Collection Data Types

Sequence Types

- A sequence type is one that supports the membership operator (in), the size function (len()), slices ([]), and is iterable.
- Python provides five built-in se- quence types: bytearray, bytes, list, str, and tuple.
- Some other sequence types are provided in the stan- dard library, most notably, collections.namedtuple.
- When iterated, all of these sequences provide their items in order.

Tuples

- A tuple is an ordered sequence of zero or more object references.
- Support the same slicing and striding syntax as strings.
- Like strings, tuples are immutable, so we cannot replace or delete any of their items.
- Want to be able to modify an ordered sequence, we simply use a list instead of a tuple.
- Already have a tuple but want to modify it, we can convert it to a list using the list() conversion.
- Tuple data type can be called as a function, tuple()—with no arguments it returns an empty tuple, with a tuple argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a tuple. It does not accept more than one argument.

- An empty tuple is created using empty parentheses, (), and a tuple of one or more items can be created by using commas.
- Tuples must be enclosed in parentheses to avoid syntactic ambiguity.
- To pass the tuple 1, 2, 3 to a function, we would write function((1, 2, 3)).

- Strings have a character at every position, tuples have an object reference at each position.

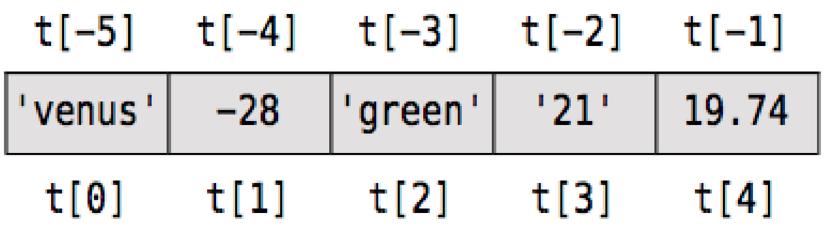


Figure 3.1 *Tuple index positions*

- Tuples provide just two methods, t.count(x), which returns the number of times object x occurs in tuple t, and t.index(x), which returns the index position of the leftmost occurrence of object x in tuple t—or raises a ValueError exception if there is no x in the tuple.

- Tuples can be used with the operators + (concatenation), * (repli- cation), and [] (slice), and with in and not in to test for membership.
- The += and *= augmented assignment operators can be used even though tuples are immutable—behind the scenes Python creates a new tuple to hold the result and sets the left-hand object reference to refer to it;

- Standard comparison operators (<, <=, ==, !=, >=, >), with the comparisons being applied item by item (and recursively for nested items such as tuples inside tuples).

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:]  # same as: hair[1:]
('brown', 'blonde', 'red')
```

```
 >>> hair[:2], "gray", hair[2:]
(('black', 'brown'), 'gray', ('blonde', 'red'))
```

- 3-tuple that contains two 2-tuples.

- This happened because we used the comma operator with three items (a tuple, a string, and a tuple).

- To get a single tuple with all the items we must concatenate tuples:

```
>>> hair[:2] + ("gray",) + hair[2:]
('black', 'brown', 'gray', 'blonde', 'red')
```

- To make a 1-tuple the comma is essential, but in this case, if we had just put in the comma we would get a TypeError (since Python would think we were trying to concatenate a string and a tuple), so here we must have the comma and parentheses.

- When we have tuples on the left-hand side of a binary operator or on the right-hand side of a unary statement, we will omit the parentheses, and in all other cases we will use parentheses.

```
  a, b = (1, 2)             # left of binary operator
  del a, b                 # right of unary statement
  def f(x):
      return x, x ** 2     # right of unary statement
  for x, y in ((1, 1), (2, 4), (3, 9)): # left of binary operator prin
t(x, y)
```

- Some programmers prefer to always use parentheses—which is the same as the tuple representational form, whereas others use them only if they are strictly necessary.

```
  >>> eyes = ("brown", "hazel", "amber", "green", "blue", "gray")
  >>> colors = (hair, eyes)
  >>> colors[1][3:-1]
  ('green', 'blue')
```

- Nestedcollectionstoany level of depth can be created.

```
  >>> things = (1, -7.5, ("pea", (5, "Xyz"), "queue"))
  >>> things[2][1][1][2]
  'z'
```

- Tuples are able to hold any items of any data type, including collection types such as tuples and lists, since what they really hold are object references.

- To give names to particular index positions.

```
  >>> MANUFACTURER, MODEL, SEATING = (0, 1, 2)
  >>> MINIMUM, MAXIMUM = (0, 1)
  >>> aircraft = ("Airbus", "A320-200", (100, 220))
  >>> aircraft[SEATING][MAXIMUM]
  220
```

- This is certainly more meaningful than writing aircraft[2][1], but it involves creating lots of variables.

- When we have a sequence on the right-hand side of an assignment (here we have tuples), and we have a tuple on the left-hand side, we say that the right-hand side has been unpacked.

- Sequence unpacking can be used to swap values.

```
a, b = (b, a)
```

- Sequence unpacking in the context of for ⋯ in loops.

```
for x, y in ((-3, 4), (5, 12), (28, -45)):
    print(math.hypot(x, y))
```

- Here we loop over a tuple of 2-tuples, unpacking each 2-tuple into variables x and y.

Named Tuples

- A named tuple behaves just like a plain tuple, and has the same performance characteristics.
- What it adds is the ability to refer to items in the tuple by name as well as by index position.
- Collections module provides the namedtuple() function.

- This function is used to create custom tuple data types.

```
Sale = collections.namedtuple("Sale",
            "productid customerid date quantity price")
```

- First argument to collections.namedtuple() is the name of the custom tuple data type that we want to be created.

- Second argument is a string of space separated names, one for each item that our custom tuples will take.
- First argument, and the names in the second argument, must all be valid Python identifiers.
- The function returns a custom class (data type) that can be used to create named tuples.

- In object-oriented terms, every class created this way is a subclass of tuple;

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

- The price of the first sale item is sales[0][-1] (i.e., 7.99)—but we can also use names, which makes things much clearer:

```
total = 0
for sale in sales:
    total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total)) # prints: Total $42.46
```

```
>>> Aircraft = collections.namedtuple("Aircraft",
                                              "manufacturer model seating")
>>> Seating = collections.namedtuple("Seating", "minimum maximum")
>>> aircraft = Aircraft("Airbus", "A320-200", Seating(100, 220))
>>> aircraft.seating.maximum
220
```

- When it comes to extracting named tuple items for use in strings there are three main approaches we can take.

```
>>> print("{0} {1}".format(aircraft.manufacturer, aircraft.model))
Airbus A320-200
```

- In Python 3.1 we could reduce this format string to just "{} {}".
- But this approach means that we must look at the arguments passed to str.format() to see what the replacement texts will be.
- This seems less clear than using named fields in the format string.

```
  "{0.manufacturer} {0.model}".format(aircraft)
```

- Used a single positional argument and used named tuple attribute names as field names in the format string.
- This is much clearer than just using positional arguments alone, but it is a pity that we must specify the positional value (even when using Python 3.1). Fortunately, there is a nicer way.
- Named tuples have a few private methods—that is, methods whose name begins with a leading underscore. One of them—namedtuple._asdict()

```
 "{manufacturer} {model}".format(**aircraft._asdict())
```

- The private namedtuple._asdict() method returns a mapping of key–value pairs, where each key is the name of a tuple element and each value is the corresponding value.
- We have used mapping unpacking to convert the mapping into key–value arguments for the str.format() method.

> ★Private methods such as namedtuple._asdict() are not guaranteed to be available in all Python 3.x versions; although the namedtuple._asdict() method is available in both Python 3.0 and 3.1.

Lists

- A list is an ordered sequence of zero or more object references.
- Lists support the same slicing and striding syntax as strings and tuples.

- Unlike strings and tuples, lists are mutable, so we can replace and delete any of their items.
- It is also possible to insert, replace, and delete slices of lists.
- list data type can be called as a function, list()—with no arguments it returns an empty list, with a list argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a list.
- It does not accept more than one argument.
- An empty list is created using empty brackets, [], and a list of one or more items can be created by using a comma-separated sequence of items inside brackets.
- Since all the items in a list are really object references, lists, like tuples, can hold items of any data type, including collection types such as lists and tuples.

- Standard comparison operators (<, <=, ==, !=, >=, >), with the comparisons being applied item by item (and recursively for nested items such as lists or tuples inside lists).

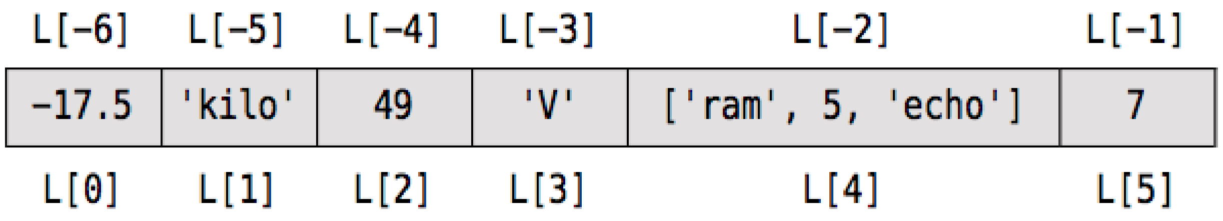


Figure 3.2 *List index positions*

- And given this list, L, we can use the slice operator—repeatedly if necessary—to access items in the list.

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

- Lists can be nested, iterated over, and sliced, the same as tuples.

- Lists support membership testing with in and not in, concatenation with +, extending with += (i.e., the appending of all the items in the right-hand operand), and replication with *and* \=.
- Lists can also be used with the built-in len() function, and with the del statement.
- We can use the slice operator to access items in a list, in some situa- tions we want to take two or more pieces of a list in one go.
- This can be done by sequence unpacking.
- Anyiterable(lists,tuples,etc.) can be unpacked using the sequence unpacking operator, an asterisk or star (*).

- When used with two or more variables on the left-hand side of an assignment, one of which is preceded by *, items are assigned to the variables, with all those left over assigned to the starred variable.

```
>>> first, *rest = [9, 2, -4, 8, 7]
```

```
  >>> first, rest
  (9, [2, -4, 8, 7])
  >>> first, *mid, last = "Charles Philip Arthur George Windsor".split
()
  >>> first, mid, last
  ('Charles', ['Philip', 'Arthur', 'George'], 'Windsor')
  >>> *directories, executable = "/usr/local/bin/gvim".split("/")
  >>> directories, executable
  (['', 'usr', 'local', 'bin'], 'gvim')
```

- The sequence unpacking operator is used like this, the expression *rest, and similar expressions, are called starred expressions.
- Python also has a related concept called starred arguments.

```
  def product(a, b, c):
      return a * b * c # here, * is the multiplication operator
```

- we can call it with three arguments, or by using starred arguments:

```
  >>> product(2, 3, 5)
  30
  >>> L = [2, 3, 5]
  >>> product(*L)
  30
  >>> product(2, *L[1:])
  30
```

Table 3.1 List Methods

Syntax	Description
L.append(x)	Appends item x to the end of list L
L.count(x)	Returns the number of times item x occurs in list L

L.extend(m)
L += m
	Appends all of iterable m’s items to the end of list L; the operator += does the same thing

L.index(x, start, end)	Returns the index position of the leftmost occurrence of item x in list L (or in the start:end slice of L); otherwise, raises a ValueError exception
L.insert(i, x)	Inserts item x into list L at index position int i
L.pop()	Returns and removes the rightmost item of list L

L.pop(i)	Returns and removes the item at index position int i in L
L.remove(x)	Removes the leftmost occurrence of item x from list L, or raises a ValueError exception if x is not found
L.reverse()	Reverses list L in-place
L.sort(⋯)	Sorts list L in-place; this method accepts the same key and reverse optional arguments as the built-in sorted()

- Three-item list is unpacked by the * operator, so as far as the function is concerned it has received the three arguments it is expecting.
- There is never any syntactic ambiguity regarding whether operator * is the multiplication or the sequence unpacking operator.
- When it appears on the left-hand side of an assignment it is the unpacking operator, and when it appears elsewhere (e.g., in a function call) it is the unpacking operator when used as a unary operator and the multiplication operator when used as a binary operator.

- Can iterate over the items in a list using the syntax for item in L:.

```
for i in range(len(L)):
    L[i] = process(L[i])
```

- The built-in range() function returns an iterator that provides integers. With one integer argument, n, the iterator range() returns, producing 0, 1, ⋯, n - 1.

Deleting Items Using the del Statement

- del statement, When applied to an object reference that refers to a data item that is not a collection, the del statement unbinds the object reference from the data item and deletes the object reference.

```
>>> x = 8143 # object ref. 'x' created; int of value 8143 created
>>> x
8143
>>> del x # object ref. 'x' deleted; int ready for garbage collectio
n
>>> x
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

- When an object reference is deleted, Python schedules the data item to which it referred to be garbage-collected if no other object references refer to the dataitem.

- Python provides two solutions to the nondeterminism.
- One is to use a try ⋯ finally block to ensure that cleanup is done, and another is to use a with

statement
- When del is used on a collection data type such as a tuple or a list, only the object reference to the collection is deleted.
- The collection and its items (and for those items that are themselves collections, for their items, recursively) are scheduled for garbage collection if no other object references refer to the collection.
- For mutable collections such as lists, del can be applied to individual items or slices—in both cases using the slice operator, [].
- If the item or items referred to are removed from the collection, and if there are no other object references referring to them, they are scheduled for garbage collection.

- To increment all the numbers in a list of integers. For example:

```
for i in range(len(numbers)):
    numbers[i] += 1
```

- We can extend the list in either of two ways:

```
woods += ["Kauri", "Larch"]
# OR
woods.extend(["Kauri", "Larch"])
```

- Individual items can be added at the end of a list using list.append().

- Items can be inserted at any index position within the list using list.insert(), or by assigning to a slice of length 0.

- Can insert a new item at index position 2 (i.e., as the list's third item) in either of two ways:

```
woods[2:2] = ["Pine"]
#OR
woods.insert(2, "Pine")
```

- Individual items can be replaced in a list by assigning to a particular index position, for example, woods[2] = "Redwood".

- Entire slices can be replaced by assigning an iterable to a slice, for example, woods[1:3] = ["Spruce", "Sugi", "Rimu"].
- The slice and the iterable don't have to be the same length.
- Inallcases, the slice's items are removed and the iterable's items are inserted.
- This makes the list shorter if the iterable has fewer items than the slice it replaces, and longer if the iterable has more items than the slice.
- Items can be removed in a number of other ways.
- We can use list.pop() with no arguments to remove the rightmost item in a list—the removed item is also returned.
- Similarly we can use list.pop() with an integer index argument to remove (and return) an item at a

particular index position.

- Another way of removing an item is to call list.remove() with the item to be removed as the argument.
- The del statement can also be used to remove individual items—for example, del woods[4]—or to remove slices of items.

- Slices can also be removed by assigning an empty list to a slice, so these two snippets are equivalent:

```
woods[2:4] = []
#OR
del woods[2:4]
```

- In the left-hand snippet we have assigned an iterable (an empty list) to a slice, so first the slice is removed, and since the iterable to insert is empty, no insertion takes place.

- x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], and we want to set every odd-indexeditem(i.e.,x[1],x[3],etc.) to0.
- Here is the complete solution: x[1::2] = [0] * len(x[1::2]).
- Now list x is [1, 0, 3, 0, 5, 0, 7, 0, 9, 0].
- Used the replication operator *, to produce a list consisting of the number of 0s we needed based on the length (i.e., the number of items) of the slice.
- The interesting aspect is that when we assign the list [0, 0, 0, 0, 0] to the strided slice, Python correctly replaces x[1]'s value with the first 0, x[3]'s value with the second 0, and so on.
- Lists can be reversed and sorted in the same way as any other iterable using the built-in reversed() and sorted() functions.
- Lists also have equivalent methods, list.reverse() and list.sort(), both of which work in-place (so they don't return anything).
- One common idiom is to case-insensitively sort a list of strings—for example, we could sort the woods list like this: woods.sort(key=str.lower).
- The key argument is used to specify a function which is applied to each item, and whose return value is used to perform the comparisons used when sorting.
- For inserting items, lists perform best when items are added or removed at the end (list.append(), list.pop()).
- The worst performance occurs when we search for items in a list, for example, using list.remove() or list.index(), or using in for membership testing.
- If fast searching or membership testing is required, a set or a dict may be a more suitable collection choice.
- lists can provide fast searching if they are kept in order by sorting them—Python's sort algorithm is especially well optimized for sorting partially sorted lists—and using a binary search (provided by the bisect module), to find items.

List Comprehensions

- Small lists are often created using list literals, but longer lists are usually created programmatically.

- For a list of integers we can use list(range(n)), or if we just need an integer iterator, range() is sufficient, but for other lists using a for…in loop is very common.

```
leaps = []
for year in range(1900, 1940):
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    leaps.append(year)
```

- When the built-in range() function is given two integer arguments, n and m, the iterator it returns produces the integers n, n + 1, ⋯, m - 1.
- A list comprehension is an expression and a loop with an optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items.
- The simplest form of a list comprehension is this:

```
[item for item in iterable]
```

- This will return a list of every item in the iterable, and is semantically no different from list(iterable).
- Two general syntaxes for list comprehensions:

```
[expression for item in iterable]
[expression for item in iterable if condition]
```

- The second syntax is equivalent to:

```
temp = []
for item in iterable:
    if condition:
        temp.append(expression)
```

- Normally, the expression will either be or involve the item.
- List comprehension does not need the temp variable needed by the for ⋯ in loop version.

```
leaps = [y for y in range(1900, 1940)]
leaps = [y for y in range(1900, 1940) if y % 4 == 0]
leaps = [y for y in range(1900, 1940)
        if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

- It is possible to nest list comprehensions. This is the equivalent of having nested for ⋯ in loops.

```
codes = []
for sex in "MF":                # Male, Female
for size in "SMLX": # Small, Medium, Large, eXtra large
    if sex == "F" and size == "X":
            continue
        for color in "BGW":  # Black, Gray, White
            codes.append(sex + size + color)
```

- This produces the 21 item list, ['MSB', 'MSG', ⋯, 'FLW']. The same thing can be achieved in just a couple of lines using a list comprehension:

```
codes = [s + z + c for s in "MF" for z in "SMLX" for c in "BGW"
if not (s == "F" and z == "X")]
```

- Here, each item in the list is produced by the expression s + z + c.

- list comprehension where we skip invalid sex/size combinations in the innermost loop, whereas the nested for ⋯ in loops version skips invalid combinations in its middle loop.
- If the generated list is very large, it may be more efficient to generate each item as it is needed rather than produce the whole list at once.
- This can be achieved by using a generator rather than a list comprehension

Set Types

- A set type is a collection data type that supports the membership operator (in), the size function (len()), and is iterable.
- In addition, set types at least provide a set.isdisjoint() method, and support for comparisons, as well as support for the bitwise operators.
- Python provides two built-in set types: the mutable set type and the immutable frozenset.
- When iterated, set types provide their items in an arbitrary order.
- Only hashable objects may be added to a set.
- Hashable objects are objects which have a **hash**() special method whose return value is always the same throughout the object's lifetime, and which can be compared for equality using the **eq**() special method.
- All the built-in immutable data types, such as float, frozenset, int, str, and tuple, are hashable and can be added to sets.
- The built-in mutable data types, such as dict, list, and set, are not hashable since their hash value changes depending on the items they contain, so they cannot be added to sets.
- Set types can be compared using the standard comparison operators (<, <=, ==, !=, >=, >).
- Note that although == and != have their usual meanings, with the comparisons being applied item by item (and recursively for nested items such as tuples or frozen sets inside sets), the other comparison operators perform subset and superset comparisons.

Sets

- A set is an unordered collection of zero or more object references that refer to hashable objects.

- Sets are mutable, so we can easily add or remove items, but since they are unordered they have no notion of index position and so cannot be sliced or strided.

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```

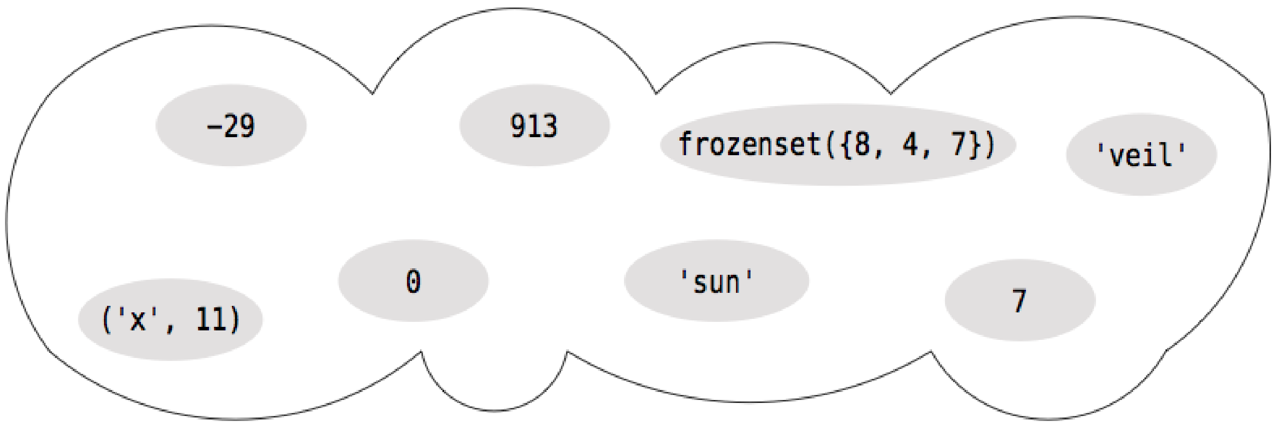
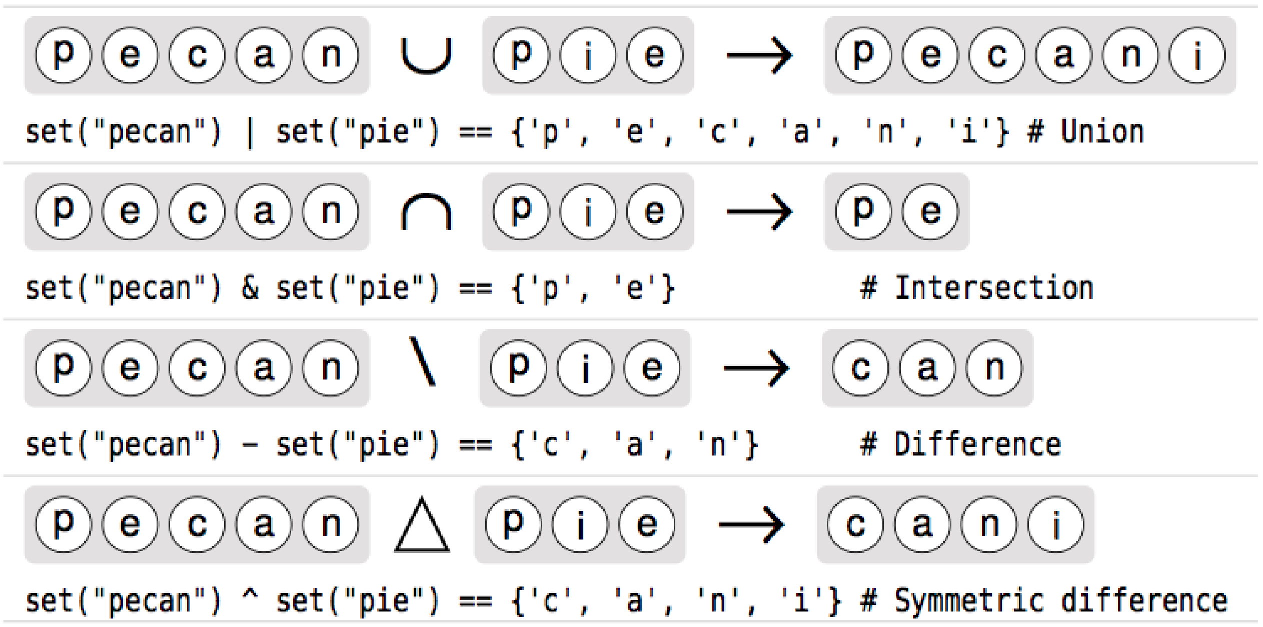


Figure 3.3 *A set is an unordered collection of unique items.*

- The set data type can be called as a function, set()—with no arguments it returns an empty set, with a set argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a set.

- It does not accept more than one argument.
- Nonempty sets can also be created without using the set() function, but the empty set must be created
- Using set(), not using empty braces.
- A set of one or more items can be created by using a comma-separated sequence of items inside braces.
- Another way of creating sets is to use a set comprehension.
- Sets always contain unique items—adding duplicate items is safe but pointless.
- In view of this, sets are often used to eliminate duplicates.

- Sets support the built-in len() function, and fast membership testing with in and not in.



```
set("pecan") | set("pie") == {'p', 'e', 'c', 'a', 'n', 'i'} # Union
set("pecan") & set("pie") == {'p', 'e'}            # Intersection
set("pecan") - set("pie") == {'c', 'a', 'n'}       # Difference
set("pecan") ^ set("pie") == {'c', 'a', 'n', 'i'} # Symmetric difference
```

Figure 3.4 *The standard set operators*

- All the “update” methods (set.update(), set.intersection_update(), etc.) accept any iterable as their argument—but the equivalent operator versions (|=, &=, etc.) require both of their operands to be sets.
- One common use case for sets is when we want fast membership testing.

```
if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
```

- Another common use case for sets is to ensure that we don’t process duplicate data.

> ★Empty braces, {}, are used to create an empty dict

Table3.2 SetMethodsandOperators

Syntax	Description
s.add(x)	Adds item x to set s if it is not already in s
s.clear()	Removes all the items from set s
s.copy()	Returns a shallow copy of set s❄
s.difference(t) s-t	Returns a new set that has every item that is in set s that is not in set t❄
s.difference_update(t) s -= t	Removes every item that is in set t from set s
s.discard(x)	Removes item x from set s if it is in s; see also set.remove()
	Returns a new set that has each item that is in both set s and set

s.intersection(t) s&t	t❄
s.intersection_update(t) s &= t	Makes set s contain the intersection of itself and set t
s.isdisjoint(t)	Returns True if sets s and t have no items in common❄
s.issubset(t) s <= t	Returns True if set s is equal to or a subset of set t; use s < t to test whether s is a proper subset of t❄
s.issuperset(t) s >= t	Returns True if set s is equal to or a superset of sett; uses >t to test whethers is a proper superset of t❄
s.pop()	Returns and removes a random item from set s, or raises a KeyError exception if s is empty
s.remove(x)	Removes item x from set s, or raises a KeyError exception if x is not in s; see also set.discard()
s.symmetric_difference(t) ^t	set s and every item that is in set t, but exclud- Returns a new set that has every item that is in s ❄ ing items that are in both sets
s.symmetric_difference_update(t) s ^= t	Makes set s contain the symmetric difference of itself and set t
s.union(t) `s OR t`	Returns a new set that has all the items in set s and all the items in set t that are not in set s❄
s.update(t) `s OR= t`	Adds every item in set t that is not in set s, to set s

> ❄This method and its operator (if it has one) can also be used with frozensets.

```
seen = set()
for ip in ips:
    if ip not in seen:
        seen.add(ip)
        process_ip(ip)


# OR
for ip in set(ips):
    process_ip(ip)
```

- For the left-hand snippet, if we haven’t processed the IP address before, we add it to the seen set and process it; otherwise, we ignoreit.
- Fortheright-handsnippet, we only ever get each unique IP address to process in the first place.

- The differences between the snippets are first that the left-hand snippet creates the seen set which the right-hand snippet doesn't need, and second that the left-hand snippet processes the IP addresses in the order they are encountered in the ips iterable while the right-hand snippet processes them in an arbitrary order.
- In theory the right-hand approach might be slower if the number of items in ips is very large, since it creates the set in one go rather than incrementally.

- Sets are also used to eliminate unwanted items.

```
filenames = set(filenames)
for makefile in {"MAKEFILE", "Makefile", "makefile"}:
    filenames.discard(makefile)
```

- This code will remove any makefile that is in the list using any of the standard capitalizations.

- Achieved in one line using the set difference (-) operator:

```
filenames = set(filenames) - {"MAKEFILE", "Makefile", "makefile"}
```

- Can also use set.remove() to remove items, although this method raises a KeyError exception if the item it is asked to remove is not in the set.

Set Comprehensions

- In addition to creating sets by calling set(), or by using a set literal, we can also create sets using set comprehensions.

- A set comprehension is an expression and a loop with an optional condition enclosed in braces.

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

- Can use these to achieve a filtering effect (providing the order doesn't matter).

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

- Given a list of filenames in files, this set comprehension makes the set html hold only those filenames that end in .htm or .html, regardless of case.

- Just like list comprehensions, the iterable used in a set comprehension can itself be a set comprehension (or any other kind of comprehension).

Frozen Sets

- A frozen set is a set that, once created, cannot be changed.
- Can of course rebind the variable that refers to a frozen set to refer to something else, though.
- Frozen sets can only be created using the frozenset data type called as a function.
- With no arguments, frozenset() returns an empty frozen set, with a frozenset argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a frozenset.
- It does not accept more than one argument.
- Frozen sets are immutable, they support only those methods and operators that produce a result without affecting the frozen set or sets to which they are applied.

- Frozen sets support

    - frozenset.copy(),
    - frozenset.difference() (-),
    - frozenset.intersection() (&),
    - frozenset.isdisjoint(),
    - frozenset.issubset() (<=; also < for proper subsets),
    - frozenset.issuperset() (>=; also > for proper supersets),
    - frozenset.union() (|),
    - frozenset.symmetric_difference() (^)

- If a binary operator is used with a set and a frozen set, the data type of the result is the same as the left-hand operand's data type.

- So if f is a frozen set and s is a set, f & s will produce a frozen set and s & f will produce a set.
- Inthe case of the == and != operators, the order of the operands does not matter, and f == s will produce True if both sets contain the same items.
- Another consequence of the immutability of frozen sets is that they meet the hashable criterion for set items, so sets and frozen sets can contain frozen sets.

Mapping Types

- A mapping type is one that supports the membership operator (in) and the size function (len()), and is iterable.
- Mappings are collections of key–value items and provide methods for accessing items and their keys and values.
- When iterated, unordered mapping types provide their items in an arbitrary order.
- Python 3.0 provides two unordered mapping types, the built-in dict type and the standard library's collections.defaultdict type.
- A new, ordered mapping type, collections.OrderedDict, was introduced with Python 3.1; this is a dictionary that has the same methods and properties (i.e., the same API) as the built-in dict, but stores its items in insertion order.★
- Only hashable objects may be used as dictionary keys, so immutable data types such as float, frozenset, int, str, and tuple can be used as dictionary keys, but mutable types such as dict, list, and set cannot.

- On the other hand, each key's associated value can be an object reference referring to an object of any type, including numbers, strings, lists, sets, dictionaries, functions, and so on.
- Dictionary types can be compared using the standard equality comparison op- erators (== and !=), with the comparisons being applied item by item (and recur- sively for nested items such as tuples or dictionaries inside dictionaries).
- Comparisons using the other comparison operators (<, <=, >=, >) are not supported since they don't make sense for unordered collections such as dictionaries.

Dictionaries

- A dict is an unordered collection of zero or more key–value pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type.
- Dictionaries are mutable, so we can easily add or remove items, but since they are unordered they have no notion of index position and so cannot be sliced or strided.
- The dict data type can be called as a function, dict()—with no arguments it returns an empty dictionary, and with a mapping argument it returns a dictionary based on the argument;
- For example, returning a shallow copy if the argument is a dictionary.
- It is also possible to use a sequence argument, providing that each item in the sequence is itself a sequence of two objects, the first of which is used as a key and the second of which is used as a value.
- Alternatively, for dictionaries where the keys are valid Python identifiers, keyword arguments can be used, with the key as the keyword and the value as the key's value.

- Dictionaries can also be created using braces—empty braces, {}, create an empty dictionary; nonempty braces must contain one or more commaseparated items, each of which consists of a key, a literal colon, and a value.

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

- The built-in zip() func- tion that is used to create dictionary d4 returns a list of tuples, the first of which has the first items of each of the zip() function's iterable arguments, the second of which has the second items, and so on.

```
d = {"root": 18, "blue": [75, "R", 2], 21: "venus", -14: None,
     "mars": "rover", (4, 11): 18, 0: 45}
```

- Dictionary keys are unique, so if we add a key–value item whose key is the same as an existing key, the effect is to replace that key's value with a new value.

- Brackets are used to access individual values.
- Brackets can also be used to add and delete dictionary items. To add an item we use the = operator, for

example, d[“X”] = 59.

- And to delete an item we use the del statement—for example, del d[“mars”] will delete the item whose key is “mars” from the dictionary, or raise a KeyError exception if no item has that key.

- Items can also be removed (and returned) from the dictionary using the dict.pop() method.

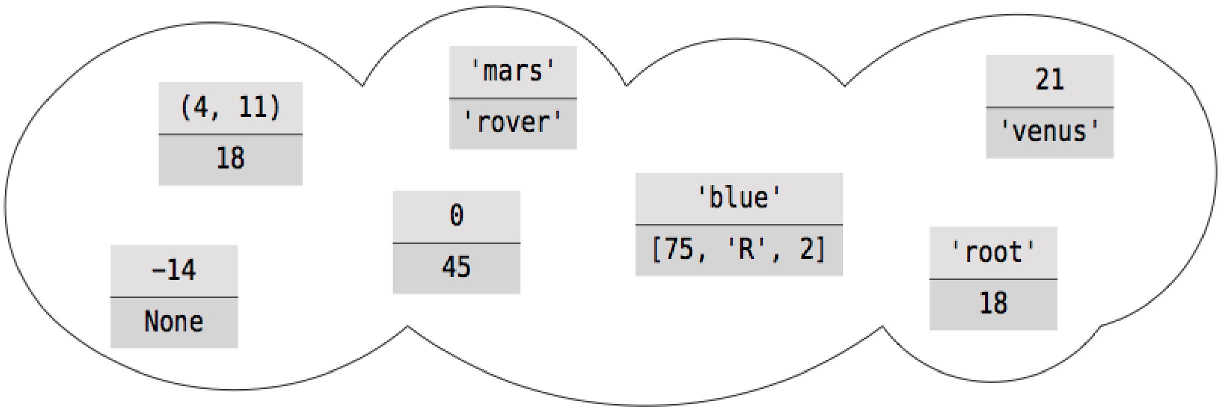


Figure 3.5 *A dictionary is an unsorted collection of (key, value) items with unique keys.*

- Dictionaries support the built-in len() function, and for their keys, fast membership testing with in and not in.

- Because dictionaries have both keys and values, we might want to iterate over a dictionary by (key, value) items, by values, or by keys.

- For example, here are two equivalent approaches to iterating by (key, value)

```
for item in d.items():
    print(item[0], item[1])

# OR
for key, value in d.items():
    print(key, value)
```

- Iterating over a dictionary’s values is very similar:

```
 for value in d.values():
    print(value)
```

- To iterate over a dictionary’s keys we can use dict.keys(), or we can simply treat the dictionary as an iterable that iterates over its keys.

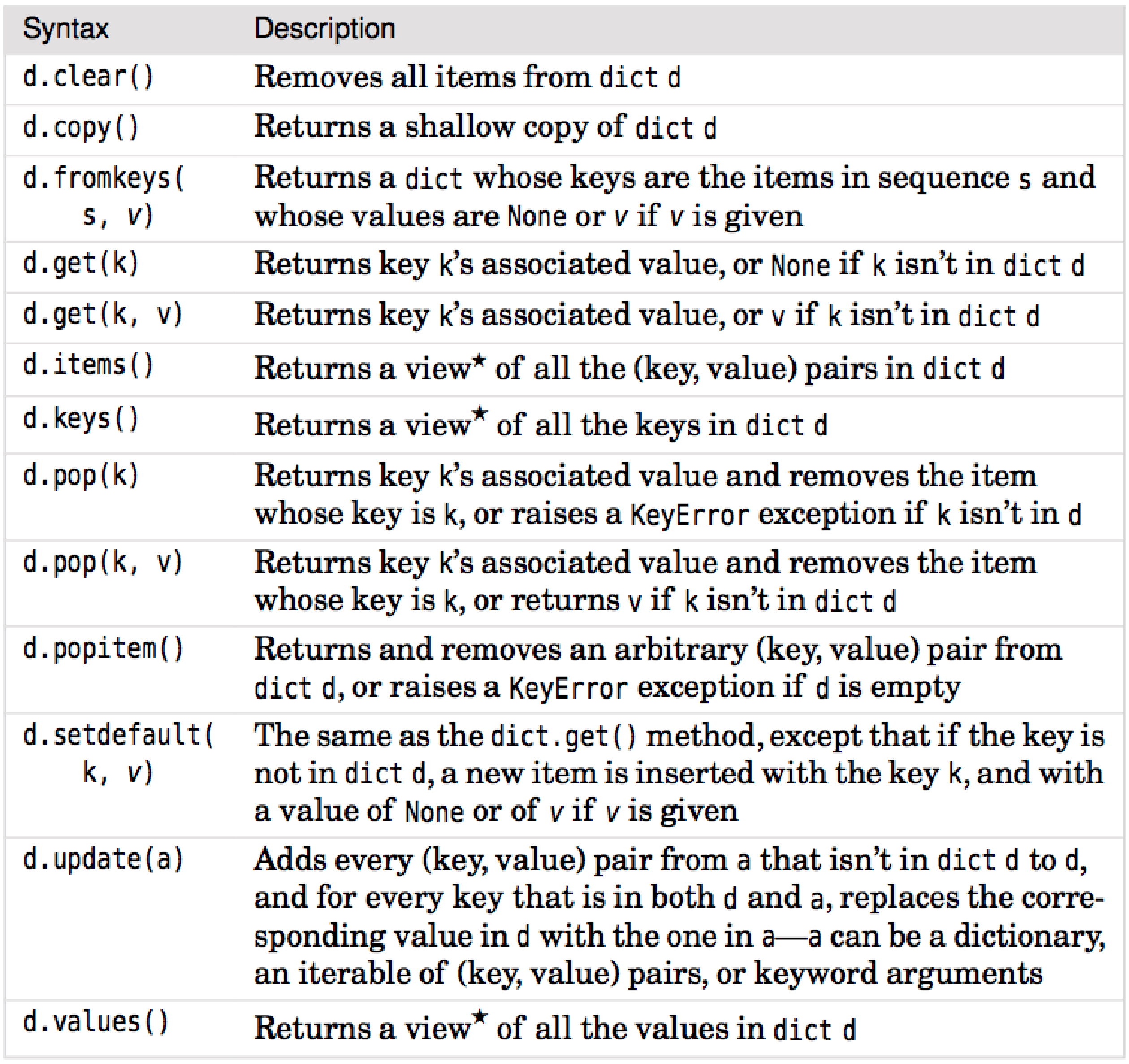
```
for key in d:
    print(key)

#OR
for key in d.keys():
  print(key)
```

- If we want to change the values in a dictionary, the idiom to use is to iterate over the keys and change the values using the brackets operator.

```
    for key in d:
      d[key] += 1
```

Table 3.3 *Dictionary Methods*



Syntax	Description
`d.clear()`	Removes all items from dict `d`
`d.copy()`	Returns a shallow copy of dict `d`
`d.fromkeys(s, v)`	Returns a `dict` whose keys are the items in sequence `s` and whose values are `None` or `v` if `v` is given
`d.get(k)`	Returns key `k`'s associated value, or `None` if `k` isn't in dict `d`
`d.get(k, v)`	Returns key `k`'s associated value, or `v` if `k` isn't in dict `d`
`d.items()`	Returns a view* of all the (key, value) pairs in dict `d`
`d.keys()`	Returns a view* of all the keys in dict `d`
`d.pop(k)`	Returns key `k`'s associated value and removes the item whose key is `k`, or raises a `KeyError` exception if `k` isn't in `d`
`d.pop(k, v)`	Returns key `k`'s associated value and removes the item whose key is `k`, or returns `v` if `k` isn't in dict `d`
`d.popitem()`	Returns and removes an arbitrary (key, value) pair from dict `d`, or raises a `KeyError` exception if `d` is empty
`d.setdefault(k, v)`	The same as the `dict.get()` method, except that if the key is not in dict `d`, a new item is inserted with the key `k`, and with a value of `None` or of `v` if `v` is given
`d.update(a)`	Adds every (key, value) pair from `a` that isn't in dict `d` to `d`, and for every key that is in both `d` and `a`, replaces the corresponding value in `d` with the one in `a`—`a` can be a dictionary, an iterable of (key, value) pairs, or keyword arguments
`d.values()`	Returns a view* of all the values in dict `d`

- The dict.items(), dict.keys(), and dict.values() methods all return dictionary views.

- A dictionary view is effectively a read-only iterable object that appears to hold the dictionary's items or keys or values, depending on the view we have asked for.
- In general, we can simply treat views as iterables.
- However, two things make a view different from a normal iterable.
- One is that if the dictionary the view refers to is changed, the view reflects the change.

- The other is that key and item views support some set-like operations.
- Given dictionary view v and set or dictionary view x, the supported operations are:

```
v & x     # Intersection
v | x     # Union
v - x     # Difference
v ^ x     # Symmetric difference
```

> ★Dictionary views can be thought of—and used as—iterables;

- Can use the membership operator, in, to see whether a particular key is in a dictionary, for example, x in d.
- And we can use the intersection operator to see which keys from a given set are in a dictionary.

```
 d = {}.fromkeys("ABCD", 3) # d == {'A': 3, 'B': 3, 'C': 3, 'D': 3}
s = set("ACX") # s == {'A', 'C', 'X'}
matches = d.keys() & s # matches == {'A', 'C'}
```

- Dictionaries are often used to keep counts of unique items.
- Counting the number of occurrences of each unique word in a file.
- Here is a complete program (uniquewords1.py)

```
import string
import sys
words = {}
strip = string.whitespace + string.punctuation + string.digits + "\"
'"
for filename in sys.argv[1:]:
    for line in open(filename):
        for word in line.lower().split():
            word = word.strip(strip)
            if len(word) > 2:
                words[word] = words.get(word, 0) + 1
for word in sorted(words):
    print("'{0}' occurs {1} times".format(word, words[word]))
```

- We cannot use the syntax words[word] += 1 because this will raise a KeyError exception the first time a new word is encountered—after all, we can’t increment the value of an item that does not yet exist in the dictionary.
- So we use a subtler approach.
- We call dict.get() with a default value of 0.

- If the word is already in the dictionary, dict.get() will return the associated number, and this value plus 1 will be set as the item's new value.
- If the word is not in the dictionary, dict.get() will return the supplied default of 0, and this value plus 1 (i.e., 1) will be set as the value of a new item whose key is the string held by word.

- To clarify, here are two code snippets that do the same thing, although the code using dict.get() is more efficient:

```
words[word] = words.get(word, 0) + 1

# OR
 if word not in words:
    words[word] = 0
 words[word] += 1
```

Reading and Writing Text Files

- Files are opened using the built-in open() function, which returns a "file object" (of type io.TextIOWrapper for text files).
- open() function takes one mandatory argument—the filename, which may include a path—and up to six optional arguments.
- The second argument is the mode—this is used to specify whether the file is to be treated as a text file or as a binary file, and whether the file is to be opened for reading, writing, appending, or a combination of these.

- For text files, Python uses an encoding that is platform-dependent.

```
 fin = open(filename, encoding="utf8") # for reading text
fout = open(filename, "w", encoding="utf8") # for writing text
```

- Because open()'s mode defaults to "read text", and by using a keyword rather than a positional argument for the encoding argument, we can omit the other optional positional arguments when opening for reading.

- Once a file is opened for reading in text mode, we can read the whole file into a single string using the file object's read() method, or into a list of strings using the file object's readlines() method.

- A very common idiom for reading line by line is to treat the file object as an iterator:

```
for line in open(filename, encoding="utf8"):
    process(line)
```

- A file object can be iterated over, just like a sequence, with each successive item being a string containing the next line from the file.

- The lines we get back include the line termination character, \n.

- Specify a mode of “w”, the file is opened in “write text” mode.
- We write to a file using the file object’s write() method, which takes a single string as its argument.
- Each line written should end with a \n.
- Python automatically translates between \n and the underlying platform’s line termination characters when reading and writing.
- Once we have finished using a file object we can call its close() method—this will cause any outstanding writes to be flushed.
- In small Python programs it is very common not to bother calling close(), since Python does this automatically when the file object goes out of scope.

- Using dict.get() allows us to easily update dictionary values, providing the values are single items like numbers or strings.

```
sites = {}
for filename in sys.argv[1:]:
for line in open(filename): i=0
    while True:
        site = None
        i = line.find("http://", i)
        if i > -1:
            i += len("http://")
            for j in range(i, len(line)):
                if not (line[j].isalnum() or line[j] in ".-"):
                    site = line[i:j].lower()
                    break
                if site and "." in site:
                    sites.setdefault(site, set()).add(filename)
                i=j else:
                break
```

- Each line may refer to any number of Web sites.

- The dict.setdefault() method returns an object reference to the item in the dictionary that has the given key (the first argument).
- If there is no such item, the method creates a new item with the key and sets its value either to None, or to the given default value (the second argument).
- The returned object reference always refers to a set (an empty set the first time any particular key, that is, site, is encountered), and we then add the filename that refers to the site to the site’s set of filenames.
- By using a set we ensure that even if a file refers to a site repeatedly, we record the filename only once for the site.

- Dict.setdefault() method’s functionality clear, here are two equivalent code snippets:

```
sites.setdefault(site, set()).add(fname)
```

```
# OR
if site not in sites:
    sites[site] = set()
sites[site].add(fname)
```

```
for site in sorted(sites):
   print("{0} is referred to in:".format(site))
   for filename in sorted(sites[site], key=str.lower):
       print("    {0}".format(filename))
```

- For small dictionaries, we can print their contents using their keys as field names and using mapping unpacking to convert the dictionary’s key–value items into key–value arguments for the str.format() method.

```
 >>> greens = dict(green="#0080000", olive="#808000", lime="#00FF00"
)
>>> print("{green} {olive} {lime}".format(**greens))
#0080000 #808000 #00FF00
```

- Using mapping unpacking (**) has exactly the same effect as writing .format(green=greens.green, olive=greens.olive, lime=greens.lime), but is easier to write and arguably clearer.

- Note that it doesn’t matter if the dictionary has more keys than we need, since only those keys whose names appear in the format string are used.

Dictionary Comprehensions

- A dictionary comprehension is an expression and a loop with an optional condition enclosed in braces, very similar to a set comprehension.

- Like list and set comprehensions, two syntaxes are supported:

```
{keyexpression: valueexpression for key, value in iterable}
{keyexpression: valueexpression for key, value in iterable if condit
ion}
```

- Use a dictionary comprehension to create a dictionary where each key is the name of a file in the current directory and each value is the size of the file in bytes:

```
file_sizes = {name: os.path.getsize(name) for name in os.listdir("."
)}
```

- The os (“operating system”) module’s os.listdir() function returns a list of the files and directories in the path it is passed, although it never includes "." or ".." in the list.

- Can avoid directories and other nonfile entries by adding a condition:

```
  file_sizes = {name: os.path.getsize(name) for name in os.listdir("."
)
  if os.path.isfile(name)}
```

- os.path module's os.path.isfile() function returns True if the path passed to it is that of a file, and False otherwise—that is, for directories, links, and so on.

```
  inverted_d = {v: k for k, v in d.items()}
```

- The resultant dictionary can be inverted back to the original dictionary if all the original dictionary's values are unique—but the inversion will fail with a TypeError being raised if any value is not hashable.

Default Dictionaries

- Default dictionaries are dictionaries—they have all the operators and methods that dictionaries provide.
- What makes default dictionaries different from plain dictionaries is the way they handle missing keys; in all other respects they behave identically to dictionaries.
- defaultdict is a subclass of dict;
- If we use a nonexistent ("missing") key when accessing a dictionary, a KeyError is raised. This is useful because we often want to know whether a key that we expected to be present is absent.
- But in some cases we want every key we use to be present, even if it means that an item with the key is inserted into the dictionary at the time we first access it.
- But if d is a suitably created default dictionary, if an item with key m is in the default dictionary, the corresponding value is returned the same as for a dictionary—but if m is not a key in the default dictionary, a new item with key m is created with a default value, and the newly created item's value is returned.
- When a default dictionary is created, we can pass in a factory function. A factory function is a function that, when called, returns an object of a particular type.
- All of Python's built-in data types can be used as factory functions.
- The factory function passed to a default dictionary is used to create default values for missing keys.
- name of a function is an object reference to the function—so when we want to pass functions as parameters, we just pass the name.
- When we use a function with parentheses, the parentheses tell Python that the function should be called.

- The default dictionary is created:

```
   words = collections.defaultdict(int)
```

- The words default dictionary will never raise a KeyError.

- If we were to write x = words[“xyz”] and there was no item with key “xyz”, when the access is attempted and the key isn’t found, the default dictionary will immediately create a new item with key “xyz” and value 0 (by calling int()), and this value is what will be assigned to x.

```
    words[word] += 1
```

Ordered Dictionaries

- Ordered dictionaries type—collections.OrderedDict—was introduced with Python 3.1 in fulfillment of PEP 372.
- Ordered dictionaries can be used as drop-in replacements for unordered dicts because they provide the same API.
- The difference between the two is that ordered dictionaries store their items in the order in which they were inserted—a feature that can be very convenient.
- Note that if an ordered dictionary is passed an unordered dict or keyword arguments when it is created, the item order will be arbitrary; this is because under the hood Python passes keyword arguments using a standard unordered dict.
- A similar effect occurs with the use of the update() method.
- For these reasons, passing keyword arguments or an unordered dict when creating an ordered dictionary or using update() on one is best avoided.

- However, if we pass a list or tuple of key–value 2-tuples when creating an ordered dictionary, the ordering is preserved (since they are passed as a single item—a list or tuple).

```
    d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
```

```
    tasks = collections.OrderedDict()
   tasks[8031] = "Backup"
   tasks[4027] = "Scan Email"
   tasks[5733] = "Build System"
```

- For ordered dictionaries, we can rely on the keys to be returned in the same order they were inserted.

- list(d.keys()), we are guaranteed to get the list [‘z’, ‘e’, ‘k’], and if we wrote list(tasks.keys()), we are guaranteed to get the list [8031, 4027, 5733].
- feature of ordered dictionaries is that if we change an item’s value—that is, if we insert an item with the same key as an existing key—the order is not changed.
- If we want to move an item to the end, we must delete it and then reinsert it.
- We can also call popitem() to remove and return the last key–value item in the ordered dictionary; or we can call popitem(last=False), in which case the first item will be removed and returned.

Iterating and Copying Collections

Iterators and Iterable Operations and Functions

- An iterable data type is one that can return each of its items one at a time.
- Any object that has an **iter**() method, or any sequence (i.e., an object that has a **getitem**() method taking integer arguments starting from 0) is an iterable and can provide an iterator.
- An iterator is an object that provides a **next**() method which returns each successive item in turn, and raises a StopIteration exception when there are no more items.
- The order in which items are returned depends on the underlying iterable. In the case of lists and tuples, items are normally returned in sequential order starting from the first item (index position 0), but some iterators return the items in an arbitrary order—for example, dictionary and set iterators.
- Built-in iter() function has two quite different behaviors.
- When given a collection data type or a sequence it returns an iterator for the object it is passed—or raises a TypeError if the object cannot be iterated.
- The second iter() behavior occurs when the function is passed a callable (afunctionormethod),and a sentinel value.
- In this case the function passed in is called once at each iteration, returning the function's return value each time, or raising a StopIteration exception if the return value equals the sentinel.
- When we use a for item in iterable loop, Python in effect calls iter(iterable) to get an iterator.
- This iterator's **next**() method is then called at each loop iteration to get the next item, and when the StopIteration exception is raised, it is caught and the loop is terminated.

- Another way to get an iterator's next item is to call the built-in next() function.

```
product = 1
for i in [1, 2, 4, 8]:
    product *= i
print(product)        # prints: 64

# OR
product = 1
i = iter([1, 2, 4, 8])
while True:
    try:
        product *= next(i)
    except StopIteration:
        break
print(product) # prints: 64
```

- Any (finite) iterable, i, can be converted into a tuple by calling tuple(i), or can be converted into a list by calling list(i).

```
  >>> x = [-2, 9, 7, -4, 3]
  >>> all(x), any(x), len(x), min(x), max(x), sum(x) (True, True, 5, -
4, 9, 13)
  >>> x.append(0)
  >>> all(x), any(x), len(x), min(x), max(x), sum(x) (False, True, 6,
-4, 9, 13)
```

- The enumerate() function takes an iterator and returns an enumerator object.
- This object can be treated like an iterator, and at each iteration it returns a 2-tuple with the tuple's first item the iteration number (by default starting from 0), and the second item the next item from the iterator enumerate() was called on.

```
grepword.py Dom data/forenames.txt
data/forenames.txt:615:Dominykas
data/forenames.txt:1435:Dominik
data/forenames.txt:1611:Domhnall
data/forenames.txt:3314:Dominic
```

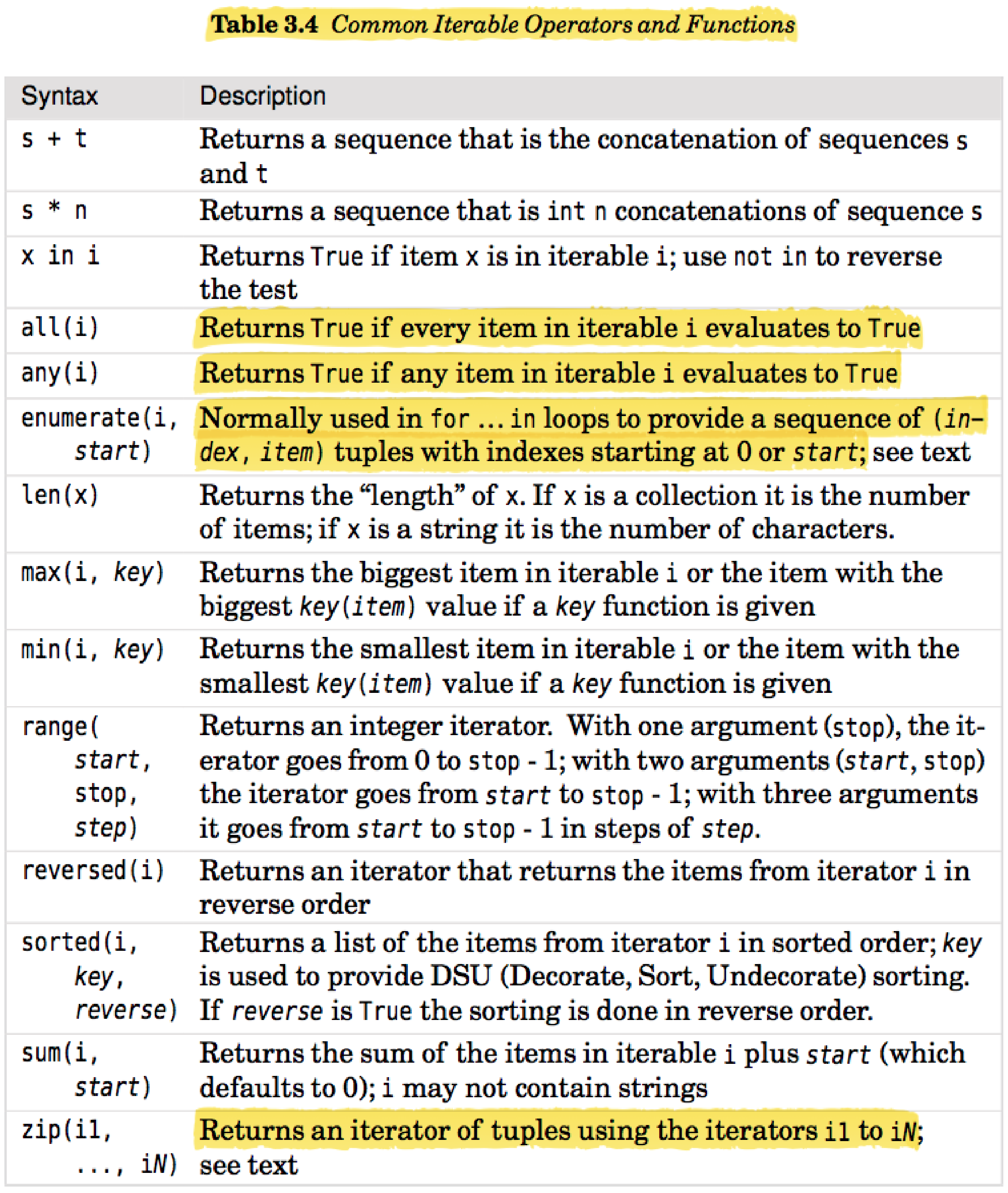


Table 3.4 *Common Iterable Operators and Functions*

Syntax	Description
`s + t`	Returns a sequence that is the concatenation of sequences `s` and `t`
`s * n`	Returns a sequence that is `int` `n` concatenations of sequence `s`
`x in i`	Returns `True` if item `x` is in iterable `i`; use `not in` to reverse the test
`all(i)`	Returns `True` if every item in iterable `i` evaluates to `True`
`any(i)`	Returns `True` if any item in iterable `i` evaluates to `True`
`enumerate(i, start)`	Normally used in `for` … `in` loops to provide a sequence of (*index*, *item*) tuples with indexes starting at 0 or *start*; see text
`len(x)`	Returns the "length" of `x`. If `x` is a collection it is the number of items; if `x` is a string it is the number of characters.
`max(i, key)`	Returns the biggest item in iterable `i` or the item with the biggest *key*(*item*) value if a *key* function is given
`min(i, key)`	Returns the smallest item in iterable `i` or the item with the smallest *key*(*item*) value if a *key* function is given
`range( start, stop, step)`	Returns an integer iterator. With one argument (`stop`), the iterator goes from 0 to `stop` - 1; with two arguments (*start*, `stop`) the iterator goes from *start* to `stop` - 1; with three arguments it goes from *start* to `stop` - 1 in steps of *step*.
`reversed(i)`	Returns an iterator that returns the items from iterator `i` in reverse order
`sorted(i, key, reverse)`	Returns a list of the items from iterator `i` in sorted order; *key* is used to provide DSU (Decorate, Sort, Undecorate) sorting. If *reverse* is `True` the sorting is done in reverse order.
`sum(i, start)`	Returns the sum of the items in iterable `i` plus *start* (which defaults to 0); `i` may not contain strings
`zip(i1, ..., iN)`	Returns an iterator of tuples using the iterators `i1` to `iN`; see text

```
if len(sys.argv) < 3:
print("usage: grepword.py word infile1 [infile2 [... infileN]]")
sys.exit()
word = sys.argv[1]
for filename in sys.argv[2:]:
    for lino, line in enumerate(open(filename), start=1):
        if word in line:
            print("{0}:{1}:{2:.40}".format(filename, lino, line.rstr
ip()))
```

- File object returned by the open() function in text mode can be used as an iterator, returning one line of the file on each iteration.

- By passing the iterator to enumerate(), we get an enumerator iterator that returns the iteration number (in variable lino, “line number”) and a line from the file, on each itera- tion.
- The enumerate() function accepts an optional keyword argument, start, which defaults to 0; we have used this argument set to 1, since by convention, text file line numbers are counted from 1.

- If we need a list or tuple of integers, we can convert the iterator returned by range()

```
 >>> list(range(5)), list(range(9, 14)), tuple(range(10, -11, -5))
([0, 1, 2, 3, 4], [9, 10, 11, 12, 13], (10, 5, 0, -5, -10))
```

- The range() function is most commonly used for two purposes: to create lists or tuples of integers, and to provide loop counting in for ⋯ in loops.

```
for i in range(len(x)):
    x[i] = abs(x[i])

# OR
i=0
while i < len(x):
    x[i] = abs(x[i])
    i += 1
```

- Since we can unpack an iterable using the * operator, we can unpack the iterator returned by the range() function.

- For example, if we have a function called calculate() that takes four arguments, here are some ways we could call it with arguments, 1, 2, 3, and 4:

```
calculate(1, 2, 3, 4)
t = (1, 2, 3, 4)
calculate(*t)
calculate(*range(1, 5))
```

- Inallthreecalls,fourargumentsarepassed. Thesecondcallunpacksa4-tuple, and the third call unpacks the iterator returned by the range() function.

```
 def get_forenames_and_surnames():
    forenames = []
    surnames = []
    for names, filename in ((forenames, "data/forenames.txt"),
                            (surnames, "data/surnames.txt")):
        for name in open(filename, encoding="utf8"):
            names.append(name.rstrip())
        return forenames, surnames
```

- Inside Python programs it is convenient to always use Unix-style paths, since they can be typed without the need for escaping, and they work on all platforms (including Windows).

```
forenames, surnames = get_forenames_and_surnames()
fh = open("test-names1.txt", "w", encoding="utf8")
for i in range(100):
    line = "{0} {1}\n".format(random.choice(forenames),
                              random.choice(surnames))
    fh.write(line)
```

- Another way of combining items from two or more lists (or other iterables) is to use the zip() function.

- The zip() function takes one or more iterables and returns an iterator that returns tuples.

- The first tuple has the first item from every iterable, the second tuple the second item from every iterable, and so on, stopping as soon as one of the iterables is exhausted.

```
>>> for t in zip(range(4), range(0, 10, 2), range(1, 10, 2)):
... print(t)
(0, 0, 1)
(1, 2, 3)
(2, 4, 5)
(3, 6, 7)
```

```
limit = 100
years = list(range(1970, 2013)) * 3
for year, forename, surname in zip(
                    random.sample(years, limit),
                    random.sample(forenames, limit),
                    random.sample(surnames, limit)):
    name = "{0} {1}".format(forename, surname)
    fh.write("{0:.<25}.{1}\n".format(name, year))
```

- We begin by setting a limit on how many names we want to generate.

- random.sample() function that we are using (instead of random.choice()) takes both an iterable and how many items it is to produce—a number that cannot be less than the number of items the iterable can return.
- The random.sample() function returns an iterator that will produce up to the specified number of items from the iterable it is given—with no repeats.
- So this version of the program will always produce unique names.

- The sorted() function returns a list with the items sorted, and the reversed() function simply returns an iterator that iterates in the reverse order to the iterator it is given as its argument.

```
 >>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(reversed(range(6)))
[5, 4, 3, 2, 1, 0]
```

```
>>> x = []
>>> for t in zip(range(-10, 0, 1), range(0, 10, 2), range(1, 10, 2))
:
...
>>> x
[-10, 0, 1, -9, 2, 3, -8, 4, 5, -7, 6, 7, -6, 8, 9]
>>> sorted(x)
[-10, -9, -8, -7, -6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sorted(x, reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -6, -7, -8, -9, -10]
>>> sorted(x, key=abs)
[0, 1, 2, 3, 4, 5, 6, -6, -7, 7, -8, 8, -9, 9, -10]
```

- The += operator extends a list, that is, it appends each item in the sequence it is given to the list.
- Notice that since Python functions are objects like any other, they can be passed as arguments to other functions, and stored in collections.
- When a key function is passed (in this case the abs() function), it is called once for every item in the list (with the item passed as the function's sole parameter), to create a "decorated" list.
- Then the decorated list is sorted, and the sorted list without the decoration is returned as the result.

```
x = sorted(x, key=str.lower)

# OR
temp = []
for item in x:
    temp.append((item.lower(), item))
x = []
for key, value in sorted(temp):
    x.append(value)
```

- Python's sort algorithm is an adaptive stable mergesort that is both fast and smart, and it is especially well optimized for partially sorted lists.
- The "adaptive" part means that the sort algorithm adapts to circumstances—for example, taking advantage of partially sorted data.
- The "stable" part means that items that sort equally are not moved in relation to each other (after all,

there is no need).

- When sorting collections of integers, strings, or other simple types their “less than” operator (<) is used. Python can sort collections that contain collections, working recursively to any depth.

```
 >>> x = list(zip((1, 3, 1, 3), ("pram", "dorie", "kayak", "canoe"))
)
>>> x
[(1, 'pram'), (3, 'dorie'), (1, 'kayak'), (3, 'canoe')]
>>> sorted(x)
[(1, 'kayak'), (1, 'pram'), (3, 'canoe'), (3, 'dorie')]
```

> ★The algorithm was created by Tim Peters. An interesting explanation and discussion of the algorithm is in the file listsort.txt which comes with Python’s source code.

- Python has sorted the list of tuples by comparing the first item of each tuple, and when these are the same, by comparing the second item.

- Can force the sort to be based on the strings and use the integers as tiebreakers by defining a simple key function:

```
 def swap(t):
    return t[1], t[0]
```

- The swap() function takes a 2-tuple and returns a new 2-tuple with the arguments swapped.

```
 >>> sorted(x, key=swap)
[(3, 'canoe'), (3, 'dorie'), (1, 'kayak'), (1, 'pram')]
```

- Lists can also be sorted in-place using the list.sort() method, which takes the same optional arguments as sorted().

- Sorting can be applied only to collections where all the items can be compared with each other:

```
 sorted([3, 8, -7.5, 0, 1.3]) # returns: [-7.5, 0, 1.3, 3, 8]
sorted([3, "spanner", -7.5, 0, 1.3]) # raises a TypeError
```

- But the second list has a string and this cannot be sensibly com- pared with a number, and so a TypeError exception is raised.

- If we want to sort a list that has integers, floating-point numbers, and strings that contain numbers, we can give float() as the key function:

```
 sorted(["1.3", -7.5, "5", 4, "-2.4", 1], key=float)
```

- This returns the list [-7.5, '-2.4', 1, '1.3', 4, '5'].

- Notice that the list's values are not changed, so strings remain strings.
- If any of the strings cannot be converted to a number (e.g., "spanner"), a ValueError exception will be raised.

Copying Collections

- Since Python uses object references, when we use the assignment operator (=),no copying takes place. If the right-hand operand is a literal such as a string or a number, the left-hand operand is set to be an object reference that refers to the in-memory object that holds the literal's value.
- If the right-hand operand is an object reference, the left-hand operand is set to be an object reference that refers to the same object as the right-hand operand.
- One consequence of this is that assignment is very efficient.

- When we assign large collections, such as long lists, the savings are very apparent. Here is an example:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

- Here, a new object reference (beatles) has been created, and both object references refer to the same list—no copying has taken place.

- Since lists are mutable, we can apply a change. For example:

```
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

- We applied the change using the beatles variable—but this is an object refer- ence referring to the same list as songs refers to.

- So any change made through either object reference is visible to the other. This is most often the behavior we want, since copying large collections is potentially expensive.
- Italsomeans, for example, that we can pass a list or other mutable collection data type as an argument to a function, modify the collection in the function, and know that the modified collection will be accessible after the function call has completed.
- However, in some situations, we really do want a separate copy of the collection (or other mutable object).

- For sequences, when we take a slice—for example, songs[:2]—the slice is always an independent copy of the items copied. So to copy an entire sequence we can do this:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

- For dictionaries and sets, copying can be achieved using dict.copy() and set.copy().

- In addition, the copy module provides the copy.copy() function that returns a copy of the object it is given.
- Another way to copy the built-in collection types is to use the type as a function with the collection to be copied as its argument.
- Note, though, that all of these copying techniques are shallow—that is, only object references are copied and not the objects themselves.

- For immutable data types like numbers and strings this has the same effect as copying (except that it is more efficient), but for mutable data types such as nested collections this means that the objects they refer to are referred to both by the original collection and by the copied collection.

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # shallow copy
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

- When list x is shallow-copied, the reference to the nested list [“A”, “B”, “C”] is copied.

- This means that both x and y have as their third item an object reference that refers to this list, so any changes to the nested list are seen by both x and y.

- If we really need independent copies of arbitrarily nested collections, we can deep-copy:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

- Here, lists x and y, and the list items they contain, are completely inde- pendent.

- Note that from now on we will use the terms copy and shallow copy inter changeably — if we mean deep copy, we will say so explicitly.