# Python Primer

## Python Overview

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communications is using a high-level computer language, such as Python. The Python programming language was originally developed by Guido van Rossum in the early 1990s, and has since become a prominently used language in industry and education. The second major version of the language, python 2, was released in 2000, and the third major version, Python 3 released in 2008. We note that there are significant incompatibilities between Python 2 and Python 3. This book is based on Python 3 (more specifically, Python 3.1 or later). The latest version of the language is freely available at [www.python.org](http://www.python.org) , along with documentation and tutorials.

In this chapter, we provide an overview of the Python programming language, and we continue this discussion in the next chapter, focusing on object-oriented principles. We assume that readers of this book have prior programming experience, although not necessarily using Python. The book does not provide a complete description of the Python language (there are numerous language references for the purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

### The Python Interpreter

Python is formally an interpreted language. Commands are executed through a piece of software known as Python interpreter. The interpreter receives a command, evaluates that command, and reports the result of the command. While the interpreter can be used interactively (especially when debugging), a programmer typically defines a series of commands in advance and saves those commands in a plain text file known as source code or a script. For Python, source code is conventionally stored in a file named with the .py suffix (e.g., demo.py).

On most operating systems, the Python interpreter can be started by typing python from the command line. By default, the interpreter starts in interactive mode with a clean workspace. Commands from a predefined script saved in a file (e.g., demo.py) are executed by invoking the interpreter with the filename as an argument (e.g., python demo.py), or using an additional -i flag in order to execute a script and then enter interactive mode (e.g., python -i demo.py).

Many integrated development environments (IDEs) provide richer software development platforms for Python, including one named IDLE that is included with the standard Python distribution. IDLE provides an embedded text-editor with support for displaying and editing Python code, and basic debugger, allowing step-by-step execution of a program while examining key variable values.

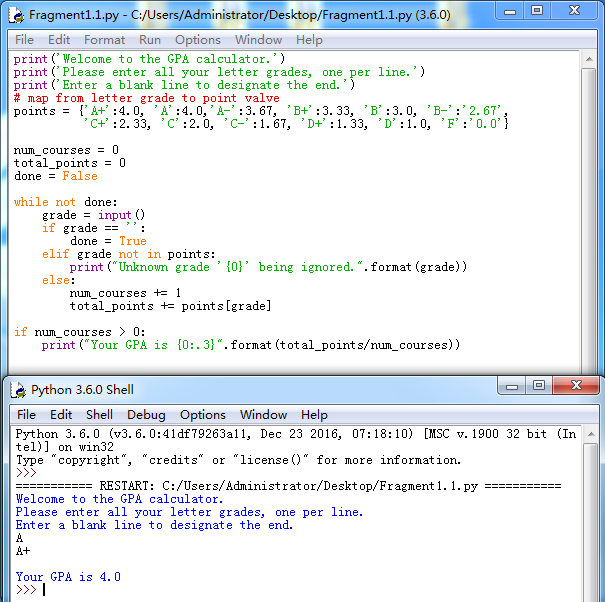
### Preview of a Python Program

As a simple introduction, Code Fragment 1‑1 presents a Python program that computes the grade-point average (GPA) for a student based on letter grades that are entered by a user. Many of the techniques demonstrated in this example will be discussed in the remainder of this chapter. At this point, we draw attention to a few high-level issues, for readers who are new to Python as a programming language.

Python's syntax relies heavily on the use of whitespace. Individual statements are typically concluded with a newline character, although a command can extend to another line, either with a concluding backslash character (\), or if an opening delimiter has not yet been closed, such as the { character in defining value\_map.

Whitespace is also key in delimiting the bodies of control structures in Python. Specifically, a block of code is indented to designate it as the body of a control structure, and nested control structures are increasing amounts of indentation. In Code Fragment 1‑1, the body of the while loop consists of the subsequent 8 lines, including a nested conditional structure.

Comments are annotations provided for human readers, yet ignored by the Python interpreter. The primary syntax for comments in Python is based on use of the # character, which designates the remainder of the line as a comment.



Fragment 1‑1 executed in python IDLE

## Object in Python

Python is an object-oriented language and classed form the basis for all data types. In this section, we describe key aspects of Python's object model, and we introduce Python's built-in classes, such as the int class for integers, the float class for floating-point values, and the str class for character strings. A more thorough presentation of object-orientation is the focus of Chapter 2.

### Identifiers, Objects, and the Assignment Statement

The most important of all Python commands is an assignment, such as

temperature = 98.6

This command establishes temperature as an identifier (also known as a name), and then associates it with the object expressed on the right-hand side of the equal sign, in this case a floating-point object with value 98.6. We portray the outcome of this assignment in Figure 1‑1

98.6

float

temperature

Figure 1‑1 The identifier temperature references an instance of the float class having value 98.6

#### Identifiers

Identifiers in Python are case-sensitive, so temperature and Temperature are distinct names. Identifiers can be composed of almost any combination of letters, numerals, and underscore characters (or more general Unicode characters). The primary restrictions are that an identifier cannot begin with a numeral (thus 9lives is a an illegal name), and that there are 33 specially reserved words that cannot be used as identifiers, as shown in Table 1‑1.

Table 1‑1 A listing of the reserved words in Python. These names cannot be used as identifiers

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Reserved Words** | | | | | | | |
| False | as | continue | else | from | In | not | return |
| yield | None | assert | def | except | global | is | or |
| try | True | break | del | finally | If | lambda | pass |
| while | and | class | elif | for | import | nonlocal | raise |
| with |  |  |  |  |  |  |  |

For readers familiar with other programming languages, the semantics of a Python identifier is most similar to reference variable in Java or a pointer variable in C++. Each identifier is implicitly associated with the memory address of the object to with it refers. A Python identifier may be assigned to a special object named None, serving a similar purpose to a null reference in Java or C++.

Unlike Java and C++, Python is a dynamically typed language, as there is no advance declaration associating an identifier with a particular data type. An identifier can associated with any type of object, and it can later be reassigned to another object of the same (or different) type. Although an identifier has no declared type, the object to which it refers has a definite type. In our first example, the characters 98.6 are recognized as a floating-point literal, and thus the identifier temperature is associated with an instance of the float class having that value.

A programmer can establish an alias by assigning a second identifier to an existing object. Continuing with our earlier example, Figure 1‑2 portrays the result of a subsequent assignment, original = temperature.

98.6

float

temperature original

Figure 1‑2 Identifiers temperature and original are aliased for the same object

Once an alias has been established, either name can be used to access the underlying object. If that object supports behaviors that affect its state, changes enacted through one alias will be apparent when using the other alias (because they refer to the same object). However, if one of the names is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias. Continuing with our concrete example, we consider the command:

temperature = temperature + 5.0

The execution of this command begins with the evaluation of the expression on the right-hand side of the = operator. That expression, temperature + 5.0, is evaluated based on the existing binding of the name temperature, and so the result has value 103.6, that is, 98.6 + 5.0. That result is stored as a new floating-point instance, and only then is the name on the left-hand side of the assignment statement, temperature, (re)assigned to the result. The subsequent configuration is diagrammed in Figure 1‑3. Of particular note, this last command had no effect on the value of the existing float instance that identifier original continues to reference.

float

float

103.6

temperature original

98.6

Figure 1‑3 The temperature identifier has been assigned to a new value while original

continues to refer to the previously existing value.

### Creating and Using objects

#### Instantiation

The process of creating a new instance of a class is known as instantiation. In general, the syntax for instantiating an object is to invoke the constructor of a class. For example, if there were a class named Widget, we could create an instance of that class using a syntax such as w = Widget(), assuming that the constructor does not require any parameters. If the constructor does require parameters, we might use a syntax such as Widget(a, b, c) to construct a new instance.

Many of Python's built-in classes (discussed in Section 1.2.3) support what is known as a literal form for designating new instances. For example, the command temperature = 98.6 results in the creation of a new instance of the float class; the term 98.6 in that expression is a literal form. We discuss further cases of Python literals in the coming section.

From a programmer's perspective, yet another way to indirectly create a new instance of a class is to call a function that creates and returns such an instance. For example, Python has a built-in function named sorted (see Section1.5.2) that takes a sequence of comparable elements as a parameter and returns a new instance of the list class containing those elements in sorted order.

#### Calling Methods

Python supports traditional functions (see Section 1.5) that are invoked with a syntax such as sorted(data), in which case data is a parameter sent to the function. Python's classes may also define one or more methods (also known as member functions), which are invoked on a specific instance of a class using the dot (“.”) operator. For example, Python's list class has a method named sort that can be invoked with a syntax such as data.sort(). This particular method rearranges the contents of the list so that they are sorted.

The expression to the left of the dot identifies the object upon which the method is invoked. Often, this will be an identifier, (e.g., data), but we can use the dot operator to invoke a method upon the immediate result of some other operation. For example, if response identifies a string instance (we will discuss strings later in this section), the syntax response.lower().startswith('y') first evaluates the method call, response.lower(), which itself returns a new string instance, and then the startwith('y') method is called on that intermediate string.

When using a method of a class, it is important to understand its behavior. Some methods return information about the state of an object, but do not change that state. These are known as accessors. Other methods, such as the sort method of the list class, do change the state of an object. These methods are known as mutators or update methods.

### Python's Built-In Classes

Table 1‑2 provides a summary of commonly used, built-in classes in Python; we take particular note of which classes are mutable and which are immutable. A class is immutable if each object of that class has a fixed value upon instantiation. Once an instance has been created, its value cannot be changed (although an identifier referencing that object can be reassigned to a different value).

Table 1‑2 Commonly used built-in classes for Python

|  |  |  |
| --- | --- | --- |
| Class | Description | Immutable |
| bool | Boolean value | √ |
| int | Integer (arbitrary magnitude) | √ |
| float | Floating-point number | √ |
| list | Mutable sequence of objects |  |
| tuple | Immutable sequence of objects | √ |
| str | Character string | √ |
| set | Unordered set of distinct objects |  |
| frozenset | Immutable form of set class | √ |
| dict | Associative mapping (aka dictionary) |  |

In this section, we provide an introduction to these classes, discussing their purpose and presenting several means for creating instances of the classes. Literal forms (such as 98.6) exist for most of the built-in classes, and all of the classes support a traditional constructor form that creates instances that are based upon one or more existing values. Operators supported by these classes are described in Section 1.3. More detailed information about these classes can be found in later chapters as follows: lists and tuples (Chapter 5); strings (Chapter 5 and 13, and Appendix A); sets and dictionaries (Chapter 10).

#### The bool Class

The bool class is used to manipulate logical (Boolean) values, and the only two instances of that class are expressed as the literals True and False. The default constructor, bool(), returns False, but there is no reason to use that syntax rather than the more direct literal form. Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo. The interpretation depends upon the type of the parameter. Numbers evaluate to False if zero, and True id nonzero. Sequences and other container types, such as strings and lists, evaluate to False if empty and True if nonempty. An important application of this interpretation is the use of a nonboolean value as a condition in a control structure.

#### The int Class

The int and float classes are primary numeric types in Python. The int class is designed to represent integer values with arbitrary magnitude. Unlike Java and C++, which support different integral types with different precisions (e.g., int, short, long), Python automatically choose the internal representation for an integer based upon the magnitude of its value. Typical literals for integers include 0, 137, and -23. In some contexts, it is convenient to express an integral value using binary, octal, or hexadecimal. That can be done by using a prefix of the number 0 and then a character to describe the base. Example of such literals are respectively 0b1011, 0o52, and 0x7f.

The integer constructor, int(), returns value 0 by default. But this constructor can be used to construct an integer value based upon an existing value of another type. For example, if f represents a floating-point value, the syntax int(f) produces the truncated value of f. For example, both int(3.14) and int(3.99) produce the value 3, while int(-3.9) produces the value -3. The constructor can also be used to parse a string that is presumed to represent an integral value (such as one entered by a user). If s represents a string, then int(s) produces the integral value that string represents. For example, the expression int('137') produces the integer value 137. If an invalid string is given as a parameter, as in int('hello'), a ValueError is raised (see Section 1.7 for discussion of Python's exceptions). By default, the string must use base 10. If conversion from a different base is desired, that base can be indicated as a second, optional, parameter. For example, the expression int('7f',16) evaluates to the integer 127.

>>> int('7f',16)

127

>>> int(3.99)

3

>>> int(-3.9)

-3

>>> int('hello')

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

int('hello')

ValueError: invalid literal for int() with base 10: 'hello'

>>>

#### The float Class

The float class is the sole floating-point type in Python, using a fixed-precision representation. Its precision is more akin to a double in Java or C++, rather than those language's float type. We have already discussed a typical literal form, 98.6. We note that the floating-point equivalent of an integral number can be expressed directly as 2.0. Technically, the trailing zero is optional, so some programmers might use the expression 2. To designete this floating-point literal. One other form of literal for floating-point values uses scientific notation. For example, the literal 6.022e23 represents the mathematical value 6.022×1023.

The constructor form of float() returns 0.0. When given a parameter, the constructor attempts to return the equivalent floating-point value. For example, the call float(2) returns the floating-point value 2.0. If the parameter to the constructor is a string, as with float('3.14'), it attempts to parse that string as a floating-point value, raising a ValueError as an exception.

#### Sequence Types: The list, tuple, and str Classes

The list, tuple, and str classes are sequence types in Python, representing a collection of values in which the order is significant. The list class is the most general, representing a sequence of arbitrary objects (akin to an “array” in other languages). The tuple class is an immutable version of the list class, benefiting from a streamlined internal representation. The str class is specially designed for representing an immutable sequence of text characters. We note that Python does not have a separate class for characters; they are just strings with length one.

#### The list Class

A list instance stores a sequence of objects. A list is a referential structure, as it technically stores a sequence of references to its elements (see Figure 1‑4). Elements of a list may be arbitrary objects (including the None object). Lists are array-based sequences and are zero-indexed, thus a list of length n has elements indexed from 0 to n-1 inclusive. Lists are perhaps the most used container type in Python and they will be extremely central to our study of data structures and algorithms. They have many valuable behaviors, including the ability to dynamically expand and contract their capacities as needed. In this chapter, we will discuss only the most basic properties of lists. We revisit the inner working of all of Python's sequence types as the focus of Chapter 5.

Python uses the characters [ ] as delimiters for a list literal, with [ ] itself being an empty list. As another example, ['red', 'green', 'blue'] is a list containing three string instances. The contents of a list literal need not be expressed as literals; if identifiers a and b have been established, then syntax [a, b] is legitimate.

The list() constructor produces an empty list by default. However, the constructor will accept any parameter that is of an iterable type. We will discuss iteration further in Section 1.8, but examples of iterable types include all of the standard container types (e.g., strings, list, tuples, sets, dictionaries). For example, the syntax list('hello') produces a list of individual characters, ['h', 'e', 'l', 'l', 'o']. Because an existing list is itself iterable, the syntax

backup = list(data)

can be used to construct a new list instance referencing the same contents as the original.

13

29

23

11

19

17

7

3

2

5

primes:

0 1 2 3 4 5 6 7 8 9

Figure 1‑4 Python's internal representation of a list of integers, instantiated as

Prime = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]. The implicit indices of the elements

Are shown below each entry.

#### The tuple Class

The tuple class provides an immutable version of a sequence, and therefore its instances have an internal representation that may be more streamlined than that of a list. While Python uses the [ ] character to delimit a list, parentheses delimit a tuple, with ( ) being an empty tuple. There is one important subtlety. To express a tuple of length as a literal, a comma must be placed after the element, but within the parentheses. For example, (17,) is a one-element tuple. The reason for this requirement is that, without the trailing comma, the expression (17) is viewed as a simple parenthesized numeric expression.

>>> a = (17)

>>> type(a)

<class 'int'>

>>> b = (17,)

>>> type(b)

<class 'tuple'>

>>>

#### The str Class

Python's str class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international set. Strings have a more compact internal representation than the referential lists and tuples, as portrayed in Figure 1‑5.

E

L

P

M

A

S

0 1 2 3 4 5

Figure 1‑5 A Python string, which is an indexed sequence of characters.

String literals can be enclosed in single quotes, as in 'hello', or double quotes, as in "hello". This choice is convenient, especially when using another of the quotation characters as an actual character in the sequence, as in "Don't worry". Alternatively, the quote delimiter can be designated using the backslash as a so-called escape character, as in 'Don\'t worry'. Because the backslash has this purpose, the backslash must itself be escaped to occur as a natural character of the string literal, as in 'C:\\Python\\', for a string that would be displayed as C:\Python\. Other commonly escaped characters are \n for newline and \t for tab. Unicode characters can be included, such as '20\u20AC', for the string 20€.

Python also supports using the delimiter ''' or """ to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as \n). This can greatly improve the readability of long multiple strings in source code. For example, at the beginning of Code Fragment 1‑1, rather than use separate print statements as follows:

print("""Welcome to the GPA calculator.

Please enter all your letter grades, one per line.

Enter a blank line to designate the end""")

#### The set and frozenset Classes

Python's set class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in a set. This is based on a data structure known as a hash table (which will be the primary topic of Chapter 10). However, there are two important restrictions due to the algorithmic underpinnings. The first is that the set does not maintain the elements in any particular order. The second is that only instances of immutable types can be added to a Python set. Therefore, objects such as integers, floating-point numbers, and character strings are eligible to be elements of a set. It is possible to maintain a set of tuples, but not a set of lists or a set of sets, as lists and sets are mutable. The frozenset class is an immutable form of the set type, so it is legal to have a set of frozensets.

Python uses curly braces { and } as delimiters for a set, for example, as { 17 } or {'red', 'green', 'blue'}. The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary (see next paragraph). Instead, the constructor syntax set() produces an empty set. If an iterable parameter is sent to the constructor, then the set of distinct elements is produced. For example, set('hello') peoduces {'l', 'h', 'o', 'e'}.

>>> {'red','green','blue'}

{'blue', 'green', 'red'}

>>> set('hello')

{'l', 'h', 'o', 'e'}

#### The dict Class

Python's dict class represents a dictionary, or mapping, from a set of distinct keys to associated values. For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades). Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.

A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary. A nonempty dictionary is expressed using a comma-separated series of key : value pairs. For example, the dictionary {'ga' : 'Irish', 'de' : 'German'} maps 'ga' to 'Irish' and 'de' to 'German'.

The constructor for the dict class accepts an existing mapping as a parameter, in which case it creates a new dictionary with identical associations as the existing one. Alternatively, the constructor accepts a sequence of key-value pairs as a parameter, as in dict(pairs) with [('ga','Irish'),('de','German')].

>>> pairs = [('ga','Irish'),('de','German')]

>>> dict(pairs)

{'ga': 'Irish', 'de': 'German'}

>>>

## Expressions, Operators, and Precedence

In the previous section, we demonstrated how names can be used to identify existing objects, and how literals and constructors can be used to create instances of built-in classes. Existing values can be combined into large syntactic expressions using a variety of special symbols and keywords known as operators. The semantics of an operator depends upon the type of its operands. For example, when a and b are numbers, the syntax a + b indicates addition, while if a and b are strings, the operator indicates concatenation. In this section, we describe Python's operators in various contexts of the built-in types.

We continue, in Section 1.3.1, by discussing compound expressions, such as a + b \* c, which rely on the evaluation of two or more operations. The order in which the operations of a compound expression are evaluated can affect the overall value of the expression. For this reason, Python defines a specific order of precedence for evaluating operators, and it allows a programmer to override this order by using explicit parentheses to group subexpressions.

#### Logical Operators

Python supports the following keyword operators for Boolean values:

not unary negation

and conditional and

or conditional or

The and and or operators short-circuit, in that they do not evaluate the second operand if the result can be determined based on the value of the first operand. This feature is useful when constructing Boolean expressions in which we first test that a certain conditional holds (such as a reference not being None), and then test a condition that could have otherwise generated an error condition had the prior test not succeeded.

#### Equality Operators

Python supports the following operators to test two notions of equality:

|  |  |
| --- | --- |
| is | same identify |
| is not | different identity |
| == | equivalent |
| != | not equivalent |

The expression a is b evaluate to True, precisely when identifiers a and b are aliases for the same object. The a == b tests a more general notion of equivalence. If identifiers a and b refer to the same object, then a == b should also evaluate to True. Yet a == b also evaluates to True when the identifiers refer to different objects that happen to have values that are deemed equivalent. The precise notion of equivalence depends on the data type. For example, two strings are considered equivalent if they match character for each character. Two sets are equivalent if they have the same contents, irrespective of order. In most programming situations, the equivalence test == and != are the appropriate operators; use of is and is not should be reserved for situations in which it is necessary to detect true aliasing.

#### Comparison Operators

Data types may define a natural order via the following operators:

|  |  |
| --- | --- |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |

These operators have expected behavior for numeric types, and are defined lexicographically, and case-sensitively, for strings. An exception is raised if operands have incomparable types, as with 5 < 'hello'.

#### Arithmetic Operators

Python supports the following arithmetic operators:

|  |  |
| --- | --- |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | True division |
| // | Integer division |
| % | The modulo operator |

The use of addition, subtraction, and multiplication is straightforward, nothing that if both operands have type int, then the result is an int as well; if one or both operands have type float, the results will be a float.

Python takes more care in its treatment of division. We first consider the case in which both operands have type int, for example, the quantity 27 divided by 4. In mathematical notation, 27 ÷ 4 = 6¾ = 6.75. In Python, the / operator designates true division, returning the floating-point result of the computation. Thus, 27 / 4 results in the float value 6.75. Python supports the pair of operators // and % to perform the integral calculations, with expression 27 // 4 evaluating to int value 6 (the mathematical floor of the quotient), and expression 27%4 evaluating to int value 3, the remainder of the integer division. We note that language such as C, C++, and Java do not support the // operator; instead, the / operator returns the truncated quotient when both operands have integral type, and the result of true division when at least one operand has a floating-point type.

Python carefully extends the semantics of // and % to cases where one or both operands are negative. For the sake of notation, let us assume that variables n and m represent respectively the dividend and divisor of a quotient n/m, and that q = n // m and r = n % m. Python guarantees that q \* m + r will return n. We already saw an example of this identity with positive operands, as 6\*4 + 3 = 27. When the divisor m is positive, Python further guarantees that 0 ≤ r < m. As a consequence, we find that -27 // 4 evaluates to -7 and

-27 % 4 evaluates to 1, as (-7) \* 4 + 1 = -27. When the divisor is negative, Python guarantees that m < r ≤ 0. As an example, 27 // -4 is -7 and 27 % -4 is -1, satisfying the identity 27 = (-7)\*(-4)+(-1).

The conventions for the // and % operators are even extended to floating-point operands, with the expression q = n // m being the integral floor of the quotient, and r = n % m being the remainder to ensure that q \* m + r equals n. For example, 8.2 // 3.14 evaluates to 2.0 and 8.2 % 3.14 evaluates to 1.92, as 2.0\*3.14+1.92 = 8.2.

>>> 27/4

6.75

>>> 27//4

6

>>> 27%-4

-1

>>> 8.2//3.14

2.0

>>> 8.2%3.14

1.919999999999999

>>>

#### Bitwise Operators

Python provides the following bitwise operators for integers:

|  |  |
| --- | --- |
| ~ | Bitwise complement (prefix unary operator) |
| & | Bitwise and |
| | | Bitwise or |
| ^ | Bitwise exclusive -or |
| << | Shift bits left, filling in with zeros |
| >> | Shift bits right, filling in with sign bit |

#### Sequence Operators

Each of Python's built-in sequence types (str, tuple, and list) support the following operator syntax:

|  |  |
| --- | --- |
| S[j] | Element at index j |
| S[start : stop] | Slice including indices [start, stop] |
| S[start: stop: step] | Slice including indices start, start + step, start + 2\*step  … up to but not equaling stop |
| S+t | Concatenation of sequences |
| K \* s | Shorthand for s + s + s + … (k times) |
| Val in s | Containment check |
| Val not in s | Non-containment check |

Python relies on zero-indexing of sequences, thus a sequence of length n has elements indexed from 0 to n-1 inclusive. Python also supports the use of negative indices, which denote a distance from the end of the sequence; index -1 denotes the last element, index -2 the second to last, and so on. Python uses a slicing notation t describe subsequences of a sequences. Slices are described as half-open intervals, with a start index that is included, and a stop index that is excluded. For example, the syntax data[3:8] denotes a subsequence including the five indices: 3, 4, 5, 6, 7. An optional "step", possibly negative, can be indicated as a third parameter of the slice. If a start index or stop index is omitted in the slicing notation, it is presumed to designate the respective extreme of the original sequence.

Because lists are mutable, the syntax s[j] = val can be used to replace an element at a given index. Lists also support a syntax, del s[j], that removes the designated element from the list. Slice notation can also be used to replace or delete a sublist.

The notation val in s can be used for any of the sequences to see if there is an element equivalent to val in the sequence. For strings, the syntax can be used to check for a single character or for a larger substring, as with 'amp' in 'example'.

All sequences define comparison operations based on lexicographic order, performing an element by element comparison until the first difference is found. For example, [5, 6, 9] < [5, 7], because of the entries at index 1. Therefore, the following operations are supported by sequence types:

|  |  |
| --- | --- |
| s == t | Equivalent (element by element) |
| s != t | Not equivalent |
| s < t | Lexicographically less than |
| s <= t | Lexicographically less than or equal to |
| s > t | Lexicographically greater than |
| s >= t | Lexicographically greater than or equal to |

#### Operators for Sets and Dictionaries

Sets and frozensets support the following operators:

|  |  |
| --- | --- |
| key in s | containment check |
| key not in s | non-containment check |
| s1 == s2 | s1 is equivalent to s2 |
| s1 != s2 | s1 is not equivalent to s2 |
| s1 <= s2 | s1 is subset of s2 |
| s1 < s2 | s1 is proper subset of set2 |
| S1 >= s2 | s1 is superset of s2 |
| S1 > s2 | s1 is proper superset of s2 |
| S1 | s2 | the union of s1 and s2 |
| S1 & s2 | the intersection of s1 and s2 |
| S1 – s2 | the set of elements in s1 but not s2 |
| S1 ^ s2 | the set of elements in precisely one of s1 or s2 |

Note well that sets do not guarantee a particular order of their elements, so the comparison operators, such as <, are not lexicographic; rather, they are based on the mathematical notion of a subset. As a result, the comparison operators define a partial order, but not a total order, as disjoint sets are neither "less than", "equal to " or "greater than" each other. Sets also support many fundamental behaviors through named methods (e.g., add, remove); we will explore their functionality more fully in Chapter 10.

#### Extended Assignment Operators

Python supports an extended assignment operator for most binary operators, for example, allowing a syntax such as count += 5. By default, this is a shorthand for the more verbose count = count + 5. For an immutable type, such as a number or a string, one should not presume that this syntax changes the value of the existing object, but instead that it will reassign the identifier to a newly constructed value. (see discussion in Figure 1‑3.) However, it is possible for a type to redefine such semantics to mutate the object, as the list class does for the += operator.

>>> alpha = [1, 2, 3]

>>> beta = alpha

>>> beta += [4, 5] # extends the original list with two more elements

>>> beta = beta + [6, 7] # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]

>>> print(alpha)

[1, 2, 3, 4, 5]

>>> print(beta)

[1, 2, 3, 4, 5, 6, 7]

>>>

This example demonstrates the subtle difference between the list semantics for the syntax beta += foo versus beta = beta + foo.

### Compound Expressions and Operator Precedence

Programming languages must have clear rules for the order in which compound expressions, such as 5 + 2 \* 3, are evaluated. The formal order of precedence for operators in Python are given in Table 1‑3. Operators in a category with higher precedence will be evaluated before those lower precedence, unless the expression is otherwise parenthesized. Therefore, we see that Python gives precedence to multiplication over addition, and therefore evaluates the expression 5 + 2 \* 3 as 5 + (2 \* 3), with value 11, but the parenthesized expression (5 + 2) \* 3 evaluates to value 21. Operators within a category are typically evaluated from left to right, thus 5 – 2 + 3 has value 6. Exceptions to this rule include that unary operators and exponentiation are evaluated from right to left.

>>> 5 + 2\*3

11

>>> (5+2)\*3

21

>>> 5-2+3

6

>>>

Python allows a chained assignment, such as x = y = 0, to assign multiple identifiers to the rightmost value. Python also allows the chaining of the comparison operators. For example, the expression 1 < x + y < 10 is evaluated as the compound (1 < x + y) and (x + y < =10), but without computing the intermediate value x + y twice.

Table 1‑3 Operator precedence in Python, with categories ordered from highest precedence to lowest precedence. When stated, we use expr to denote a literal, identifier, or result of a previously evaluated expression. All operators without explicit mention of expr are binary operators, with syntax expr1 operator expr2.

|  |  |  |
| --- | --- | --- |
| **Operator Precedence** | | |
|  | **Type** | **Symbols** |
| 1 | Member access | Expr.member |
| 2 | Function/method calls  Container subscripts/slices | Expr(…)  Expr[…]=- |
| 3 | Exponentiation | \*\* |
| 4 | Unary operators | +expr, -expr, ~expr |
| 5 | Multiplication, division | \*, /, //, % |
| 6 | Addition, subtraction | +, - |
| 7 | Bitwise shifting | <<, >> |
| 8 | Bitwise-and | & |
| 9 | Bitwise-x or | ^ |
| 10 | Bitwise-or | | |
| 11 | Comparisons  Containment | Is, is not, ==, !=, <, <=, >, >=  In, not in |
| 12 | Logical-not | Not expr |
| 13 | Logical-and | And |
| 14 | Logical-or | Or |
| 15 | Conditional | Val1 if cond else val2 |
| 16 | Assignments | =, +=, -=, \*=, etc. |

## Control Flow

In this section, we review Python's most fundamental control structures: conditional statements and loops. Common to all control structures is the syntax used in Python for defining blocks of code. The colon character is used to delimit the beginning of a block of code that acts as a body for a control structure. If the body can be stated as a single executable statement, it can be technically placed on the same line, to the right of the colon. However, a body is more typically typeset as an indented block starting on the line following the colon. Python relies on the indentation level to designate the extent of that block of code, or any nested blocks of code within. The same principles will be applied when designating the body of a function (see Section 1.5), and the body of a class (see Section 2.3).

### Conditionals

Conditional constructs (also known as if statements) provide a way to execute a chosen block of code based on the run-time evaluation of one or more Boolean expressions. In Python, the most general form of a conditional is written as follows:

if first\_condition:

first\_body

elif second\_condition:

second\_body

elif third\_condition:

third\_body

else:

forth\_body

Each condition is a Boolean expression, and each body contains one or more commands that are to be executed conditionally. If the first condition succeeds, the first body will be executed; no other conditions or bodies are evaluated in that case. If the first condition fails, then the process continues in similar manner with the evaluation of the second condition. The execution of this overall construct will cause precisely one of the bodies to be executed. There may be any number of elif clauses (including zero), and the final else clause is optional. As described on Section 1.2.3, nonboolean types may be evaluated as Booleans with intuitive meanings. For example, if response is a string that was entered by a user, and we want to condition a behavior on this being a nonempty string, we may write:

if response:

as a shorthand for the equivalent,

if response != '':

As a simple example, a robot controller might have the following logic:

if door\_is\_closed:

open\_door()

advance()

Notice that the final command, advance(), is not indented and therefore not part of the conditional body. It will be executed unconditionally (although after opening a closed door).

We may nest one control structure within another, relying on indentation to make clear the extent of the various bodies. Revisiting our robot example, here is a more complex control that accounts for unlocking a closed door.

if door\_is\_closed:

if door\_is\_locked:

unlock\_door()

open\_door()

advance()

The logic expressed by this example can be diagrammed as a traditional flowchart, as portrayed in Figure 1‑6.

True

door\_is\_closed

False True

door\_is\_locked

False

unlock\_door()

open\_door()

advance()

Figure 1‑6 A flowchart describing the logic of a nested conditional statements.

### Loops

Python offers two distinct looping constructs. A while loop allows general repetition based upon the repeated testing of a Boolean condition. A for loop provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range). We discuss both forms in this section.

#### While Loops

The syntax for a while loop in Python is as follows:

while condition:

body

As with an if statement, condition can be an arbitrary Boolean expression, and body can be an arbitrary block of code (including nested control structure). The execution of a while loop begins with a test of the Boolean condition. If that condition evaluates to True, the body of the loop is performed. After each execution of the body, the loop condition is retested, and if it evaluates to True, another iteration of the body is performed. When the conditional test evaluates to False (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop.

As an example, here is a loop that advances an index through a sequence of characters until finding an entry with value 'X' or reaching the end of the sequence.

j = 0

while j < len(data) and data[j] != 'X':

j += 1

The len function, which we will introduce in Section 1.5.2, returns the length of a sequence such as a list or a string. The correctness of this loop relies on the short-circuiting behavior of the and operator, as described on Section 1.3. We intentionally test j < len(data) to ensure that j is a valid index, prior to accessing element data[j] would eventually raise an IndexError when 'X' is not found. (see Section 1.7 for discussion of exceptions.)

As written, when this loop terminates, variable j's value will be the index of the leftmost occurrence of 'X', if found, or otherwise the length of the sequence (which is recognizable as an invalid index to indicate failure of the search). It is worth nothing that this code behaves correctly, even in the special case when the list is empty, as the condition j < len(data) will initially fail and the body of the loop will never be executed.

#### For loops

Python's for-loop syntax is a more convenient alternative to a while loop when iterating through a series of elements. The for-loop syntax can be used on any type of iterable structure, such as a list, tuple str, set, dict, or file (we will discuss iterators more formally in Section1.8). Its general syntax appears as follows.

for element in iterable:

body # body may refer to 'element' as an identifier

For readers familiar with Java, the semantics of Python's for loop is similar to the "for each" loop style introduced in Java 1.5.

As an instructive example of such a loop, we consider the task of computing the sum of a list and numbers. (Admittedly, Python has a built-in function, sum, for this purpose.) We perform the calculation with a for loop as follows, assuming that data identifiers the list:

>>> total = 0

>>> for val in data:

total += val

The loop body executes once for each element of the data sequence, with the identifier, val, from the for-loop syntax assigned at the beginning of each pass to a respective element. It is worth nothing that val is treated as a standard identifier. If the element of the original data happens to be mutable, the val identifier can be used to invoke its methods. But a reassignment of identifier val to a new value has no effect on the original data, nor on the next iteration of the loop.

As a second classic example, we consider the task of finding the maximum value in a list of elements (again, admitting that Python's built-in max function already provides this support). If we can assume that the list, data, has at least one element, we could implement this task as follows:

biggest = data[0]

for val in data:

if val > biggest:

biggest = val

Although we could accomplish both of the above tasks with a while loop, the for-loop syntax had an advantage of simplicity, as there is no need to manage an explicit index into the list nor to author a Boolean loop condition. Furthermore, we can use a for loop in cases for which a while loop does not apply, such as when iterating through a collection, such as a set, that does not support any direct form of indexing.

#### Index-Based For Loops

The simplicity of a standard for loop over the elements of a list is wonderful; however, one limitation of that form is that we do not know where an element resides within the sequence. In some applications, we need knowledge of the index of an element within the sequence. For example, suppose that we want to know where the maximum element in a list resides.

Rather than directly looping over the elements of the list in that case, we prefer to loop over all possible indices of the list. For this purpose, Python provides a built-in class named range that generates integer sequences. (We will discuss generators in Section 1.8.) In simplest form, the syntax range(n) generates the series of n values from 0 to n-1. Conveniently, there are precisely the series of valid indices into a sequence of length n. Therefore, a standard Python idiom for looping through the series of indices of a data sequence uses a syntax:

for j in range(len(data)):

In this case, identifier j is not an element of the data—it is an integer. But the expression data[j] can be used to retrieve the respective element. For example, we can find the index of the maximum element of a list as follows:

big\_index = 0

for j in range(len(data)):

if data[j] > data[big\_index]:

big\_index = j

#### Break and Continue Statements

Python supports a break statement that immediately terminate a while or for loop when executed within its body. More formally, if applied within nested control structures, it causes the termination of the most immediately enclosing loop. As a typical example, here is code that determines whether a target value occurs in a data set:

found = False

for item in data:

if item == target:

found = True

break

Python also supports a continue statement that causes the current iteration of a loop body to stop, but with subsequent passes to the loop proceeding as expected.

We recommend that the break and continue statements be used sparingly. Yet, there are situations in which these commands can be effectively used to avoid introducing overly complex logical conditions.

## Functions

In this section, we explore the creation of and use of functions in Python. As we did in Section 1.2.2, we draw a distinction between functions and methods. We use the general term function to describe a traditional, stateless function that is invoked without the context of a particular class or an instance of that class, such as sorted(data). We use more specific term method to describe a member function that is invoked upon a specific object-oriented message passing syntax, such that data.sort(). In this section, we only consider pure functions; methods will be explored with more general object-oriented principles in Chapter 2.

We begin with an example to demonstrate the syntax for defining functions in Python. The following function counts the number of occurrences of a given target value within any form of iterable data set.

def count(data, target):

n = 0

for item in data:

if item == target:

n += 1

return n

The first line, beginning with the keyword def, serves as the function' signature. This establishes a new identifier as the name of the function (count, in this example), and it establishes the number of parameters that it expects, as well as names identifying those parameters (data and target, in this example). Unlike Java and C++, Python is dynamically typed language, and therefore a Python signature does not designate the types of those parameters, nor the type (if any) of a return value. Those expectations should be stated in the function's documentation (see Section 2.2.3) and can be enforced within the body of the function, but misuse of a function will only be detected at run-time.

The remainder of the function definition is known as the body of the function. As is the case with control structures in Python, the body of a function is typically expressed as an indented block of code. Each time a function is called, Python creates a dedicated activation record that stores information relevant to the current call. This activation record includes what is known as a namespace (see Section 1.10) to manage all identifiers that have local scope within the current call. The namespace includes the function's parameters and any other identifiers that are defined locally within the body of the function. An identifier in the local scope of the function caller has no relation to any identifier with the same name in the caller's scope (although identifiers in different scopes may be aliases to the same object). In our first example, the identifier n has scope that is local to the function call, as does the identifier item, which is established as the loop variable.

#### Return Statement

A return statement is used within the body of a function to indicate that the function should immediately cease execution, and that an expressed value should be returned to the caller. If a return statement is executed without an explicit argument, the None value is automatically returned. Likewise, None well be returned if the flow of control ever reaches the end of a function body without having executed a return statement. Often, a return statement will be the final command within the body of the function, as was the case in our earlier example of a count function. However, there can be multiple return statements in the same function, with conditional logic controlling which such command is executed, if any. As a further example, consider the following function that tests if a value exists in a sequence.

def contains(data, target):

for item in target:

if item == target:

return true

return False

If the conditional within the loop body is ever satisfied, the return True statement is executed and the function immediately ends, with True designating that the target value was found. Conversely, if the for loop reaches its conclusion without ever finding the match, the final return False statement will be executed.

### Information Passing

To be a successful programmer, one must have clear understanding of the mechanism in which a programming language passes information to and from a function. In the context of a function signature, the identifiers used to describe the expected parameters are known as formal parameters, and the objects sent by the caller when invoking the function are the actual parameters. Parameter passing in Python follows the semantics of the standard assignment statement. When a function is invoked, each identifier that serves as a formal parameter is assigned, in the function's local scope, to the respective actual parameter that is provided by the caller of the function.

For example, consider the following call to our count function in Section 1.5:

prizes = count(grade, 'A')

Just before the function body is executed, the actual parameters, grades and 'A', are implicitly assigned to the formal parameters, data and target, as follows:

data = grades

target = 'A'

These assignment statements establish identifier data as an alias for grades and target as a name for the string literal 'A'. (See Figure 1‑7.)

grades data target

list

list

'A'

…

Figure 1‑7: A portrayal of parameter passing in Python, for the functional call

count(grade, 'A'). Identifiers data and target are formal parameters defined

within the local scope of the count function.

The communication of a return value from the function back to the caller is similarly implemented as an assignment. Therefore, with our sample invocation of

prize = count(grade, 'A'), the identifier prizes in the caller's scope is assigned to the object that is identified as n in the return statement within our function body.

An advantage to Python's mechanism for passing information to and from a function is that objects are not copied. This ensures that the invocation of a function is efficient, even in a case where a parameter or return value is a complex object.

#### Mutable Parameters

Python's parameter passing model has additional implications when a parameter is a mutable object. Because the formal parameter is an alias for the actual parameter, the body of the function may interact with the object in ways that change its state. Considering again our sample invocation of the count function, if the body of the function executes the command data.append('F'), the new entry is added to the end of the list identified as data within the function, which is one and the same as the list known to the caller as grades. As an aside, we note that reassigning a new value to a formal parameter with a function body, such as by setting data = [ ], does not alter the actual parameter; such a reassignment simply breaks the alias.

Our hypothetical example of a count method that appends a new element to a list lacks common sense. There is no reason to expect such a behavior, and it would be quite a poor design to have such an unexpected effect on the parameter. There are, however, many legitimate cases in which a function may be designed (and clearly documented) to modify the state of a parameter. As a concrete example, we present the following implementation of a method named scale that's primary purpose is to multiply all entries of a numeric data set by a given factor.

def scale(data, factor):

for j in range(len(data)):

data[j] \*= factor

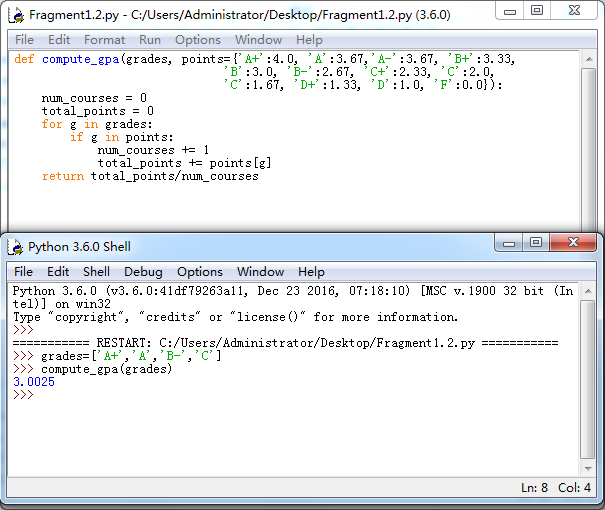
#### Default Parameter Values

Python provides means for functions to support more than one possible calling signature. Such a function is said to be polymorphic (which is Greek for "many forms"). Most notably, functions can declare one or more default values for parameters, thereby allowing the caller to invoke a function with varying numbers of actual parameters. As an artificial example, if a function is declared with signature

def foo (a, b=15, c=27):

there are three parameters, the last two of which offer default values. A caller is welcome to send three actual parameters, as in foo(4, 12, 8), in which case the default values are not used. If, on the other hand, the caller only sends one parameter, foo(4), the function will execute with parameters values a=4, b=15, c=27. If a caller sends two parameters, they assumed to be first two, with the third being the default. Thus, foo(8, 20) executes with a=8, b=20, c=27. However, it is illegal to define a function with a signature such as bar(a, b=15, c) with b having a default value, yet not the subsequent c; if a default parameter value is present for one parameter, it must be present for all further parameters.

As a more motivating example for the use of a default parameter, we revisit the task of computing a student's GPA (see Code Fragment 1‑1). Rather than assume direct input and output with the console, we prefer to design a function that computes and returns a GPA. Our original implementation uses a fixed mapping from each letter grade (such as a B-) to a corresponding point value (such as 2.67). While that point system is somewhat common, it may not agree with the system used by all schools. (For example, some may assign an 'A+' grade a value higher than 4.0.) Therefore, we design a compute\_gpa function, given in Code Fragment 1‑2, which allows the caller to specify a custom mapping from grades to values, while offering the standard point system as a default.



Fragment 1‑2 : A function that computes a student's GPA with a point value system that can be customized as an optional parameter.

As an additional example of an interesting polymorphic function, we consider Python's support for range. (Technically, this is a constructor for the range class, but for the sake of this discussion, we can treat it as a pure function.) Three calling syntaxes are supported. The one-parameter form, range(n), generates a sequence of integers from 0 up to but not including n. A two-parameter form, range(start, stop) generates integers from start up to, but not including, stop. A three-parameter form, range(start, stop, step), generates a similar range as range(start, stop), but with increments of size step rather than 1.

This combination of forms seems to violate the rules for default parameters. In particular, when a single parameter is sent, as in range(n), it serves as the stop value (which is the second parameter); the value of start is effectively 0 in that case. However, this effect can be achieved with some sleight of hand, as follows:

def range(start, stop=None, step=1):

if stop is None:

stop = start

start = 0

...

From a technical perspective, when range(n) is invoked, the actual parameter n will be assigned to formal parameter set. Within the body, if only one parameter is received, the start and stop values are reassigned to provide the desired semantics.

#### Keyword Parameters

The traditional mechanism for matching the actual parameters sent by a caller, to the formal parameters declared by the function signature is based on the concept of positional arguments. For example, with signature foo(a=10, b=20, c=30), parameters sent by the caller are matched, in the given order, to the formal parameters. As invocation of foo(5) indicates that a=5, while b and c are assigned their default values.

Python supports an alternative mechanism for sending a parameter to a function known as a keyword argument. A keyword argument is specified by explicitly assigning an actual parameter to a formal parameter by name. For example, with the above definition of function foo, a call foo(c=5) will invoke the function with parameters a=10, b=20, c=5.

A function's author can require that certain parameters be sent only through the keyword-argument syntax. We never place such a restriction in our own function definitions, but we will see several important uses of keyword-only parameters in Python's standard libraries. As an example, the built-in max function accepts a keyword parameter, coincidentally named key, that can be used to vary the notion of "maximum" that is used.

By default, max operators based upon the natural order of elements according to the < operator for that type. But the maximum can be computed by comparing some other aspect of the element. This is done by providing an auxiliary function that converts a natural element to some other value for the sake of comparison. For example, if we are interested in finding a numeric value with magnitude that is maximal (i.e., considering -35 to be larger than +20), we can use the calling syntax max(a, b, key=abs). In this case, the built-in abs function is itself sent as the value associated with the keyword parameter key. (Functions are first-class objects in Python; see Section 1.10.) When a max is called in this way, it will compare abs(a) to abs(b), rather than a to b. The motivation for the keyword syntax as an alternate to positional arguments is important in the case of max. This function is polymorphic in the number of arguments, allowing a call such as max(a, b, c, d); therefore, it is not possible to designate a key function as a traditional positional element. Sorting functions in Python also support a similar key parameter for indicating a nonstandard order. (We explore this further in Section 9.4 and in Section 12.6.1, when discussing sorting algorithms).

### Python's Built-In Functions

Table 1‑4 provides an overview of common functions that are automatically available in Python, including, the previously discussed abs, max, and range. When choosing names for the parameters, we use identifiers x, y, z for arbitrary comparable types. We use the identifier, iterable, to represent an instance of any iterable type (e.g., str, list, tuple, set, dict); we will discuss iterators and iterable data types in Section 1.8. A sequence represents a more narrow category of indexable classes, including str, list, and tuple, but neither set nor dict. Most of the entries in Table 1‑4 can be categorized according to their functionality as follows:

**Input/Output:** print, input, and open will be more fully explained in Section 1.6.

**Character Encoding:** ord and chr relate characters and their integer code points. For example, ord('A') is 65 and chr(65) is 'A'.

**Mathematics:** abs, divmod, pow, round, and sum provide common mathematical functionality; an additional math module will be introduced in Section 1.11.

**Ordering:** max and min apply to any data type that supports a notion of comparison, or to any collection of such values. Likewise, sorted can be used to produce an ordered list of elements drawn from any existing collection.

**Collections/Iterations:** range generates a new sequence of numbers; len reports the length of any existing collection; functions reversed, all, any, and map operate on arbitrary iterations as well; iter and next provide a general framework for iteration through elements of a collection, and are discussed in Section 1.8.

Table 1‑4 Commonly used built-in functions in Python

|  |  |
| --- | --- |
| **Common Built\_In Functions** | |
| **Calling Syntax** | **Description** |
| abs(x) | Return the absolute value of a number. |
| any(iterable) | Return True if bool(e) is True for each element e. |
| chr(integer) | Return True if bool(e) is True for at least one element e. |
| divmod(x, y) | Return (x//y, x % y) as tuple, if x and y are integers. |
| hash(obj) | Return an integer hash value for the object (see Chapter 10) |
| id(obj) | Return the unique integer serving as an |
| input(prompt) | Return a string from standard input; the prompt is optional. |
| isinstance(obj, cls) | Determine if obj is an instance of the class (or a subclass). |
| iter(iterable) | Return a new iterator object for the parameter(see Section 1.8) |
| len(iterable) | Return the number of elements in the given iteration. |
| map(f, iter1, iter2,…) | Return an iterator yielding the result of function calls f(e1, e2, …)  for respective element e1∈iter1, e2∈iter2, … |
| max(iterable) | Return the largest element of the given iteration. |
| max(a, b, c, …) | Return the largest of the arguments. |
| min(iterable) | Return the smallest element of the given iteration. |
| min(a, b, c, …) | Return the smallest argument. |
| next(iterator) | Return the next element reported by the iterator. (see Section 1.8) |
| open(filename, mode) | Open a file with the given name and access mode. |
| ord(char) | Return the Unicode code point of the given character. |
| pow(x, y) | Return the value xy (as an integer if x and y are integers);  Equivalent to x\*\*y. |
| pow(x, y, z) | Return the value (xy mod z) as an integer. |
| print(obj1, obj2) | Print the arguments, with separating spaces and trailing newline. |
| range(stop) | Construct an iteration of values 0, 2, …, stop-1. |
| range(start, stop) | Construct an iteration of values start, start+1, …, stop-1 |
| range(start, stop, step) | Construct an iteration of values start, start + step, start+2\*step,… |
| reversed(sequence) | Return an iteration of the sequence in reverse. |
| round(x) | Return the nearest int value (a tie is broken toward the even value) |
| round(x, k) | Return the value rounded to the nearest 10-k (return-type matches x) |
| sorted(iterable) | Return a list containing elements of the iterable in sorted order. |
| sum(iterable) | Return the sum of the elements in the iterable (must be numeric) |
| type(obj) | Return the class to which the instance obj belongs. |

## Simple Input and Output

In this section, we address the basics of input and output in Python, describing standard input and output through the user console, and Python's support for reading and writing text files.

### Console Input and Output

#### The print Function

The built-in function, print, is used to generate standard output to the console. In its simplest form, it prints an arbitrary sequence of arguments, separated by spaces, and followed by a trailing newline character. For example, the command print('maroon', 5) outputs the string 'maroon 5\n'. Note that arguments need not be string instances. A nonstring argument x will be displayed as str(x). Without any arguments, the command print() outputs a single newline character.

The print function can be customized through the use of the following keyword parameters (see Section 1.5 for a discussion of keyword parameters):

* 1. By default, the print function inserts a separating space into the output between each pair of arguments. The separator can be customized by providing a desired separating string as a keyword parameter, sep. For example, colon-separated output can be produced as print(a, b, c, sep=' : '). The separating string need not be a single character; it can be a longer string, and it can be the empty string, sep= ' ', causing successive arguments to be directly concatenated.
  2. By default, a trailing newline is output after the final argument. An alternative trailing string can be designated using a keyword parameter, end. Designating the empty string end= ' ' suppresses all trailing characters.
  3. By default, the print function sends its output to the standard console. However, output can be directed to a file by indicating an output file stream (see Section 1.6.2) using file as a keyword parameter.

#### The input Function

The primary means for acquiring information from the user console is a built-in function named input. This function displays a prompt, if given as an optional parameter, and then waits until the user enters some sequence of characters followed by the return key. The formal return value of the function is the string of characters that were entered strictly before the return key (i.e., no newline character exists in the returned string).

When reading a numeric value from the user, a programmer must use the input function to get the string of characters, and then use the int or float syntax to construct the numeric value that character string prebsents. That is, if a call to response = input() reports that the user entered the characters, '2013', the syntax int(response) could be used to produce the integer value 2013. It is quite common to combine these operations with a syntax such as

year = int(input('In what year were you born? '))

if we assume that the user will enter an appropriate response. (In Section 1.7 we discuss error handling in such a situation.)

Because input returns a string as its results, use of that function can be combined with the existing functionality of the string class, as described in Appendix A. For example, if the user enters multiple pieces of information on the same line, it is common to call the split method on the result, as in

>>> reply = input('Enter x and y, separated by spaces: ')

Enter x and y, separated by spaces: 230 233

>>> pieces = reply.split() #return a list of strings, as separated by spaces

>>> x = float(pieces[0])

>>> y = float(pieces[1])

>>> print(x,y)

230.0 233.0

>>>

#### A Sample Program

Here is a sample, but complete, program that demonstrates the use of the input and print functions. The tools for formatting the final output is discussed in Appendix A.

>>> age = int(input('Enter your age in year: '))

Enter your age in year: 22

>>> max\_heart\_rate = 206.9 - (0.67 \* age)

>>> target = 0.65 \* max\_heart\_rate

>>> print('Your target fat-burning heart rate is',target)

Your target fat-burning heart rate is 124.904

>>>

### Files

Files are typically accessed in Python beginning with a call to a built-in function, named open, that returns a proxy for interactions with the underlying file. For example, the command, fp = open('sample.txt'), attempts to open a file named sample.txt, returning a proxy that allows read-only access to the text file.

The open function accepts an optional second parameter that determines the access mode. The default mode is 'r' for reading. Other common modes are 'w' for writing to the file (causing any existing file with that name to be overwritten), or 'a' for appending to the end of an existing file. Although we focus on the use of text files, it is possible to work with binary files, using access modes such as 'rb' or 'wb'.

When processing a file, the proxy maintains a current position within the file as an offset from the beginning, measured in number of bytes. When opening a file with mode 'r' or 'w', the position is initially 0; if opened in append mode, 'a', the position is initially at the end of the file. The syntax fp.close() closes the file associated with proxy fp, ensuring that any written contents are saved. A summary of methods for reading and writing a file is given in Table 1‑5

Table 1‑5 Behaviors for interacting with a text file via a file proxy (named fp).

|  |  |
| --- | --- |
| **Calling Syntax** | **Description** |
| fp.read() | Return the (remaining) contents of a readable file as a string. |
| fp.read(k) | Return the next k bytes of a readable file as a string. |
| fp.readline() | Return (remainder of) the current line of a readable file as a string. |
| fp.readlines() | Return (remaining) lines of a readable file as a list of strings. |
| for line in fp: | Iterate all (remaining) lines of a readable file. |
| fp.seek(k) | Change the current position to be at the kth byte of the file. |
| fp.tell() | Return the current position, measured as byte-offset from the start. |
| fp.write(string) | Write given string at current position of the writable file. |
| fp.writelines(seq) | Write each of the strings of the given sequence at the current  position of the writable file. This command does not insert any  newlines, beyond those that are embedded in the strings. |
| print(…, file=fp) | Redirect output of print function to the file. |

#### Reading from a File

The most basic command for reading via a proxy is the read method. When invoked on file proxy fp, as fp.read(k), the command returns a string representing the next k bytes of the file, starting at the current position. Without a parameter, the syntax fp.read() returns the remaining contents of the file in entirely. For convenience, files can be read a line at a time, using the readline method to read one line, or the readlines method to return a list of all remaining lines. Files also support the for-loop syntax, with iteration being line by line (e.g., for line in fp:).

#### Writing to a File

When a file proxy is writable, for example, if created with access mode 'w' or 'a', text can be written using methods write or writelines. For example, if we define fp=open('results.txt', 'w'), the syntax fp.write('Hello World.\n') writes a single line to the file with the given string. Note well that write does not explicitly add a trailing newline, so desired newline characters must be embedded directly in the string parameter. Recall that the output of the print method can be redirected to a file using a keyword parameter, as described in Section 1.6.

## Exception Handling

Exceptions are unexpected events that occur during the execution of a program. An exception might result from a logical error or an unanticipated situation. In Python, exceptions (also known as errors) are objects that are raised (or thrown) by code that encounters an unexpected circumstance. The Python interpreter can also raise an exception should it encounter an unexpected condition, like running out of memory. A raised error may be caught by a surrounding context that "handles" the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console. In this section, we examine the most common error types in Python, the mechanism for catching and handling errors that have been raised, and the syntax for raising errors from within user-defined blocks of the code.

#### Common Exception Types

Python includes a rich hierarchy of exception classes that designate various categories of errors; Table 1‑6 shows many of those classes. The Exception class serves as a base class for most other types. An instance of the various subclasses encodes details about a problem that has occurred. Several of these errors may be raised in exceptional cases by behaviors introduced in this chapter. For example, use of an undefined identifier in an expression causes a NameError, and errant use of the dot notation, as in foo.bar(), will generate an AttributeError if object foo does not support a member named bar.

Table 1‑6 Common exception classes in Python

|  |  |
| --- | --- |
| **Class** | **Description** |
| Exception | A base class for most error types |
| AttributeError | Raised by syntax obj.foo, if obj has no member named foo |
| EOFError | Raised if "end of file" reached for console or file input |
| IOError | Raised upon failure of I/O operation (e.g., opening file) |
| IndexError | Raised if index to sequence is out of bounds |
| KeyError | Raised if nonexistent key requested for set or dictionary |
| KeyboardInterrupt | Raised if user types Ctrl-C while program is executing |
| NameError | Raised if nonexistent identifier used |
| StopIteration | Raised by next(iterator) if no element; see Section 1.8 |
| TypeError | Raised when wrong type of parameter is sent to a function |
| ValueError | Raised when parameter has invalid value (e.g., sqrt(-5)) |
| ZeroDivisionError | Raised when any division operator used with 0 as divisor |

Sending the wrong number, type, or value of parameters to a function is another common cause for an exception. For example, a call to abs("hello") will raise a TypeError because the parameter is not numeric, and a call to abs(3, 5) will raise a TypeError because one parameter is expected. A ValueError is typically raised when the correct number and type of parameters are sent, but a value is illegitimate for the context of the function. For example, the int constructor accepts a string, as with int("137"), but a ValueError is raised if that string does not represent an integer, as with int("3.14") or int("hello").

Python's sequence type (e.g., list, tuple, and str) raise an IndexError when syntax such as data[k] is used with an integer k that is not a valid index for the given sequence (as described in Section 1.2.3). Sets and dictionaries raise a KeyError when an attempt is made to access a nonexistent element.

### Raising an Exception

An exception is thrown by executing the raise statement, with appropriate instance of an exception class as an argument that designates the problem. For example, if a function for computing a sequence root is sent a negative value as a parameter, it can raise an exception with the command:

raise ValueError('x cannot be negative')

This syntax raises a newly created instance of the ValueError class, with the error message serving as a parameter to the constructor. If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context (and possibly beyond).

When checking the validity of parameters sent to a function, it is customary to first verify that a parameter is of an appropriate type, and then to verify that it has an appropriate value. For example, the sprt function in Python's math library performs error-checking that might be implemented as follows:

>>> def sqrt(x):

if not isinstance(x, (int, float)):

raise TypeError('x must be numeric')

elif x < 0:

raise ValueError('x cannot be negative')

>>> sqrt(-1)

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

sqrt(-1)

File "<pyshell#5>", line 5, in sqrt

raise ValueError('x cannot be negative')

ValueError: x cannot be negative

>>> sqrt('hello')

Traceback (most recent call last):

File "<pyshell#7>", line 1, in <module>

sqrt('hello')

File "<pyshell#5>", line 3, in sqrt

raise TypeError('x must be numeric')

TypeError: x must be numeric

>>>

Checking the type of an object can be performed at run-time using the built-in function, isinstance. In simplest form, isinstance(obj, cls) returns True if object, obj, is an instance of class, cls, or any subclass of that type. In the above example, a more general form is used with a tuple of allowable types indicated with the second parameter. After confirming that the parameter is numeric, the function enforces an expectation that the number be nonnegative, raising a ValueError otherwise.

How much error-checking to perform within a function is a matter of debate. Checking the type and value of each parameter demands additional execution time and, if taken to an extreme, seems counter to the nature of Python. Consider the built-in sum function, which computes a sum of a collection of numbers. As implementation with rigorous error-checking might be written as follows:

def sum(values):

if not isinstance(values, collections.Iterable):

raise TypeError('parameter must be an iterable type')

total = 0

for v in values:

if not isinstance(v, (int, float)):

raise TypeError('elements must be numeric')

total = total + v

return total

>>> import collections

>>> a = [2,3,4]

>>> sum(a)

9

>>>

The abstract base class, collections. Iterable, includes all of Python's iterable containers types that guarantee support for the for-loop syntax (e.g., list, tuple, set); we discuss iterables in Section 1.8, and the use of modules, such as collections, in Section 1.11. Within the body of the loop, each element is verified as numeric before being added to the total. A far more direct and clear implementation of this function can be written as follows:

def sums(values):

total = 0

for v in values:

total = total + v

return total

Interestingly, this simple implementation performs exactly like Python's built-in version of the function. Even without the explicit checks, appropriate exceptions are raised naturally by the code. In particular, if values is not an iterable type, the attempt to use the for-loop syntax raises a TypeError reporting that the object is not iterable. In the case when a user sends an iterable type that includes a nonnumerical element, such as sum([3.14, 'opps']), a TypeError is naturally raised by the evaluation of expression total + v. The error message

unsupported operand type(s) for +: 'float' and 'str'

should be sufficiently informative to the caller. Perhaps slightly less obvious is the error that results from sum(['alpha', 'beta']). It will technically report a failed attempt to add an int and str, due to the initial evaluation of total + 'appha', when total has been initialized to 0.

In the remainder of this book, we tend to favor the simpler implementations in the interest of clean presentation, performing minimal error-checking in most situations.

### Catching an Exception

There are several philosophies regarding how to cope with possible exceptional cases when writing code. For example, if a division x/y is to be computed, there is clear risk that a ZeroDivisionError will be raised when variable y has value 0. In an ideal situation, the logic of the program may dictate that y has nonzero value, thereby removing the concern for error. However, for more complex code, or in a case where the value of y depends on some external input to the program, there remains some possibility of an error.

One philosophy for managing exceptional cases is to "look before you leap." The goal is to entirely avoid the possibility of an exception being raised through the use of a proactive conditional test. Revisiting our division example, we might avoid the offending situation by writing:

if y != 0:

ratio = x/y

else:

… do something else …

A second philosophy, often embraced by Python programmers, is that "*it is easier to ask for forgiveness than it is to get permission*." This quote is attributed to Grace Hopper, an early pioneer in computer science. The sentiment is that we need not spend extra execution time safeguarding against every possible exceptional case, as long as there is a mechanism for coping with a problem after it arises. In Python, this philosophy is implemented using a try-except control structure. Revisiting our first example, the division operation can be guarded as follows:

try:

ratio = x/y

except ZeroDivisionError:

... do something else ...

In this structure, the "try" block is the primary code to be executed. Although it is a single command in this example, it can more generally be a larger block of indented code. Following the try-block are one or more "except" cases, each with an identified error type and an indented block of code that should be executed if the designated error is raised within the try-block.

The relative advantage of using a try-except structure is that the non-exceptional case runs efficiently, without extraneous checks for the exceptional condition. However, handling the exceptional case requires slightly more time when using a try-except clause is best used when there is reason to believe that the exceptional case is relatively unlikely, or when it is prohibitively expensive to proactively evaluate a condition to avoid the exception.

Exception handling is particularly useful when working with user input, or when reading from or writing to files, because such interactions are inherently less predictable. In Section 1.6.2, we suggest the syntax, fp = open('sample.txt') for opening a file with read access. That command may raise an IOError for a variety of reasons, such as a non-existent file, or lack of sufficient privilege for opening a file. It is significantly easier to attempt the command and catch the resulting error than it is to accurately predict whether the command will succeed.

We continue by demonstrating a few other forms of the try-except syntax. Exceptions are objects that can be examined when caught. To do so, an identifier must be established with a syntax as follows:

try:

fp = open('sample.txt')

except IOError as e:

print('Unable to open the file',e)

Unable to open the file [Errno 2] No such file or directory: 'sample.txt'

>>>

In this case, the name, e, denotes the instance of the exception that was thrown, and printing it causes a detailed error message to be displayed (e.g., "file not found").

A try-statement may handle more than one type of exception. For example, consider the following command from Section 1.6.1:

age = int(input('Enter your age in years: '))

This command could fail for a variety of reasons. The call to input will raise an EOFError if the console input fails. If the call to input completes successfully, the int constructor raises a ValueError if the user has not entered characters representing a valid integer. If we want to handle two or more types of errors in the same way, we can use a single except-statement, as in the following example:

>>> age = -1

>>> while age <= 0:

try:

age = int(input('Enter your age in years: '))

if age <= 0:

print('Your age must be positive')

except (ValueError, EOFError):

print('Invalid response')

Enter your age in years: -2

Your age must be positive

Enter your age in years: hello

Invalid response

Enter your age in years: 22

>>>

We use the tuple, (ValueError, EOFError), to designate the types of errors that we wish to catch with the except-clause. In this implementation, we watch either error, print a response, and continue with another pass of the enclosing while loop. We note that when an error is raised within the try-block, the remainder of that body is immediately skipped. In this example, if the exception arises within the call to input, or the subsequent call to the int constructor, the assignment to age never occurs, nor the message about needing a positive value. Because the value of age will be unchanged, the while loop will continue. If we preferred to have the while loop continue without printing the 'Invalid response' message, we could have written the exception-clause as

except (valueError, EOFError):

pass

The keyword, pass, is a statement that does nothing, yet it can serve syntactically as a body of a control structure. In this way, we quietly catch the exception, thereby allowing the surrounding while loop to continue.

In order to provide different responses to different types of errors, we may use two or more except-clauses as part of a try-structure. In our previous example, an EOFError suggests a more insurmountable error than simply an errant value being entered. In this case, we might wish to provide a more specific error message, or perhaps to allow the exception to interrupt the loop and be propagated to a higher context. We could implement such behavior as follows: (press Ctrl-D to see what happens)

>>> age = -1

>>> while age <= 0:

try:

age = int(input('Enter your age in years: '))

if age <= 0:

print('Your age must be positive')

except ValueError:

print('That is an invalid age specification')

except EOFError:

print('There was an unexpected error reading input.')

raise

In this implementation, we have separate except-clauses for the ValueError and EOFError cases. The body of the clause for handling an EOFError relies on another technique in Python. It uses the raise statement without any subsequent argument, to re-raise the same exception, and then to interrupt the whole loop and propagate the exception upward.

In closing, we note two additional features of try-except structures in Python. It is permissible to have a final except-clause without any identified error types, using syntax except:, to catch any other exceptions that occurred. However, this technique should be used sparingly, as it is difficult to suggest how to handle an error of an unknown type. A try-statement can have a finally clause, with a body of code that will always be executed in the standard or exceptional cases, even when an uncaught or re-raised exception occurs. That block is typically used for critical cleanup work, such as closing an open file.

## Iterators and Generators

In Section 1.4.2, we introduced the for-loop syntax beginning as:

for element in iterable:

and we noted that there are many types of objects in Python that qualify as being iterable. Basic container types, such as list, tuple, and set, qualify as iterable types. Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines. User-defined types may also support iteration. In Python, the mechanism for iteration is based upon the following conventions:

1. An iterator is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the built-in function, next(i), produces a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.
2. An iterable is an object, obj, that produces an iterator via the syntax iter(obj).

By these definitions, an instance of a list is an iterable, but not itself an iterator. With data = [1, 2, 4, 8], it is not legal to call next(data). However, an iterator object can be produced with syntax, i = iter(data), and each subsequent call to next(i) will return an element of that list. The for-loop syntax in Python simply automates this process, creating an iterator for the given iterable, and then repeatedly calling for the next element until catching the Stop-Iteration exception.

More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress. However, iterators typically maintain their state with indirect reference back to the original collection of elements. For example, calling iter(data) on a list instance produces an instance of the list\_iterator class. That iterator does not store its own copy of the list of elements. Instead, it maintains a current index into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the iterator is constructed, but before the iteration is complete, the iterator will be reporting the updated contents of the list.

Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once. For example, the call range(1000000) does not return a list of numbers; it returns a range object that is iterable. This object generates the million values one at a time, and only as needed. Such a *lazy evaluation* technique has great advantage. In the case of range, it allows a loop of the form, for j in range(1000000):, to execute without setting aside memory for storing one million values. Also, if such a loop were to be interrupted in some fashion, no time will have been spent computing unused values of the range.

We see lazy evaluation used in many of Python's libraries. For example, the dictionary class supports methods key(), values(), and items(), which respectively produce a "view" of all keys, values, or(key, value) pairs within a dictionary. None of these methods produces an explicit list of results. Instead, the views that are produced are iterable objects based upon the actual contents of the dictionary. An explicit list of values from such an iteration can be immediately constructed by calling the list class constructor with the iteration as a parameter. For example, the syntax list(d.values()) produces a list that has elements based upon the current values of dictionary d. We can similarly construct a tuple or set instance based upon a given iterable.

#### Generators

In Section 2.3.4, we will explain how to define a class whose instances serve as iterators. However, the most convenient technique for creating iterators in Python is through the use of generators. A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of the series. As an example, consider the goal of determining all factors of a positive integer. For example, the number 100 has factors 1, 2, 4, 5, 10, 20, 25, 50, 100. A traditional function might produce and return a list containing all factors, implemented as:

def factors(n):

result = []

for k in range(1, n+1):

if n % k == 0:

result.append(k)

return result

>>> factors(100)

[1, 2, 4, 5, 10, 20, 25, 50, 100]

>>>

In contrast, an implementation of a generator for computing those factors could be implemented as follows:

def factors(n):

for k in range(1, n+1):

if n % k == 0:

yield k

Notice use of the keyword **yield** rather than **return** to indicate a result. This indicates to Python that we are defining a generator, rather than a traditional function. It is illegal to combine yield and return statements in the same implementation, other than a zero-argument return statement to cause a generator to end its execution. If a programmer writes a loop such as for factor in factor(100):, an instance of our generator is created. For each iteration of the loop, Python executes our procedure until a yield statement indicates the next value. At that point, the procedure is temporarily interrupted, only to be resumed when another value is requested. When the flow of control naturally reaches the end of our procedure (or a zero-argument return statement), a StopIteration exception is automatically raised. Although this particular example uses a single yield statement in the source code, a generator can rely on multiple yield statements in different constructs, with the generated series determined by the natural flow of control. For example, we can greatly improve the efficiency of our generator for computing factors of a number, n, by only-testing values up to the square root of that number, while reporting the factor n//k that is associated with each k (unless n//k equals k). We might implement such a generator as follows:

def factors(n):

k = 1

while k \* k < n:

if n % k == 0:

yield k

yield n//k

k += 1

if k \* k == n: # special case if n is perfect square 10\*10=100

yield k

>>> for i in factors(100):

print(i)

1

100

2

50

4

25

5

20

10

>>>

We should note that this generator differs from our first version in that the factors are not generated in strictly increasing order. For example, factors(100) generates the series 1, 100, 2, 50, 4, 25, 5, 20, 10.

In closing, we wish to emphasize the benefits of lazy evaluation when using a generator rather than a traditional function. The results are only computed if requested, and the entire series need not reside in memory at one time. In fact, a generator can effectively produce an infinite series of values. As an example, the Fibonacci numbers form a classic mathematical sequence, starting with value 0, then value 1, and then each subsequent value being the sum of the two preceding values. Hence, the Fibonacci series begins as: 0, 1, 2, 3, 5, 8, 13, … The following generator produces this infinite series.

def fibonacci():

a = 0

b = 1

while True: # or a < 1000000, a < n, something like that

yield a

future = a + b

a = b

b = future

## Additional Python Conveniences

In this section, we introduce several features of Python that are particularly convenient for writing clean, concise code. Each of these syntaxes provide functionality that could otherwise be accomplished using functionality that we have introduced easier in this chapter. However, at times, the new syntax is a more clear and direct expression of the logic.

### Conditional Expressions

Python supports a conditional expression syntax that can replace a simple control structure. The general syntax is an expression of the form:

expr1 if condition else expr2

This compound expression evaluates to expr1 if the condition is true, and otherwise evaluates to expr2. For those familiar with Java or C++, this is equivalent to the syntax, *condition ? expr : expr2*, in those languages.

As an example, consider the goal of sending the absolute value of a variable, n, to a function (and without relying on the built-in abs function, for the sake of example). Using a traditional control structure, we might accomplish this as follows:

if n >= 0:

param = n

else:

param = -n

result = foo(param)

With the conditional expression syntax, we can directly assign a value to variable, param, as follows:

param = n if n >= 0 else -n

result = foo(param)

In fact, there is no need to assign the compound expression to a variable. A conditional expression can itself serve as a parameter to the function, written as follows:

result = foo(n if n >= 0 else -n)

Sometimes, the mere shortening of source of code is advantageous because it avoids the distraction of a more cumbersome control structure. However, we recommend that a conditional expression be used only when it improves the readability of the source code, and when the first of two options is the most "natural" case, given its prominence in the syntax. (We prefer to view the alternative value as more exceptional.)

### Comprehension Syntax

A very common programming task is to produce one series of values based upon the processing of another series. Often, this task can be accomplished quite simply in Python using what is known as a **comprehension syntax**. We begin by demonstrating list comprehension, as this was the first form to be supported by Python. Its general form is as follows:

[ expression **for** value **in** iterable **if** condition]

We note that both expression and condition may depend on value, and that the if-clause is optional. The evaluation of the comprehension is logically equivalent to the following traditional control structure for computing a resulting list:

result = []

for value in iterable:

if condition:

result.append(expression)

As a concrete example, a list of the squares of numbers from 1 to n, that is [1,4,9,16,25, …, n2], can be created by traditional means as follows:

squares = []

for k in range(1, n+1):

squares.append(k\*k)

With list comprehension, this logic is expressed as follows:

squares = [k\*k for k in range(1, n+1)]

As a second example, Section 1.8 introduced the goal of producing a list of factors for an integer n. That task is accomplished with the following list comprehension:

factors = [k for k in range(1, n+1) if n % k == 0]

Python supports a similar comprehension syntax that respectively produce a set, generator, or dictionary. We compare those syntaxes using our example for producing the square of numbers.

>>> n = 9

>>> [ k\*k for k in range(1, n+1)] # list comprehension

[1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> { k\*k for k in range(1, n+1)} # set comprehension

{64, 1, 4, 36, 9, 16, 49, 81, 25}

>>> ( k\*k for k in range(1, n+1)) # generator comprehension

<generator object <genexpr> at 0x01EE4CC0>

>>> { k : k\*k for k in range(1, n+1)} # dictionary comprehension

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>>

>>> total = sum(k\*k for k in range(1, n+1))

>>> print(total)

285

>>>

The generator syntax is particularly attractive when results do not need to be stored in memory. For example, to compute the sum of the first n squares, the generator syntax,

total = sum(k\*k for k in range(1, n +1)),

is preferred to the use of an explicitly list comprehension as the parameter.

### Packing and Unpacking of Sequences

Python provides two additional conveniences involving the treatment of tuples and other sequence types. The first is rather cosmetic. If a series of comma-separated expressions are given in a larger context, they will be treated as a single tuple, even if no enclosing parentheses are provided. For example, the assignment

>>> data = 2,4,6,8

>>> data

(2, 4, 6, 8)

>>>

Result in identifier, data, being assigned to the tuple (2, 4, 6, 8). This behavior is called automatic packing of a tuple. One common use of packing in Python is when returning multiple values from a function. If the body of a function executes the command,

return x, y

it will be formally returning a single object that is the tuple (x, y).

As a dual to the packing behavior, Python can automatically unpack a sequence. As an example, we can write

>>> a, b, c, d = range(7, 11)

>>> a

7

>>>

which has the effect of assigning a = 7, b = 8, c = 9, and d = 10, as those are the four values in the sequence returned by the call to range. For this syntax, the right-hand side expression can be any iterable type, as long as the number of variables on the left-hand side is the same as the number of elements in the iteration.

This technique can be used to unpack tuples returned by a function. For example, the built-in function, divmod(a, b), returns the pair of values (a//b, a % b) associated with an integer division. Although the caller can consider the return value to be a single tuple, it is possible to write

quotient, remainder = divmod(a, b)

to separately identify the two entries of the returned tuple. This syntax can also be used in the context of a for loop, when iterating over a sequence of iterables, as in

>>> for x, y in [(7,2),(5,8),(6,4)]:

print(x, y)

7 2

5 8

6 4

>>>

In this example, there will be three iterations of the loop. During the first pass, x=7 and y=2, and so on. This style of loop is quite commonly used to iterate through key-value pairs that are returned by the items() method of the dict class, as in:

for k, v in mapping.items()

#### Simultaneous Assignments

The combination of automatic packing and unpacking forms a technique known as **simultaneous assignment**, whereby we explicitly assign a series of values to a series of identifiers, using a syntax:

x, y, z = 6, 2, 5

In effect, the right-hand side of this assignment is automatically packed into a tuple, and then automatically unpacked with its elements assigned to the three identifiers on the left-hand side.

When using a simultaneous assignment, all of the expressions are evaluated on the right-hand side before any of the assignments are made to the left-hand variables. This is significant, as it provides a convenient means for swapping the values associated with two variables:

j, k = k, j

With this command, j will be assigned to the old value of k, and k will be assigned to the old value of j. Without simultaneous assignment, a swap typically requires more delicate use of a temporary variable, such as

temp = j

j = k

k = temp

With the simultaneous assignment, the unnamed tuple representing the packed values on the right-hand side implicitly serves as the temporary variable when performing such a swap.

The use of simultaneous assignments can greatly simplify the presentation of code. As an example, we reconsider the generator in Section 1.8 that produces the Fibonacci series. Within each pass of the loop, the goal was to reassign a and b, respectively, to the values of b and a+b. At the time, we accomplished this with brief use of a third variable. With simultaneous assignments, that generator can be implemented more directly as follows:

def fibonacci():

a, b = 0, 1

while True:

yield a

a, b = b, a +b

>>> print(fibonacci)

<function fibonacci at 0x02216858>

>>> for i in fibonacci():

print(i)

## Scopes and Namespaces

When computing a sum with the syntax x + y in Python, the names x and y must have been previously associated with objects that serves as values; a NameError will be raised if no such definitions are found. The process of determining the value associated with an identifier is known as **name resolution**.

Whenever an identifier is assigned to a value, that definition is made with a specific scope. Top-level assignments are typically made in what is known as global scope. Assignments made within the body of a function typically have scope that is local to that function call. Therefore, an assignment, x = 5, within a function has no effect on the identifier, x, in the broader scope.

Each distinct scope in Python is represented using an abstraction known as a namespace. A namespace manages all identifiers that are currently defined in a given scope. Figure 1‑8 portrays two namespaces, one being that of a caller to our count function from Section 1.5, and the order being the local namespace during the execution of that function.

float

str

3.65

'A'

n

gpa

int

target

gradee

2

str

list

data

major

'CS'

item

str

str

str

'A-'

'B+'

'A-'

Figure 1‑8 A portrayal of the two namespaces associated with a user's call count(grade, 'A'),

as defined in Section 1.5. The left namespace is the caller's and the right namespace

represents the local scope of the function.

Python implements a namespace with its own dictionary that maps each identifying string (e.g., 'n') to its associated value. Python provides several ways to examine a given namespace. The function, dir, reports the names of the identifiers in a given namespace (i.e., the keys of the dictionary), while the function, vars, returns the full dictionary. By default, calls to dir() and vars() report on the most locally enclosing namespace in which they are executed.

When an identifier is indicated in a command, Python searches a series of namespaces in the process of name resolution. First, the most locally enclosing scape is searched for a given name. If not found there, the next outer scope is searched, and so on. We will continue our examination of namespaces, in Section 2.5, when discussing Python's treatment of object-orientation. We will see that each object has its own namespace to store its attribute, and that classes each have a namespace as well.

#### First-Class Objects

In the terminology of programming languages, first-class objects are instances of a type that can be assigned to an identifier, passed as a parameter, or returned by a function. All of the data types we introduced in Section 1.2.3, such as int and list, are clearly first-class types in Python. In Python, functions and classes are also treated as first-class objects. For example, we could write the following:

>>> scream = print

>>> scream('Hello world!')

Hello world!

>>>

In this case, we have not created a new function, we have simply defined scream as an alias for the existing print function. While there is little motivation for precisely this example, it demonstrates the mechanism that is used by Python to allow one function to be passed as a parameter to another. In Section 1.5.2, we noted that the built-in function, max, accepts an optional keyword parameter to specify a non-default order when computing a maximum. For example, a caller can use the syntax max(a, b, key=abs), to determine which value has the larger absolute value. Within the body of that function, the formal parameter, key, is an identifier that will be assigned to the actual parameters, abs.

In terms of namespaces, an assignment such as scream = print, introduces the identifier, scream, into the current namespace, with its value being the object that represents the built-in function, print. The same mechanism is applied when a user-defined function is declared. For example, our count function from Section 1.5 beings with the following syntax:

def count(data, target):

…

Such a declaration introduces the identifier, count, into the current namespace, with the value being a function instance representing its implementation. In similar fashion, the name of a newly defined class is associated with a representation of that class as its value. (Class definitions will be introduced in the next chapter.)

## Modules and the Import Statement

We have already introduced many functions (e.g., max) and classes (e.g., list) that are defined within Python's built-in namespace. Depending on the version of Python, there are approximately 130-150 definitions that were deemed significant enough to be included in that built-in namespace.

Beyond the built-in definitions, the standard Python distribution includes perhaps tens of thousands of other values, functions, and classes that are organized in additional libraries, known as **modules**, that can be **imported** from within a program. As an example, we consider the math module. While the built-in namespace includes a few mathematical functions (e.g., abs, min, max, round), many more are relegated to the math module (e.g., sin, cos, sqrt). That module also defines approximate values for the mathematical constants, pi and e.

Python's import statement loads definitions from a module into the current namespace. One form of an import statement uses a syntax such as the following:

from math import pi, sqrt

This command adds both pi and sqrt, as defined in the math module, into the current namespace, allowing direct use of the identifier, pi, or a call of the function, sqrt(2). If there are many definitions form the same module to be imported, an asterisk may be used as a wild card, as in **from** math **import** \*, but this form should be used sparingly. The danger is that some of the names defined in the module may conflict with names already in the current namespace (or being imported from another module), and the import causes the new definitions to replace existing ones.

Another approach that can be used to access many definitions from the same module is to import the module itself, using a syntax such as:

import math

Formally, this adds the identifier, math, to the current namespace, with the module as its value. (Modules are also first-class objects in Python.) Once imported, individual definitions from the module can be accessed using a fully-qualified name, such as math.pi or math.sqrt(2).

#### Creating a new Module

To create a new module, one simply has to put the relevant definitions in a file named with a .py suffix. Those definitions can be imported from any other .py file within the same project directory. For example, if we were to put the definition of our count function (see Section 1.5) into a file named utility.py, we could import that function using the syntax,

from utility import count

It is worth nothing that top-level commands with the module source code are executed when the module is first imported, almost as if the module were its own script. There is a special construct for embedding commands within the module that will be executed if the module is directly invoked as a script, but not when the module is imported from another script. Such commands should be placed in a body of a conditional statement of the following form,

if \_\_name\_\_ == '\_\_main\_\_':

Using our hypothetical utility.py module as an example, such commands will be executed if the interpreter is started with a command **python utility.py**, but not when the utility module is imported into another context. This approach is often used to embed what known as **unit tests** within the module; we will discuss unit testing further in Section 2.2.4.

### Existing Modules

Table 1‑7 provides a summary of a few available modules that are relevant to a study of data structures. We have already discussed the math module briefly. In the remainder of this section, we highlight another module that is particularly important for some of the data structures and algorithms that we will study later in this book.

Table 1‑7 Some existing Python modules relevant to data structures and algorithms.

|  |  |
| --- | --- |
| **Existing Moduels** | |
| **Module Name** | **Description** |
| array | Provides compact array storage for primitive types. |
| collections | Defines additional data structures and abstract base class  Involving collections of objects |
| copy | Defines general functions for making copies of objects. |
| heapq | Provides heap-based priority queue functions (see Section 9.3.7). |
| math | Defines common mathematical constants and functions. |
| os | Provides support for interactions with the operating system. |
| random | Provides random number generation. |
| re | Provides support for processing regular expressions. |
| sys | Provides additional level of interactions with the Python interpreter. |
| time | Provides support for measuring time, or delaying a program. |

#### Pseudo-Random Number Generation

Python's random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessary truly random). A pseudo-random number generator uses a deterministic formula to generate the next number un a sequence based upon one or more past numbers that is has generated. Indeed, a simple yet popular pseudo-random number generator chooses its next number based solely on the most recently chosen number and some additional parameters using the following formula.

next = (a\*current + b) % n;

where a, b, and n are appropriately chosen integers. Python uses a more advanced technique known as a **Mersenne twister.** It turns out that the sequences generated by these techniques can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers, such as games. For applications, such as computer security settings, where one needs unpredictable random sequences, this kind of formula should not be used. Instead, one should ideally sample from a source that is actually random, such as radio static coming from outer space.

Since the next number is a pseudo-random generator is determined by the previous number(s), such a generator always needs a place to start, which is called its seed. The sequence of numbers generated for a given seed will always be the same. One common trick to get a different sequence each time a program is run is to use a seed that will be different for each run. For example, we could use some timed input from a user or the current system time in milliseconds.

Python's random module provides support for pseudo-random number generation by defining a Random class; instances of that class serve as generators with independent state. This allows different aspects of a program to rely on their own pseudo-random number generator, so that calls to one generator do not affect the sequence of numbers produced by another. For convenience, all of the methods supported by a Random class are also supported as stand-alone functions of the random module (essentially using a single generator instance for all top-level calls).

Table 1‑8 Methods supported by instances of the Random class, and as top-level functions of the random module.

|  |  |
| --- | --- |
| **Syntax** | **Description** |
| seed(hashable) | Initializes the pseudo-random number generator  based upon the hash value of the parameter |
| random() | Returns a pseudo-random floating-point value  in the interval [0.0, 1.0]. |
| randint(a, b) | Returns a pseudo-random integer in the closed  interval [a, b]. |
| randrange(start, stop, step) | Returns a pseudo-random integer in the standard  Python range indicated by the parameter. |
| choice(seq) | Returns an element of the given sequence chosen  pseudo-randomly. |
| shuffle(seq) | Reorders the elements of the given sequence  pseudo-randomly. |

## Exercises

#### Reinforcement

**R-1.1** Write a short Python function, is\_multiple(n, m), that takes two integer values and returns True if n is a multiple of m, that is, n = mi for some integer i, and False otherwise.

def is\_multiple(n, m):

if n % m == 0:

return True

else:

return False

**R-1.2** Write a short Python function, is\_even(k), that takes an integer value and returns True if k is even, and False otherwise. However, your function cannot use the multiplication, modulo, or division operators.

def is\_even(k):

abs\_k = abs(k)

while abs\_k > 0:

abs\_k -= 2

if abs\_k == 0:

return True

else:

return False

**R-1.3** Write a short Python function, minmax(data), that takes a sequence of one or more numbers, and returns the smallest and largest numbers, in the form of a tuple of length two. Do not use the built-in functions min or max in implementing your solution.

def minmax(data):

min\_one = 0

max\_one = 0

for i in range(len(data)):

if data[i] < min\_one:

min\_one = data[i]

elif data[i] > max\_one:

max\_one = data[i]

return min\_one, max\_one

>>> minmax((2, 5, 6, -1))

(-1, 6)

**R-1.4** Write a short Python function that takes a positive integer n and returns the sum of the squares of all the positive integers smaller than n.

def sub\_squares(n):

result = 0

for i in range(1,n):

result += i\*i

return result

>>> sub\_squares(5)

30

>>> sub\_squares(2)

1

>>>

**R-1.5** Given a single command that computes the sum from Exercise R-1.4, relying on Python's comprehension syntax and the built-in sum function.

result = sum(i\*i for i in range(1, n))

**R-1.6** Write a short Python function that takes a positive integer n and returns the sum of the squares of all the odd positive integers smaller than n.

def odd\_squares(n):

result = 0

for i in range(1, n):

if i %2 != 0:

result += i\*i

if n <= 0:

result = "Please type a positive integer"

return result

**R-1.7** Give a single command that computes the sum from Exercise R-1.6, relying on Python's comprehension syntax and the built-in sum function.

**Solution:**

result = sum(i\*i for i in range(1,9) if i % 2 != 0)

**R-1.8** Python allows negative integers to be used as indices into a sequence, such as a string. If a string s has length n, and expression s[k] is used for index -n≤k<0, what is the equivalent index j≥0 such that s[j] references the same element?

**Solution:**

-n≤j<0

**R-1.9** What parameters should be sent to the range constructor, to produce a range with values 50, 60, 70, 80?

**Solution:**

range(50, 81, 10)

**R-1.10** What parameters should be sent to the range constructor, to produce a range with values 8, 6, 4, 2, 0, -2, -4, -6, -8?

**Solution:**

for i in range(-8,9,2):

print(-i)

**R-1.11** Demonstrate how to use Python's list comprehension syntax to produce the list [1, 2, 4, 8, 16, 32, 64, 128, 256].

**Solution:**

>>> some\_list = [2\*\*i for i in range(0,9)]

>>> some\_list

[1, 2, 4, 8, 16, 32, 64, 128, 256]

**R-1.12** Python's random module includes a function choice(data) that returns a random element from a non-empty sequence. The random module includes a more basic function randrange, with parameterization similar to the built-in range function, that return a random choice from the given range. Using only the randrange function, implement your own version of the choice function.

**Solution:**

def choice(data):

result = data[randrange(0,len(data))]

return result

>>> data = "Hello World!"

>>> choice(data)

'o'

>>> choice(data)

'l'

#### Creativity

**C-1.13** Write a pseudo-code description of a function that reverses a list of n integers, so that the numbers are listed in the opposite order than they were before, and compare this method to an equivalent Python function for doing the same thing (reversed(data)).

**Solution:**

>>> data = [1,2,3,4,5]

>>> data\_reversed = []

>>> for i in range(len(data)):

data\_reversed.append(data[-i-1])

>>> data\_reversed

[5, 4, 3, 2, 1]

>>>

**C-1.14** Write a short function that takes a sequence of integer values and determines if there is a distinct pair of numbers in the sequence whose product if odd.

**Solution:**

def is\_odd(data):

odd, even = 0, 0

for i in range(len(data)):

if data[i] % 2 == 0 and data[i] != 0:

even += 1

elif data[i] % 2 != 0:

odd += 1

if odd > 0 and even > 0:

return True

return False

>>> is\_odd([2,0,4])

False

>>> is\_odd([2,3,4])

True

>>>

**C-1.15** Write a Python function that takes a sequence of numbers and determines if all the numbers are different from each other (that is, they are distinct).

**Solution:**

def is\_distinct(data):

ordered\_data = sorted(data)

for i in range(len(data)-1):

if ordered\_data[i] == ordered\_data[i+1]:

return False

return True

**C-1.16** In our implementation of the scale function, the body of the loop executes the command data[j] \*= factor. We have discussed that numeric types are immutable, and that use of the \*= operator in this context causes the creation of a new instance (not the mutation of an existing instance). How is it still possible, then, that our implementation of the scale changes the actual parameter sent by the caller?

**Solutions:**

(See more in Section 1.5.1)

When a function is invoked, each identifier that serves as a formal parameter is assigned, in the function's local scope, to the respective actual parameter that is provided by the caller of the function.

def scale(data, factor):

for i in range(len(data)):

data[i] \*= factor

return data

>>> data = [1,2,3,4,5]

>>> scale(data, 3)

[3, 6, 9, 12, 15]

>>> data

[3, 6, 9, 12, 15]

>>>

**C-1.17** Had we implemented the scale function as follows, does it work properly?

def scale(data, factor):

for val in data:

val \*= factor

return data

Explain why or why not.

**Solution:**

No, it doesn't..

val \*= factor created a new val variable, but did not assign it's value to the element in data.

**C-1.18** Demonstrate how to use Python's list comprehension syntax to produce the list [0, 2, 6, 12, 20, 30, 42, 56, 72, 90].

**Solution:**

>>> list1 = [i\*(i+1) for i in range(10)]

>>> list1

[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]

**C-1.19** Demonstrate how to use Python's list comprehension syntax to produce the list ['a', 'b', …, 'z'], but without having to type all 26 such characters literally.

**Solution:**

list2 = [chr(i+97) for i in range(26)]

>>> list2

['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']

**C-1.20** Python's random module includes a function shuffle(data), that accepts a list of elements and randomly reorders the elements so that each possible order occurs with equal probability. The random module includes a more basic function randint(a, b) that returns a uniformly random integer from a to b (including both endpoints). Using only the randint function, implement your own version of the shuffle function.

**Solution:**

>>> from random import randint

def shuffle(data):

len\_data = len(data)

for i in range(len\_data):

rand\_int = randint(i,len\_data-1)

data[i],data[rand\_int] = data[rand\_int], data[i]

return data

>>> data = [1,2,3,4,5]

>>> shuffle(data)

[4, 2, 1, 5, 3]

>>> shuffle(data)

[4, 5, 2, 3, 1]

>>>

**C-1.21** Write a Python program that repeatedly reads lines from standard input until an EOFError is raised, and then outputs those lines in reverse order (a user can indicate end of input by typing Ctrl-D).

**Solution:**

try:

list3 = []

while True:

a = input('Type something: ')

list3.append(a)

except:

print('The reversed list:')

for i in range(len(list3)):

print(list3[-i-1])

**C-1.22** Write a short Python program that takes two arrays a and b of length n storing int values, and returns the dot product of a and b. That is, it returns an array c of length n such that c[i] = a[i] \* b[i], for i=0, …, n-1.

**Solution:**

def dot\_product(A1, A2):

A3 = []

for i in range(len(A1)):

A3.append(A1[i] \* A2[i])

return A3

>>> A1 = [2,3,4,5]

>>> A2 = [5,4,3,2]

>>> dot\_product(A1,A2)

[10, 12, 12, 10]

**C-1.23** Give an example of a Python code fragment that attempts to write an element to a list based on an index that may be out of bounds. If that index is out of bounds, the program should catch the exception that results, and print the following error message:

"Don't try buffer overflow attacks in Python!"

**Solution:**

>>> L1 = [1,2,3,4]

try:

L1[4] = 5

except IndexError as e:

print(e, "\nDo't try buffer overflow attacks in Python!")

list assignment index out of range

Do't try buffer overflow attacks in Python!

>>>

**C-1.24** Write a short Python function that counts the number of vowels in a given character string.

**Solution:**

def count\_vowels(string):

vowels = ['a','e','i','o','u','A','E','I','O','U']

count = 0

for i in range(len(string)):

if string[i] in vowels:

count += 1

return count

**C-1.25** Write a short Python function that takes a string s, representing a sequence, and returns a copy of the string with all punctuation removed. For example, if given the string "Let's try, Mike.", this function would return "Lets try Mike."

**Solution:**

def remove\_punc(String):

S = [' ']

for i in range(65,90):

S.append(chr(i))

for j in range(97,123):

S.append(chr(j))

right = len(String)

result = ''

for k in range(len(String)):

if String[k] in S:

result += String[k]

return result

>>> String = "Let's try, Mike."

>>> remove\_punc(String)

'Lets try Mike'

>>>

**C-1.26** Write a short program that takes as input three integers, a, b and c from the console and determines if they can be used in a correct arithmetic formula (in the given order), like "a+b=c", "a=b-c", "a\*b=c"

**Solution:**

def is\_arithmetic():

a = int(input("Type an integer: "))

b = int(input("Type an integer: "))

c = int(input("Type an integer: "))

if a+b == c or a + c == b or b + c == a:

return True

elif a \* b == c or b \* c == a or a \*c == b:

return True

return False

>>> is\_arithmetic

<function is\_arithmetic at 0x024558A0>

>>> is\_arithmetic()

Type an integer: 2

Type an integer: 3

Type an integer: 5

True

>>>

**C-1.27** In Section 1.8, we provided three different implementations of a generator that computes factors of a given integer. The third of those implementations was the most efficient, but we noted that it did not yield the factors in increasing order, modify the generator so that it reports factors in increasing order, while maintaining its general performance advantages.

**Solution:**

def factor(n):

k = 1

L = []

while k \* k < n:

if n % k == 0:

L.append(k)

L.append(n // k)

k += 1

if k \* k == n:

L.append(k)

sorted\_L = sorted(L)

for i in sorted\_L:

yield i

**C-1.28** The p-norm of a vector v = (v1, v2, …, vn), in n-dimensional space is defined as

For the special case of p = 2, this results in the traditional Euclidean norm, which represents the length of vector. For example, the Euclidean norm of .

Given an implementation of a function named norm such that norm(v, p) returns the p-norm value of v and norm(v) returns the Euclidean norm of v. You may assume that v is a list of numbers.

**Solution:**

def norm(v,p=2):

from math import sqrt

square = 0

for i in v:

square += i\*\*p

left = 0

right = square

ans = (left + right)/2

while abs(ans \*\* p - square) > 0.00001:

if ans\*\*p < square:

left = ans

else:

right = ans

ans = (left + right)/2

return ans

#### Projects

**P-1.29** Write a Python program that outputs all possible strings formed by using the characters 'c', 'a', 't', 'd', 'o', and 'g' exactly once.

**Solution:**

def \_permute(result, temp, nums):

if nums == []:

result += [temp]

else:

for i in range(len(nums)):

\_permute(result, temp + [nums[i]], nums[:i] + nums[i+1:])

>>> result = []

>>> \_permute(result, [], L)

>>> for i in range(len(result)):

s = ''

for j in result[i]:

s += j

print(s)

**P-1.30** Write a Python program that can take a positive integer greater than 2 as input and write out the number of times one must repeatedly divide this number by 2 before getting a value less than 2.

**Solution:**

>>> a = int(input('Please type a positive integer: '))

Please type a positive integer: 5555555

>>> count = 0

>>> while a > 2:

a /= 2

count +=1

>>> count

22

>>> a \* 2\*\*22

5555555.0

>>>

def p131():

a = int(input('Please type a positive integer: '))

count = 0

while a > 2:

a /= 2

count += 1

return count

**P-1.31** Write a Python program that can "make change." Your program should take two numbers as input, one that is a monetary amount charged and the other that is a monetary given. It should then return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins can be based on the monetary system of any current or former government. Try to design your program so that it returns as few bills and coins as possible.

**Solution:**

def make\_change(charge, given):

diff = given - charge

C\_100 = diff // 100

rest1 = diff - C\_100\*100

C\_50 = rest1 // 50

rest2 = rest1 - C\_50 \* 50

C\_20 = rest2 // 20

rest3 = rest2 - C\_20 \* 20

C\_10 = rest3 // 10

rest4 = rest3 - C\_10 \* 10

C\_1 = rest4 // 1

msg = """The change:\n100 yuan: %s\n50 yuan: %s\n20 yuan: %s\n10 yuan: %s\ncoins: %s"""%(C\_100, C\_50, C\_20, C\_10, C\_1)

return msg

>>> print(make\_change(888,1000))

The change:

100 yuan: 1

50 yuan: 0

20 yuan: 0

10 yuan: 1

coins: 2

>>>

>>> print(make\_change(6888, 10000))

The change:

100 yuan: 31

50 yuan: 0

20 yuan: 0

10 yuan: 1

coins: 2

>>>

**P-1.32** Write a Python program that can simulate a simple calculator, using the console as the exclusive input and output device. That is, each input to the calculator, be it a number, like 12.34 or 1034, or an operator, like + or =, can be done on a separate line. After each such input, you should output to the Python console what would be displayed on your calculator.

(http://www.cnblogs.com/Wxtrkbc/p/5453349.html)

**Solution:**

while True:

a = float(input('>'))

b = input('>')

c = float(input('>'))

d = input('>')

if d == '=':

if b == '+':

print(a+c)

elif b == '-':

print(a-c)

elif b == '\*':

print(a\*c)

elif b == '/':

print(a/c)

else:

print("Please type something")

>25

>-

>50

>=

-25.0

>

**P-1.34** A common punishment for school children is to write out a sentence multiple times. Write a Python stand-alone program that will write out the following sentence one hundred times: "I will never spam my friends again." Your program should number each of the sentences and it should make eight difference random-looking typos.

**Solution:**

L = ['I will never spam my friends again.', "Trust me, I won't do that again.", "I won't spam my friends again", 'Do not spam friends in the future', 'Never spam my friends.', 'Never spam my friends', 'Remember not to spam friends','I swear not to spam my friends again.']

for i in range(100):

j = randint(0,7)

result = L[j]

print(result)

**P-1.35** The birthday paradox says that the probability that two people in a room will have the same birthday is more than half, provided n, the number of people in the room, is more than 23. This property is not really a paradox, but many people find it surprising. Design a Python program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for n = 5, 10, 20, …, 100.

**Solution:**

def is\_distinct(L,n):

SL = sorted(L)

for i in range(n-1):

if SL[i] == SL[i+1]:

return 1

return 0

def test\_bir(n,m):

"""n for the number of people, m for the test times."""

from random import randint

all\_L = []

count = 0

for i in range(m):

all\_L.append([])

for j in range(n):

all\_L[i].append(randint(1,365))

count += is\_distinct(all\_L[i],n)

return count/m

>>> test\_bir(23,1000)

0.501

>>> test\_bir(23,10000)

0.5074

>>> test\_bir(5,1000)

0.03

>>> test\_bir(10,1000)

0.113

>>> test\_bir(30,1000)

0.688

>>> test\_bir(50,1000)

0.968

>>> test\_bir(100,1000)

1.0

**P-1.36** Write a Python program that inputs a list of words, separated by whitespace, and outputs how many times each word appears in the list. You need not worry about efficiency at this point, however, as this topic is something that will be addressed later in this book.

**Solution:**

>>> D = {}

>>> S = input("Type some words, separated by whitespace:")

Type some words, separated by whitespace:hello world hello everyone

>>> L = S.split(' ')

>>> print(L)

['hello', 'world', 'hello', 'everyone']

>>> for i in range(len(L)):

if L[i] not in D.keys():

value = 1

D[L[i]] = value

else:

D[L[i]] += 1

>>> D

{'hello': 2, 'world': 1, 'everyone': 1}

>>>

>>> L1 = [i for i in D.keys()]

>>> L2 = [i for i in D.values()]

>>> for i in range(len(D)):

print("The word %s comes out %s times."%(L1[i],L2[i]))

The word hello comes out 2 times.

The word world comes out 1 times.

The word everyone comes out 1 times.

>>>

**P-1.33** Write a Python program that simulate a handheld calculator. Your program should process input from the Python console representing buttons that are "pushed", and then output the contents of the screen after each operation is performed. Minimally, your calculator should be able to process the basic arithmetic operations and a reset/clear operation.

**Solution:**

http://www.cnblogs.com/Wxtrkbc/p/5453349.html