collectively, these operators are called ***relational operators*** because they test the relationship between two values. Like the arithmetic operators introduced in Chapter 1, relational operators appear between the two values to which they apply. For example, if you need to check whether the value of `x` is less than 0, you can use the expression `x < 0`.

Logical operators

In addition to the relational operators, which take values of any type and produce Boolean results, Python defines three operators that take Boolean operands and combine them to form other Boolean values:

`not`	Logical not (`True` if the following operand is `False`)
`and`	Logical and (`True` if both operands are `True`)
`or`	Logical or (`True` if either or both operands are `True`)

These operators are called ***logical operators*** and are listed in decreasing order of precedence.

Although the operators `and`, `or`, and `not` correspond to the English words *and*, *or*, and *not,* it is important to remember that English is somewhat imprecise when it comes to logic. To avoid that imprecision, it helps to think of these operators in a more formal, mathematical way. Logicians define these operators using ***truth tables,*** which show how the value of a Boolean expression changes as the values of its operands change. For example, the truth table for the `and` operator, given Boolean values `p` and `q`, is

p	q	p and q
False	False	False
False	True	False
True	False	False
True	True	True

The last column of the table indicates the value of the Boolean expression `p and q`, given the individual values of the Boolean variables `p` and `q` shown in the first two columns. Thus, the first line in the truth table shows that when `p` is `False` and `q` is `False`, the value of the expression `p and q` is also `False`.

The truth table for `or` is

p	q	p or q
False	False	False
False	True	True
True	False	True
True	True	True

Even though the `or` operator corresponds to the English word *or,* it does not indicate *one or the other*, as it often does in English, but instead indicates *either or both*, which is its mathematical meaning.

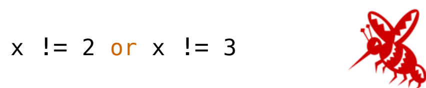
The `not` operator has the following simple truth table:

p	not p
False	True
True	False

If you need to determine how a more complex logical expression operates, you can break it down into these primitive operations and build up a truth table for the individual pieces of the expression.

In most cases, logical expressions are not so complicated that you need a truth table to figure them out. The only case that often causes confusion is when the `not` operator comes up in conjunction with `and` or `or`. When English speakers talk about situations that are not true (as is the case when you work with the `not` operator), a statement whose meaning is clear to human listeners is often at odds with mathematical logic. Whenever you find that you need to express a condition involving the word *not,* you should use extra care to avoid errors.

As an example, suppose you wanted to express the idea "x is not equal to either 2 or 3" as part of a program. Just reading from the English version of this conditional test, new programmers are likely to code this expression as follows:



As noted in Chapter 1, this book uses the bug symbol to mark sections of code that contain deliberate errors. In this case, the problem is that an informal English translation of the code does not correspond to its interpretation in Python. If you look at this conditional test from a mathematical point of view, you can see that the expression is `True` if either (a) `x` is not equal to 2 or (b) `x` is not equal to 3. No matter what value `x` has, one of the statements must be `True`, since, if `x` is 2, it cannot also be equal to 3, and vice versa.

To fix this problem, you need to refine your understanding of the English expression so that it states the condition more precisely. That is, you want the condition to be `True` whenever "it is not the case that either `x` is 2 or `x` is 3." You could translate this expression directly to Python by writing

```
not (x == 2 or x == 3)
```

but the resulting expression would be a bit ungainly. The question you really want to ask is whether *both* of the following conditions are `True`:

- `x` is not equal to 2, *and*
- `x` is not equal to 3.

If you think about the question in this form, you can write the test as

```
x != 2 and x != 3
```

This simplification is a specific illustration of the following more general relationship from mathematical logic:

`not (p or q)` *is equivalent to* `not p and not q`

for any logical expressions p and q. This transformation rule and its symmetric counterpart

```
not (p and q)      is equivalent to      not p or not q
```

Augustus De Morgan

are called ***De Morgan's laws*** after the British mathematician Augustus De Morgan. Forgetting to apply these rules and relying instead on the English style of logic can lead to programming errors that are difficult to find.

Short-circuit evaluation

Python interprets the `and` and `or` operators in a way that differs from the interpretation used in many other programming languages. In the programming language Pascal, for example, evaluating these operators requires evaluating both halves of the condition, even when the result can be determined partway through the process.

The designers of Python (or, more accurately, the designers of earlier languages that influenced Python's design) took a different approach that is usually more convenient for programmers. Whenever Python evaluates an expression of the form

exp_1 `and` exp_2

or

exp_1 `or` exp_2

the individual subexpressions are always evaluated from left to right, and evaluation ends as soon as the answer can be determined. For example, if exp_1 is `False` in the expression involving `and`, there is no need to evaluate exp_2 since the final answer will always be `False`. Similarly, in the example using `or`, there is no need to evaluate the second operand if the first operand is `True`. This style of evaluation, which stops as soon as the answer is known, is called ***short-circuit evaluation.***

A primary advantage of short-circuit evaluation is that it allows one condition to control the execution of a second one. In many situations, the second part of a compound condition is meaningful only if the first part comes out a certain way. For example, suppose you want to express the combined condition that (1) the value of the integer `x` is nonzero and (2) `x` divides evenly into `y`. You can express this conditional test in Python as

```
(x != 0) and (y % x == 0)
```

because the expression `y % x` is evaluated only if x is nonzero. The corresponding expression in Pascal fails to generate the desired result, because both parts of the Pascal condition will always be evaluated. Thus, if `x` is 0, a Pascal program containing this expression will end up dividing by 0 even though it appears to have a

conditional test to check for that case. Conditions that protect against evaluation errors in subsequent parts of a compound condition, such as the conditional test

```
(x != 0)
```

in the preceding example, are called ***guards.***

Avoiding fuzzy standards of truth

In the programs included in this book, every conditional test produces a Boolean value, which means that it will always be either `True` or `False`. Unfortunately, the Python language is rather less disciplined on this point. Python defines the following values (a couple of which you have not yet seen) to be ***falsy,*** presumably to imply that they are like the legitimate Boolean value `False`:

> `False`, `0`, `None`, `math.nan`, and any sequence of length 0 including `""`

Conversely, Python defines any other value to be ***truthy.*** In any conditional context, any "falsy" value is treated as if it were the value `False`; any "truthy" value is treated as if it were the value `True`.

The complexity of this situation is increased further by the fact that the `and` and `or` operators are implemented so that they allow operands to be of any type. When Python evaluates a sequence of expressions joined together by the `and` operator, it returns the first falsy value in the sequence, so that the expression

```
0 and True
```

returns the integer 0, because 0 is falsy and thus determines the value of the entire expression. Conversely, a sequence of expressions joined together by the `or` operator returns the first truthy value in the sequence.

Overly clever programmers will find uses for Python's rather convoluted interpretation of Boolean values. If, however, you want to write programs that are easy to read and maintain, you should avoid relying on these fuzzy definitions of truth and falsity and make sure—as this book does—that every test produces a legitimate Boolean value. In his book, *JavaScript: The Good Parts,* Douglas Crockford lists the "surprisingly large number of falsy values" in his appendix on the "awful parts" of JavaScript. That feature is no less awful in Python. But you might also take the following advice from a somewhat older source:

> Let what you say be simply "Yes" or "No"; anything more than this comes from evil.
>
> —Matthew 5:37, *The New English Bible*

Repeating a string

In Python, you can use the * operator to specify a string composed by concatenating multiple copies of a shorter string. For example, the expression

```
"ab" * 3
```

returns the six-character string `"ababab"`. Python's use of the * operator seems appropriate, not only because it suggests multiplicity but also because it corresponds to the mathematical definition of multiplication as repeated addition, as follows:

`"ab" * 3` *is the same as* `"ab" + "ab" + "ab"`

As it does in arithmetic expressions, the * operator takes precedence over the + operator, so that the expression

```
"Rose" + " is a rose" * 3 + "."
```

performs the * operator first and therefore returns the string

```
"Rose is a rose is a rose is a rose."
```

This sentence appears in Gertrude Stein's poem "Sacred Emily" from 1913.

Selecting an individual character

You can select an individual character from a Python string by enclosing its index in square brackets. Character positions in a string are numbered starting from 0. For example, the characters in the constant `ALPHABET` defined on the previous page are numbered as in the following diagram:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

The expression `ALPHABET[10]`, for example, is the one-character string `"K"`.

It is often useful, however, to specify a character by indicating how far that character is from the end of the string. Python allows a string index to be negative, in which case the position is determined by counting backwards from the end. The characters in `ALPHABET` can therefore also be numbered like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
−26	−25	−24	−23	−22	−21	−20	−19	−18	−17	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1

Using this numbering scheme, the expression `ALPHABET[-3]` selects the third character from the end, or `"X"`.

Negative index numbers are never necessary but in some cases turn out to be convenient. In particular, it is more concise to select the last character in the string `s` by writing `s[-1]` than the longer and less evocative `s[len(s) - 1]`.

Slicing

While concatenation makes longer strings from shorter pieces, you often need to do the reverse: separate a string into the shorter pieces it contains. A string that is part of a longer string is called a ***substring.*** Python makes it easy and convenient to extract substrings by extending the square-bracket notation for character selection so that you can specify not only a single index position but also a range of index positions marking the boundaries of a substring. In Python, using square brackets to select a range of characters is called ***slicing.***

In its simplest form, a slice in Python is written using two indices separated by a colon inside the square brackets, like this:

str[*start*:*limit*]

As with the `range` function defined in Chapter 2, the index expressions inside the square brackets specify a half-open interval in the sense that the index range includes *start* but stops just before *limit.* Thus, the expression

```
ALPHABET[1:4]
```

returns the three-character substring `"BCD"`, which starts at index position 1 and ends just before index position 4. Similarly, the expression

```
ALPHABET[1:-1]
```

returns the 24-character substring `"BCDEFGHIJKLMNOPQRSTUVWXY"`, which stops just short of the index position indicated by −1, which uses negative indexing to specify the last character in the string.

Python allows you to leave out the index expressions on either side of the colon. If the first index is missing, it is assumed to be the beginning of the string, so that

```
ALPHABET[:5]
```

selects the substring `"ABCDE"` consisting of the first five characters in `ALPHABET`. If the second expression is missing, it is taken to be the length of the string. Thus,

```
ALPHABET[13:]
```

selects the substring `"NOPQRSTUVWXYZ"`, which contains the characters from index position 13 up to the end of the string.

The square-bracket syntax also accepts an optional third component, as follows:

str[*start*:*limit*:*stride*]

When the *stride* component appears, it indicates the distance between characters chosen for inclusion in the substring. For example, the expression

```
ALPHABET[9:20:5]
```

selects characters from `ALPHABET` starting at position 9, ending before position 20, and moving ahead five characters on each stride. This expression therefore selects the characters in index positions 9, 14, and 19 to produce the string `"JOT"`. The expression

```
ALPHABET[::2]
```

uses the default values for *start* and *limit* but uses a *stride* of 2 to select every other character from `ALPHABET`, which produces the string `"ACEGIKMOQSUWY"`.

As with the built-in `range` function, the *stride* component can be negative, in which case the characters are selected by counting backwards through the string. When the *stride* value is negative, the *start* component defaults to the last character in the string, and the *limit* component defaults to the beginning of the string. For example, the expression

```
ALPHABET[::-1]
```

returns the characters in `ALPHABET`, chosen from back to front to produce the 26-character string `"ZYXWVUTSRQPONMLKJIHGFEDCBA"`.

Programmers entranced by Python's particularly succinct style of expression are often tempted to use this form of slicing to reverse a string. Doing so, however, makes the resulting program difficult to follow for programmers unfamiliar with this Python-specific idiom. One way to restore the desired readability is to embed this operation in a function whose name makes the effect of the operation clear, like this:

```
def reverse_string(s):
    return s[::-1]
```

Although some readers may be mystified as to how this implementation achieves the desired effect, those readers can use the name of the function to understand the program on a more holistic level.

7.3 Common string patterns

Although section 7.2 gives you a sense of what string operators Python offers, the discussion gives you little guidance as to how you can use these operators most effectively. When you are learning to program, it is often easier to ignore as many details as possible and instead write your programs by relying on code patterns that implement common operations. The two most important string patterns are iterating through the characters in a string and growing a string by concatenation. The sections that follow describe these patterns.

Iterating through the characters in a string

When you work with strings, one of the most important patterns involves iterating through the characters in a string. In its simplest form, which you have already seen in Chapter 1, iterating through the characters in a string requires the following code:

```
for ch in s:
    . . . body of loop that uses the character ch . . .
```

On each loop cycle, the variable `ch` is bound to a one-character string chosen from successive index positions of the string `s`. The body of the loop then uses that character to perform some computation. You can, for example, count the number of spaces in a string using the following function:

```
def count_spaces(s):
    ns = 0
    for ch in s:
        if ch == " ":
            ns += 1
    return ns
```

Growing a string through concatenation

The other string pattern that is important to memorize involves creating a new string one character at a time. The details of the loop depend on the application, but the general pattern for creating a string by concatenation looks like this:

```
result = ""
for whatever loop header line fits the application:
    result += the next piece of the result
```

For example, the `n_copies` function returns a string consisting of `n` copies of `s`, achieving the effect of the expression `s * n` without using the `*` operator:

```
def n_copies(n, s):
    result = ""
    for i in range(n):
        result += s
    return result
```

Combining the iteration and concatenation patterns

Many string-processing functions use the iteration and concatenation patterns together. For example, the following function returns a copy of the string `s` with all spaces removed:

```
def remove_spaces(s):
    result = ""
    for ch in s:
        if ch != " ":
            result += ch
    return result
```

As a second example, the following function offers another strategy—arguably more readable but certainly less efficient—for implementing the `reverse_string` function first defined on page 228:

```
def reverse_string(s):
    result = ""
    for ch in s:
        result = ch + result
    return result
```

This implementation builds up the reversed string by concatenating each character onto the front of the existing result. For example, calling `reverse("stressed")` assigns the following values to `result` as it goes through the `for` loop:

```
""
"s"
"ts"
"rts"
"erts"
"serts"
"sserts"
"esserts"
"desserts"
```

breaks, Python uses that spacing to define the hierarchical structure of a program. In Python, a line break ordinarily signals the end of a statement. Because the `return` statement includes a Boolean expression that doesn't fit comfortably on a single line, you need to find some way to let the expression extend across more than one line. This example solves the problem by preceding the line break in the middle of the expression by a backward slash (\), which causes Python to treat the following line as part of this one. Python also ignores any line breaks that occur within parentheses, square brackets, or curly braces, but that rule doesn't apply in this example as it appears. You will have many opportunities to see applications of this second rule, which removes the need for the line-continuation character.

2.2 The `if` statement

The simplest way to express conditional execution in Python is by using the `if` statement, which comes in three forms, as shown in the syntax boxes on the left. The first form of the `if` statement is useful when you want to perform an action only under certain conditions. The second is appropriate when you need to choose between two alternative courses of action. The third, which can contain any number of `elif` clauses, makes sense if you need to choose among several different courses of action.

```
if condition:
    statements
```

```
if condition:
    statements
else:
    statements
```

```
if condition1:
    statements
elif condition2:
    statements
elif condition3:
    statements
else:
    statements
```

The *condition* component of these templates is a Boolean expression, as defined in the preceding section. This Boolean expression can be a simple comparison, a logical expression involving the `and`, `or`, and `not` operators, or a call to a predicate function. For example, if you want to test whether the number stored in `year` corresponds to a leap year, you can use the following `if` statement, which calls the `is_leap_year` function defined on the preceding page:

```
if is_leap_year(year):
```

In the first form of the `if` statement, Python executes the block of statements only if the conditional test evaluates to `True`. If the conditional test is `False`, Python skips the body of the `if` statement entirely. In the second form, Python executes the first block of statements if the condition is `True` and the second if the condition is `False`. In the third form, Python evaluates each of the conditions in turn and executes the statements associated with the first condition that evaluates to `True`. If none of the conditions apply, Python executes the statements associated with the `else` keyword.

You can use the `if` statement to implement your own versions of Python's built-in functions. For example, you can implement `abs`—at least for integers and floating-point numbers—as follows:

```
def abs(x):
    if x < 0:
        return -x
    else:
        return x
```

Similarly, you can implement `max` for two arguments like this:

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

As a third example, you can use the following definition to implement `sign(x)`, which returns –1, 0, or 1, depending on the sign of `x`:

```
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    else:
        return 1
```

Choosing which form of the `if` statement to use requires you to think about the structure of the problem. You use the simple `if` statement when the problem requires code to be executed only if a particular condition applies. You use the `if-else` form for situations in which the program must choose between two independent sets of actions. You can often make this decision based on how you would describe the problem in English. If that description contains the word *otherwise* or some similar expression, there is a good chance that you'll need the `if-else` form. If the English description conveys no such notion, the simple form of the `if` statement is probably sufficient. Finally, you use the `if-elif-else` form to express a choice among several different options.

2.3 The `while` statement

The simplest iterative construct is the `while` statement, which repeatedly executes a simple statement or block until the conditional expression becomes `False`. The template for the `while` statement appears in the syntax box on the right. The entire statement, including both the `while` control line itself and the statements enclosed within the body, constitutes a ***while loop.*** When the program executes a `while` statement, it first evaluates the conditional expression to see if it is `True` or `False`. If the condition is `False`, the loop ***terminates*** and the program continues with the next

```
while condition:
    statements
```

FIGURE 2-2 Program to add a list of integers

```
# File: AddIntegerList.py

"""This program adds a list of integers entered by the user."""

def add_integer_list():
    print("This program adds a list of integers.")
    print("Enter a blank line to stop.")
    total = 0
    finished = False
    while not finished:
        line = input(" ? ")
        if line == "":
            finished = True
        else:
            total += int(line)
    print(f"The sum is {total}")

# Startup code

if __name__ == "__main__":
    add_integer_list()
```

2.4 The `for` statement

```
for var in iterable:
    statements
```

The most important control statement in Python is the `for` statement, which is typically used in situations in which you know how many cycles the loop will run before it begins. The general form of the `for` statement appears in the syntax box on the left. When Python encounters a loop of this sort, it executes the statement in the body with the variable indicated by the placeholder *var* set to each element in the collection of values specified by *iterable*. Python uses the term ***iterable*** to specify a data value that supports ***iteration,*** which is the formal term computer scientists use for the process of looping through a collection one value at a time

Iterating over a range of integers

One of the most common uses of the `for` statement is to cycle through a range of integers. In this case, the *iterable* placeholder in the `for` loop paradigm consists of a call to the built-in function `range`, which returns an iterable value whose elements are the desired integers. The `for` loop then executes one cycle for each value.

The `range` function offers several different patterns that give you considerable control over the order in which the `for` loop processes the elements. These patterns are determined by the number of arguments, as follows:

- If you call `range` with one argument, as in `range(`*n*`)`, the result generates a sequence of n values beginning with 0 and extending up to the value $n-1$.
- If you call `range` with two arguments in the form `range(`*start*`,` *limit*`)`, the result generates a sequence beginning with *start* and continuing up to but not including the value of *limit.*
- If you call `range` with three arguments in the form `range(`*start*`,` *limit*`,` *step*`)`, the result generates a sequence beginning with *start* and then counts in increments of *step* up to but not including *limit.* If the value of *step* is negative, the sequence begins with *start* and then counts backwards down to but not including *limit.*

Figure 2-3 illustrates each of these argument patterns in the context of a `for` loop that displays each of the values in the range.

The variable that appears in the `for` loop pattern is called an ***index variable.*** In each of the examples in Figure 2-3, the index variable is named `i`. Although using single-letter names can sometimes make programs more difficult to understand, using `i` as an index variable follows a well-established programming convention. Just as the single-letter variables names `x` and `y` are perfectly appropriate if they refer to coordinate values, programmers immediately recognize the variable name `i` as a loop index that cycles through a sequence of integers.

FIGURE 2-3 Examples of `for` loops using the range function

IDLE
```
>>> for i in range(5):
        print(i)

0
1
2
3
4
>>>
```

IDLE
```
>>> for i in range(-2, 3):
        print(i)

-2
-1
0
1
2
>>>
```

IDLE
```
>>> for i in range(1, 10, 2):
        print(i)

1
3
5
7
9
>>>
```

IDLE
```
>>> for i in range(5, 0, -1):
        print(i)

5
4
3
2
1
>>>
```

The last example in Figure 2-3 shows that you can use the `range` function to count backwards. You could use this feature to write a function that simulates a countdown from the early days of the space program:

```
def countdown(n):
    for t in range(n, -1, -1):
        print(t)
```

Calling `countdown(10)` produces the following output on the console:

The `countdown` function also demonstrates that any variable can be used as an index variable. In this case, the variable is called `t`, presumably because that is traditional for a rocket countdown, as in the phrase “T minus 10 seconds and counting.”

The factorial function

The ***factorial*** of a nonnegative integer *n,* which is traditionally written as *n*! in mathematics, is defined to be the product of the integers between 1 and `n`. The first ten factorials are shown in the following table:

0!	=	1		(by definition)
1!	=	1	=	1
2!	=	2	=	1 × 2
3!	=	6	=	1 × 2 × 3
4!	=	24	=	1 × 2 × 3 × 4
5!	=	120	=	1 × 2 × 3 × 4 × 5
6!	=	720	=	1 × 2 × 3 × 4 × 5 × 6
7!	=	5040	=	1 × 2 × 3 × 4 × 5 × 6 × 7
8!	=	40320	=	1 × 2 × 3 × 4 × 5 × 6 × 7 × 8
9!	=	362880	=	1 × 2 × 3 × 4 × 5 × 6 × 7 × 8 × 9

Factorials have extensive applications in statistics, combinatorial mathematics, and computer science. A function to compute factorials is therefore a useful tool for solving problems in those domains. You can implement a function `fact(n)` by initializing a variable called `result` to 1 and then multiplying `result` by each of the integers between 1 and `n`, inclusive. The resulting code looks like this:

```
def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

Note that the `for` loop specifies the upper limit of the range as `n + 1` to ensure that the value `n` is included in the product.

The `FactorialTable.py` program in Figure 2-4 on the next page displays a list of the factorials starting at `LOWER_LIMIT` and extending up to but not including `UPPER_LIMIT`, as illustrated by the following sample run:

The code for this program is divided into multiple modules. The main program is stored in the `FactorialTable.py` module shown at the top of Figure 2-4. The code in `FactorialTable.py` produces the table shown in the sample run but relies on a separate `factorial.py` model for the `fact` function and an `alignment.py` module (which you will have a chance to write in exercise 9) for the `align_right` function defined on page 47.

The module that together define the `FactorialTable` application play slightly different roles. The `FactorialTable.py` module defines a Python ***program*** that delivers output to the user through the use of `print` statements. The `factorial.py` and the as-yet-unwritten `alignment.py` modules each represent a Python ***library*** that performs a service for the main program without communicating directly with the user. The functions in the library modules communicate with their callers by taking arguments as input and returning results.

The names of these modules reflect a convention that applies throughout this text. Modules that are intended to be run as programs use camel-case names beginning with an uppercase letter, as illustrated by the module name `FactorialTable.py`. Modules intended to be used as libraries, such as `factorial.py` and `alignment.py` in this chapter or like the `temperature.py` module from Chapter 1 have names written entirely in lowercase letters.

The syntax of a function definition

```
def name(parameters):
    statements
```

```
return value
```

A typical function definition has the form shown in the syntax box on the right. The *name* component of this pattern indicates the function name, *parameters* is the list of parameter names that receive the values of the arguments, and *statements* represents the body of the function. Functions that return a value to the caller must contain at least one `return` statement that specifies the value of the function, as illustrated in the second syntax box.

These syntactic patterns are illustrated in the definition of the `max` function from Chapter 2, which looks like this:

```
def max(x, y):
   if x > y:
      return x
   else:
      return y
```

This function has the name `max` and takes two parameters, `x` and `y`. The statements in the body decide which of these two values is larger and then return that value.

Functions, however, are often called simply for their effect and need not return a value. For example, the Python functions that implement complete programs don't include a `return` statement. Some languages distinguish a function that returns a value from one that doesn't by calling the latter a *procedure.* Python uses the term *function* for both types. This terminology is technically accurate because Python functions always return a value, which is the Python constant `None` if no `return` statement appears.

Parameter passing

In the function calls you have seen so far, the arguments supplied by the caller are copied to the parameter variables in the order in which they appear. The first argument is assigned to the first parameter variable, the second argument to the second parameter variable, and so on. Parameters passed by their order in the argument list are called ***positional parameters.***

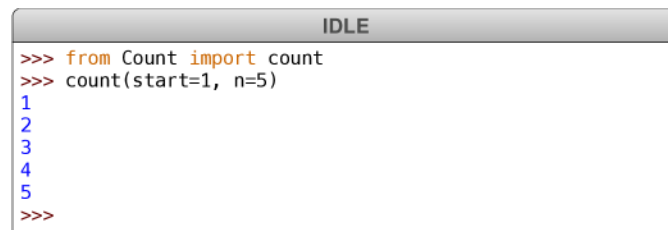
When you use positional parameters, the variable names in the caller and the called function are completely irrelevant to the process by which parameter values are assigned. There may well be a variable named `x` in both the calling function and in the parameter list for the function being called. That reuse of the same name, however, is merely a coincidence. Local variable names and parameter names are visible only inside the function in which their declarations appear.

Python allows a function to specify a value for a parameter that the caller fails to supply. Such parameters are called ***default parameters.*** Default parameters appear in the function header line with an equal sign and a default value. For example, the following function displays `n` consecutive integers, beginning with the value `start` if two arguments are supplied and with the value 1 if the second argument is missing:

```
def count(n, start=1):
    for i in range(n):
        print(start + i)
```

The following IDLE session illustrates the operation of `count`, both when it is given a second argument and when it is not:

Python also allows callers to pass arguments by including the parameter name and an equal sign in the function call. For example, if you cannot remember the order of parameters for the `count` function, you can write the arguments in either order by including the parameter names, as follows:



Parameters identified by name are called ***keyword parameters,*** even though the names are not in any way related to Python keywords like `def` or `while`.

Default and keyword parameters are useful in designing library functions that are easy to use. The section entitled “Designing your own libraries” later in this chapter includes several examples of each of these styles.

5.2 The mechanics of function calls

Although you can certainly get by with an intuitive understanding of how the function-calling process works, it helps to understand precisely what happens when one function calls another in Python. The sections that follow describe the process in detail and then walk you through a simple example.

The steps in calling a function

Whenever a function call occurs, Python executes the following operations:

1. The calling function computes values for each argument using the bindings of local variables in its own context. Because the arguments are expressions, this computation can involve operators and other functions; the calling function evaluates these expressions before execution of the new function begins.
2. The system creates new space for all the local variables required by the new function, including the variables in the parameter list. These variables are allocated together in a block, which is called a ***stack frame.***
3. Each positional argument is copied into the corresponding parameter variable.
4. All keyword arguments are copied to the parameter with the same name.
5. For parameters that include default values, Python assigns those values to any arguments that are still unspecified. If any parameters are still unassigned after this step, Python reports an error.
6. The statements in the function body are executed until the program encounters a `return` statement or there are no more statements to execute.
7. The value of the `return` expression, if any, is evaluated and returned as the value of the function.
8. The stack frame created for this function call is discarded. In the process, all local variables disappear.
9. The calling program continues, with the returned value substituted in place of the call. The point to which the function returns is called the ***return address.***

Although this process may seem to make at least some sense, you probably need to work through an example or two before you understand it fully. Reading through the example in the next section will give you some insight into the process, but it will be even more helpful to take one of your own programs and walk through it at the same level of detail. And while you can trace through a program on paper or a whiteboard, it may be best to get yourself a supply of 3×5 index cards and then use a card to represent each stack frame. The advantage of the index-card model is that you can create a stack of index cards that closely models the operation of the computer. Calling a function adds a card; returning from the function removes it.

The combinations function

The function-calling process is most easily illustrated in the context of a specific example. Suppose that you have a collection of six coins, which in the United States might be a penny, a nickel, a dime, a quarter, a half-dollar, and a dollar. Given those six coins, how many ways are there to choose two of them? As you can see from the full enumeration of the possibilities in Figure 5-1, the answer is 15. However, as a computer scientist, you should immediately think about the more general question: given a set containing n distinct elements, how many ways can you choose a subset with k elements? The answer to that question is computed by the ***combinations function*** $C(n, k)$, which is defined as

$$C(n, k) = \frac{n!}{k! \times (n-k)!}$$

where the exclamation point indicates the factorial function, which you saw in Chapter 2. The code to compute the combinations function in Python appears in Figure 5-2.

FIGURE 5-1 **Illustration of the combinations function**

If you start with six coins

there are 15 ways to choose two coins:

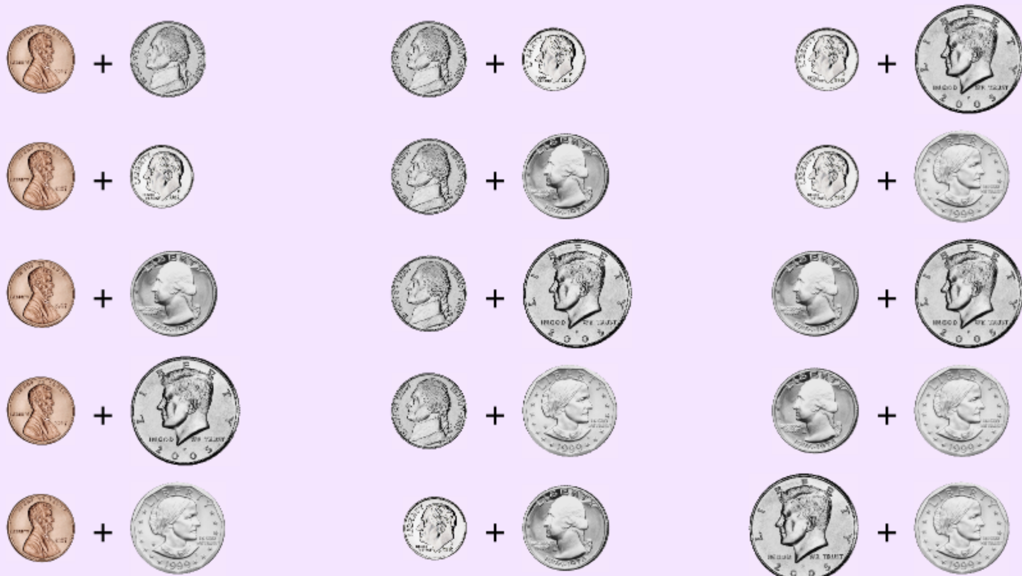


FIGURE 5-2 Python implementation of the mathemematical combinations function $C(n, k)$

```
# File: combinations.py

"""
This module exports an implementation of the mathematical combinations
function C(n, k), which is the number of ways of selecting k objects
from a set of n distinct objects.
"""

def combinations(n, k):
    """
    Returns the mathematical combinations function C(n, k), which is
    the number of ways one can choose k elements from a set of size n.
    """
    return fact(n) // (fact(k) * fact(n - k))

def fact(n):
    """
    Returns the factorial of n, which is the product of all the
    integers between 1 and n, inclusive.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

As you can see from Figure 5-2, the `combinations.py` file contains two functions. The `combinations` function computes the value of $C(n, k)$, and the now-familiar `fact` function computes factorials. An IDLE session just before making the call to `combinations(6, 2)` might look like this:

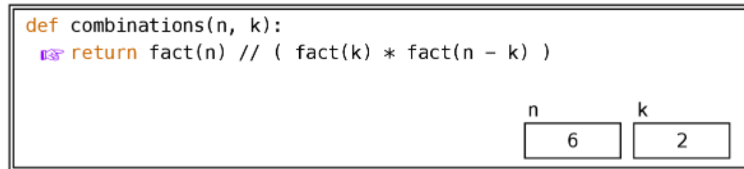
Tracing the combinations function

While the `combinations` function is interesting in its own right, the purpose of the current example is to illustrate the steps involved in calling functions. When the user enters a function call in the IDLE window, the Python interpreter invokes the standard steps in the function-calling process.

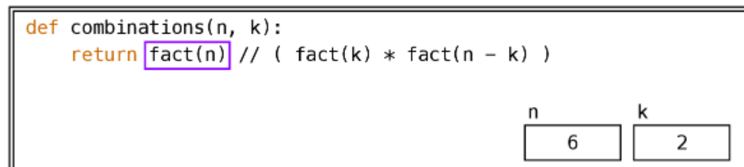
As always, the first step is to evaluate the arguments in the current context. In this example, the arguments are the numbers 6 and 2, so the evaluation process simply keeps track of these two values.

The second step is to create a frame for the `combinations` function that contains space for the variables that are stored as part of that frame, which are the parameters and any variables that appear in declarations within the function. The `combinations` function has two positional parameters and no local variables, so the frame only requires enough space for the parameter variables `n` and `k`. After the Python interpreter creates the frame, it copies the argument values into these variables in order. Thus, the parameter variable `n` is initialized to 6, and the parameter variable `k` is initialized to 2.

In the diagrams in this book, each stack frame appears as a rectangle surrounded by a double line. Each stack-frame diagram shows the code for the function along with a pointing-hand icon that makes it easy to keep track of the current execution point. The frame also contains labeled boxes for each of the local variables. The stack frame for the `combinations` function therefore looks like this after the parameters have been initialized but before execution of the function begins:



To compute the value of the `combinations` function, the program must make three calls to the function `fact`. In Python, function calls are evaluated from left to right, so the first call is the one to `fact(n)`, as follows:



To evaluate this function, the system must create yet another stack frame, this time for the function `fact` with an argument value of 6. The frame for `fact` has both parameters and local variables. The parameter `n` is initialized to the value of the calling argument and therefore has the value 6. The two local variables, `i` and `result`, have not yet been initialized, which is indicated in stack diagrams using an empty box. The new frame for `fact` gets stacked on top of the old one, which allows the Python interpreter to remember the values in the earlier stack frame, even though they are not currently visible. The situation after creating the new frame and initializing the parameters looks like this:

```
def combinations(n, k):
  def fact(n):
  ☞ result = 1
    for i in range(1, n + 1):
        result *= i                 n        result      i
    return result                  [ 6 ]    [     ]    [     ]
```

The system then executes the statements in the function `fact`. In this instance, the body of the `for` loop is executed six times. On each cycle, the value of `result` is multiplied by the loop index `i`, which means that it will eventually hold the value 720 ($1 \times 2 \times 3 \times 4 \times 5 \times 6$ or 6!). When the program reaches the `return` statement, the stack frame looks like this:

```
def combinations(n, k):
  def fact(n):
    result = 1
    for i in range(1, n + 1):
        result *= i                 n        result      i
  ☞ return result                  [ 6 ]    [ 720 ]    [ 7 ]
```

Returning from a function involves copying the value of the `return` expression (in this case the local variable `result`), to the point at which the call occurred. The frame for `fact` is then discarded, which leads to the following configuration:

```
def combinations(n, k):
    return [fact(n)] // ( fact(k) * fact(n - k) )
             ↑
             └── 720                           n        k
                                             [ 6 ]    [ 2 ]
```

The next step in the process is to make a second call to `fact`, this time with the argument `k`. In the calling frame, `k` has the value 2. That value is then used to initialize the parameter `n` in the new stack frame, as follows:

```
def combinations(n, k):
  def fact(n):
  ☞ result = 1
    for i in range(1, n + 1):
        result *= i                 n        result      i
    return result                  [ 2 ]    [     ]    [     ]
```

The computation of `fact(2)` is easier to perform in one's head than the earlier call to `fact(6)`. This time around, the value of `result` will be 2, which is then returned to the calling frame, like this:

```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
              720          2
                                          n     k
                                          6     2
```

The code for `combinations` makes one more call to `fact`, this time with the argument `n - k`. Evaluating this call therefore creates a new stack frame with `n` equal to 4:

```
def combinations(n, k):
  def fact(n):
      result = 1
      for i in range(1, n + 1):
          result *= i
      return result
                                  n     result   i
                                  4
```

The value of `fact(4)` is $1 \times 2 \times 3 \times 4$, or 24. When this call returns, the system is able to fill in the last of the missing values in the calculation, as follows:

```
def combinations(n, k):
    return fact(n) // ( fact(k) * fact(n - k) )
              720          2            24
                                          n     k
                                          6     2
```

The computer then divides 720 by the product of 2 and 24 to get the answer 15. This value is returned to the Python interpreter running in the IDLE console window. The interpreter prints that value on the console, like this:

```
IDLE
>>> from combinations import combinations
>>> combinations(6, 2)
15
>>>
```

5.3 Libraries and interfaces

Writing a program to solve a large or difficult problem inevitably forces you to manage at least some amount of complexity. There are algorithms to design, special cases to consider, user requirements to meet, and innumerable details to get right. To make programming manageable, you must reduce the complexity of the programming process as much as possible. Functions reduce some of the complexity; libraries offer