When slicing from the start of a `list`, you should leave out the zero index to reduce visual noise:

```
assert a[:5] == a[0:5]
```

When slicing to the end of a `list`, you should leave out the final index because it's redundant:

```
assert a[5:] == a[5:len(a)]
```

Using negative numbers for slicing is helpful for doing offsets relative to the end of a `list`. All of these forms of slicing would be clear to a new reader of your code:

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #                     ['e', 'f', 'g', 'h']
a[-3:]    #                          ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]   #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #                          ['f', 'g']
```

There are no surprises here, and I encourage you to use these variations.

Slicing deals properly with start and end indexes that are beyond the boundaries of a `list` by silently omitting missing items. This behavior makes it easy for your code to establish a maximum length to consider for an input sequence:

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:]
```

In contrast, accessing the same index directly causes an exception:

```
a[20]
```

```
>>>
Traceback ...
IndexError: list index out of range
```

Note

Beware that indexing a `list` by a negated variable is one of the few situations in which you can get surprising results from slicing. For example, the expression `somelist[-n:]` will work fine when n is greater than one (e.g., `somelist[-3:]`). However, when n is zero, the expression `somelist[-0:]` is equivalent to `somelist[:]` and will result in a copy of the original `list`.

The result of slicing a `list` is a whole new `list`. References to the objects from the original `list` are maintained. Modifying the result of slicing won't affect the original `list`:

```
b = a[3:]
print('Before:   ', b)
b[1] = 99
print('After:    ', b)
print('No change:', a)
```

```
>>>
Before:    ['d', 'e', 'f', 'g', 'h']
After:     ['d', 99, 'f', 'g', 'h']
No change: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

When used in assignments, slices replace the specified range in the original `list`. Unlike unpacking assignments (such as `a, b = c[:2]`; see Item 6: "Prefer Multiple Assignment Unpacking Over Indexing"), the lengths of slice assignments don't need to be the same. The values before and after the assigned slice will be preserved. Here, the `list` shrinks because the replacement `list` is shorter than the specified slice:

```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After  ', a)
```

```
>>>
Before  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After   ['a', 'b', 99, 22, 14, 'h']
```

And here the `list` grows because the assigned `list` is longer than the specific slice:

```
print('Before ', a)
a[2:3] = [47, 11]
print('After  ', a)
```

```
>>>
Before  ['a', 'b', 99, 22, 14, 'h']
After   ['a', 'b', 47, 11, 22, 14, 'h']
```

If you leave out both the start and the end indexes when slicing, you end up with a copy of the original `list`:

```
b = a[:]
assert b == a and b is not a
```

If you assign to a slice with no start or end indexes, you replace the entire contents of the list with a copy of what's referenced (instead of allocating a new list):

```
b = a
print('Before a', a)
print('Before b', b)
a[:] = [101, 102, 103]
assert a is b             # Still the same list object
print('After a ', a)      # Now has different contents
print('After b ', b)      # Same list, so same contents as a
```

```
>>>
Before a ['a', 'b', 47, 11, 22, 14, 'h']
Before b ['a', 'b', 47, 11, 22, 14, 'h']
After a  [101, 102, 103]
After b  [101, 102, 103]
```

Things to Remember

- ✦ Avoid being verbose when slicing: Don't supply 0 for the start index or the length of the sequence for the end index.
- ✦ Slicing is forgiving of start or end indexes that are out of bounds, which means it's easy to express slices on the front or back boundaries of a sequence (like a[:20] or a[-20:]).
- ✦ Assigning to a list slice replaces that range in the original sequence with what's referenced even if the lengths are different.

Item 12: Avoid Striding and Slicing in a Single Expression

In addition to basic slicing (see Item 11: "Know How to Slice Sequences"), Python has special syntax for the stride of a slice in the form somelist[start:end:stride]. This lets you take every nth item when slicing a sequence. For example, the stride makes it easy to group by even and odd indexes in a list:

```
x = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = x[::2]
evens = x[1::2]
print(odds)
print(evens)
```

```
>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

The problem is that the stride syntax often causes unexpected behavior that can introduce bugs. For example, a common Python trick for reversing a byte string is to slice the string with a stride of -1:

```
x = b'mongoose'
y = x[::-1]
print(y)

>>>
b'esoognom'
```

This also works correctly for Unicode strings (see Item 3: "Know the Differences Between bytes and str"):

```
x = '寿司'
y = x[::-1]
print(y)

>>>
司寿
```

But it will break when Unicode data is encoded as a UTF-8 byte string:

```
w = '寿司'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')

>>>
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb8 in
position 0: invalid start byte
```

Are negative strides besides -1 useful? Consider the following examples:

```
x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
x[::2]   # ['a', 'c', 'e', 'g']
x[::-2]  # ['h', 'f', 'd', 'b']
```

Here, ::2 means "Select every second item starting at the beginning." Trickier, ::-2 means "Select every second item starting at the end and moving backward."

What do you think 2::2 means? What about -2::-2 vs. -2:2:-2 vs. 2:2:-2?

```
x[2::2]     # ['c', 'e', 'g']
x[-2::-2]   # ['g', 'e', 'c', 'a']
x[-2:2:-2]  # ['g', 'e']
x[2:2:-2]   # []
```

The point is that the stride part of the slicing syntax can be extremely confusing. Having three numbers within the brackets is hard enough to read because of its density. Then, it's not obvious when the start and end indexes come into effect relative to the stride value, especially when the stride is negative.

To prevent problems, I suggest you avoid using a stride along with start and end indexes. If you must use a stride, prefer making it a positive value and omit start and end indexes. If you must use a stride with start or end indexes, consider using one assignment for striding and another for slicing:

```
y = x[::2]   # ['a', 'c', 'e', 'g']
z = y[1:-1]  # ['c', 'e']
```

Striding and then slicing creates an extra shallow copy of the data. The first operation should try to reduce the size of the resulting slice by as much as possible. If your program can't afford the time or memory required for two steps, consider using the `itertools` built-in module's `islice` method (see Item 36: "Consider `itertools` for Working with Iterators and Generators"), which is clearer to read and doesn't permit negative values for start, end, or stride.

Things to Remember

- ✦ Specifying start, end, and stride in a slice can be extremely confusing.
- ✦ Prefer using positive stride values in slices without start or end indexes. Avoid negative stride values if possible.
- ✦ Avoid using start, end, and stride together in a single slice. If you need all three parameters, consider doing two assignments (one to stride and another to slice) or using `islice` from the `itertools` built-in module.

Item 13: Prefer Catch-All Unpacking Over Slicing

One limitation of basic unpacking (see Item 6: "Prefer Multiple Assignment Unpacking Over Indexing") is that you must know the length of the sequences you're unpacking in advance. For example, here I have a `list` of the ages of cars that are being traded in at a dealership. When I try to take the first two items of the `list` with basic unpacking, an exception is raised at runtime:

```
car_ages = [0, 9, 4, 8, 7, 20, 19, 1, 6, 15]
car_ages_descending = sorted(car_ages, reverse=True)
oldest, second_oldest = car_ages_descending
```

```
>>>
Traceback ...
ValueError: too many values to unpack (expected 2)
```

Newcomers to Python often rely on indexing and slicing (see Item 11: "Know How to Slice Sequences") for this situation. For example, here I extract the oldest, second oldest, and other car ages from a `list` of at least two items:

```
oldest = car_ages_descending[0]
second_oldest = car_ages_descending[1]
others = car_ages_descending[2:]
print(oldest, second_oldest, others)
```

```
>>>
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

This works, but all of the indexing and slicing is visually noisy. In practice, it's also error prone to divide the members of a sequence into various subsets this way because you're much more likely to make off-by-one errors; for example, you might change boundaries on one line and forget to update the others.

To better handle this situation, Python also supports catch-all unpacking through a *starred expression*. This syntax allows one part of the unpacking assignment to receive all values that didn't match any other part of the unpacking pattern. Here, I use a starred expression to achieve the same result as above without indexing or slicing:

```
oldest, second_oldest, *others = car_ages_descending
print(oldest, second_oldest, others)
```

```
>>>
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

This code is shorter, easier to read, and no longer has the error-prone brittleness of boundary indexes that must be kept in sync between lines.

A starred expression may appear in any position, so you can get the benefits of catch-all unpacking anytime you need to extract one slice:

```
oldest, *others, youngest = car_ages_descending
print(oldest, youngest, others)

*others, second_youngest, youngest = car_ages_descending
print(youngest, second_youngest, others)
```

```
>>>
20 0 [19, 15, 9, 8, 7, 6, 4, 1]
0 1 [20, 19, 15, 9, 8, 7, 6, 4]
```

However, to unpack assignments that contain a starred expression, you must have at least one required part, or else you'll get a SyntaxError. You can't use a catch-all expression on its own:

```
*others = car_ages_descending

>>>
Traceback ...
SyntaxError: starred assignment target must be in a list or
➥tuple
```

You also can't use multiple catch-all expressions in a single-level unpacking pattern:

```
first, *middle, *second_middle, last = [1, 2, 3, 4]

>>>
Traceback ...
SyntaxError: two starred expressions in assignment
```

But it is possible to use multiple starred expressions in an unpacking assignment statement, as long as they're catch-alls for different parts of the multilevel structure being unpacked. I don't recommend doing the following (see Item 19: "Never Unpack More Than Three Variables When Functions Return Multiple Values" for related guidance), but understanding it should help you develop an intuition for how starred expressions can be used in unpacking assignments:

```
car_inventory = {
    'Downtown': ('Silver Shadow', 'Pinto', 'DMC'),
    'Airport': ('Skyline', 'Viper', 'Gremlin', 'Nova'),
}

((loc1, (best1, *rest1)),
 (loc2, (best2, *rest2))) = car_inventory.items()
print(f'Best at {loc1} is {best1}, {len(rest1)} others')
print(f'Best at {loc2} is {best2}, {len(rest2)} others')

>>>
Best at Downtown is Silver Shadow, 2 others
Best at Airport is Skyline, 3 others
```

Starred expressions become list instances in all cases. If there are no leftover items from the sequence being unpacked, the catch-all part will be an empty list. This is especially useful when you're processing a sequence that you know in advance has at least *N* elements:

```
short_list = [1, 2]
first, second, *rest = short_list
print(first, second, rest)
```

```
>>>
1 2 []
```

You can also unpack arbitrary iterators with the unpacking syntax. This isn't worth much with a basic multiple-assignment statement. For example, here I unpack the values from iterating over a `range` of length 2. This doesn't seem useful because it would be easier to just assign to a static `list` that matches the unpacking pattern (e.g., `[1, 2]`):

```
it = iter(range(1, 3))
first, second = it
print(f'{first} and {second}')
```

```
>>>
1 and 2
```

But with the addition of starred expressions, the value of unpacking iterators becomes clear. For example, here I have a generator that yields the rows of a CSV file containing all car orders from the dealership this week:

```
def generate_csv():
    yield ('Date', 'Make' , 'Model', 'Year', 'Price')
    ...
```

Processing the results of this generator using indexes and slices is fine, but it requires multiple lines and is visually noisy:

```
all_csv_rows = list(generate_csv())
header = all_csv_rows[0]
rows = all_csv_rows[1:]
print('CSV Header:', header)
print('Row count: ', len(rows))
```

```
>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count:  200
```

Unpacking with a starred expression makes it easy to process the first row—the header—separately from the rest of the iterator's contents. This is much clearer:

```
it = generate_csv()
header, *rows = it
print('CSV Header:', header)
print('Row count: ', len(rows))
```

```
>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count:  200
```

Keep in mind, however, that because a starred expression is always turned into a list, unpacking an iterator also risks the potential of using up all of the memory on your computer and causing your program to crash. So you should only use catch-all unpacking on iterators when you have good reason to believe that the result data will all fit in memory (see Item 31: "Be Defensive When Iterating Over Arguments" for another approach).

Things to Remember

- ✦ Unpacking assignments may use a starred expression to catch all values that weren't assigned to the other parts of the unpacking pattern into a list.
- ✦ Starred expressions may appear in any position, and they will always become a list containing the zero or more values they receive.
- ✦ When dividing a list into non-overlapping pieces, catch-all unpacking is much less error prone than slicing and indexing.

Item 14: Sort by Complex Criteria Using the key Parameter

The list built-in type provides a sort method for ordering the items in a list instance based on a variety of criteria. By default, sort will order a list's contents by the natural ascending order of the items. For example, here I sort a list of integers from smallest to largest:

```
numbers = [93, 86, 11, 68, 70]
numbers.sort()
print(numbers)
```

```
>>>
[11, 68, 70, 86, 93]
```

The sort method works for nearly all built-in types (strings, floats, etc.) that have a natural ordering to them. What does sort do with objects? For example, here I define a class—including a __repr__ method so instances are printable; see Item 75: "Use repr Strings for Debugging Output"—to represent various tools you may need to use on a construction site:

```
class Tool:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def __repr__(self):
        return f'Tool({self.name!r}, {self.weight})'
```

```
tools = [
    Tool('level', 3.5),
    Tool('hammer', 1.25),
    Tool('screwdriver', 0.5),
    Tool('chisel', 0.25),
]
```

Sorting objects of this type doesn't work because the `sort` method tries to call comparison special methods that aren't defined by the class:

```
tools.sort()
```

```
>>>
Traceback ...
TypeError: '<' not supported between instances of 'Tool' and
'Tool'
```

If your class should have a natural ordering like integers do, then you can define the necessary special methods (see Item 73: "Know How to Use `heapq` for Priority Queues" for an example) to make `sort` work without extra parameters. But the more common case is that your objects may need to support multiple orderings, in which case defining a natural ordering really doesn't make sense.

Often there's an attribute on the object that you'd like to use for sorting. To support this use case, the `sort` method accepts a `key` parameter that's expected to be a function. The `key` function is passed a single argument, which is an item from the `list` that is being sorted. The return value of the `key` function should be a comparable value (i.e., with a natural ordering) to use in place of an item for sorting purposes.

Here, I use the `lambda` keyword to define a function for the `key` parameter that enables me to sort the `list` of `Tool` objects alphabetically by their `name`:

```
print('Unsorted:', repr(tools))
tools.sort(key=lambda x: x.name)
print('\nSorted:  ', tools)
```

```
>>>
Unsorted: [Tool('level',       3.5),
           Tool('hammer',      1.25),
           Tool('screwdriver', 0.5),
           Tool('chisel',      0.25)]
```

```
Sorted:   [Tool('chisel',      0.25),
           Tool('hammer',      1.25),
           Tool('level',       3.5),
           Tool('screwdriver', 0.5)]
```

I can just as easily define another lambda function to sort by `weight` and pass it as the `key` parameter to the `sort` method:

```
tools.sort(key=lambda x: x.weight)
print('By weight:', tools)

>>>
By weight: [Tool('chisel',      0.25),
            Tool('screwdriver', 0.5),
            Tool('hammer',      1.25),
            Tool('level',       3.5)]
```

Within the lambda function passed as the `key` parameter you can access attributes of items as I've done here, index into items (for sequences, tuples, and dictionaries), or use any other valid expression.

For basic types like strings, you may even want to use the `key` function to do transformations on the values before sorting. For example, here I apply the `lower` method to each item in a `list` of place names to ensure that they're in alphabetical order, ignoring any capitalization (since in the natural lexical ordering of strings, capital letters come before lowercase letters):

```
places = ['home', 'work', 'New York', 'Paris']
places.sort()
print('Case sensitive:  ', places)
places.sort(key=lambda x: x.lower())
print('Case insensitive:', places)

>>>
Case sensitive:   ['New York', 'Paris',    'home',   'work']
Case insensitive: ['home',     'New York', 'Paris', 'work']
```

Sometimes you may need to use multiple criteria for sorting. For example, say that I have a `list` of power tools and I want to sort them first by `weight` and then by `name`. How can I accomplish this?

```
power_tools = [
    Tool('drill', 4),
    Tool('circular saw', 5),
    Tool('jackhammer', 40),
    Tool('sander', 4),
]
```

The simplest solution in Python is to use the tuple type. Tuples are immutable sequences of arbitrary Python values. Tuples are comparable by default and have a natural ordering, meaning that they implement all of the special methods, such as __lt__, that are required by the sort method. Tuples implement these special method comparators by iterating over each position in the tuple and comparing the corresponding values one index at a time. Here, I show how this works when one tool is heavier than another:

```
saw = (5, 'circular saw')
jackhammer = (40, 'jackhammer')
assert not (jackhammer < saw)  # Matches expectations
```

If the first position in the tuples being compared are equal—weight in this case—then the tuple comparison will move on to the second position, and so on:

```
drill = (4, 'drill')
sander = (4, 'sander')
assert drill[0] == sander[0]  # Same weight
assert drill[1] < sander[1]   # Alphabetically less
assert drill < sander         # Thus, drill comes first
```

You can take advantage of this tuple comparison behavior in order to sort the list of power tools first by weight and then by name. Here, I define a key function that returns a tuple containing the two attributes that I want to sort on in order of priority:

```
power_tools.sort(key=lambda x: (x.weight, x.name))
print(power_tools)

>>>
[Tool('drill',        4),
 Tool('sander',       4),
 Tool('circular saw', 5),
 Tool('jackhammer',   40)]
```

One limitation of having the key function return a tuple is that the direction of sorting for all criteria must be the same (either all in ascending order, or all in descending order). If I provide the reverse parameter to the sort method, it will affect both criteria in the tuple the same way (note how 'sander' now comes before 'drill' instead of after):

```
power_tools.sort(key=lambda x: (x.weight, x.name),
                 reverse=True)  # Makes all criteria descending
print(power_tools)
```

```
>>>
[Tool('jackhammer',   40),
 Tool('circular saw', 5),
 Tool('sander',       4),
 Tool('drill',        4)]
```

For numerical values it's possible to mix sorting directions by using the unary minus operator in the `key` function. This negates one of the values in the returned `tuple`, effectively reversing its sort order while leaving the others intact. Here, I use this approach to sort by `weight` descending, and then by `name` ascending (note how `'sander'` now comes after `'drill'` instead of before):

```
power_tools.sort(key=lambda x: (-x.weight, x.name))
print(power_tools)
```

```
>>>
[Tool('jackhammer',   40),
 Tool('circular saw', 5),
 Tool('drill',        4),
 Tool('sander',       4)]
```

Unfortunately, unary negation isn't possible for all types. Here, I try to achieve the same outcome by using the `reverse` argument to sort by `weight` descending and then negating `name` to put it in ascending order:

```
power_tools.sort(key=lambda x: (x.weight, -x.name),
                 reverse=True)
```

```
>>>
Traceback ...
TypeError: bad operand type for unary -: 'str'
```

For situations like this, Python provides a *stable* sorting algorithm. The `sort` method of the `list` type will preserve the order of the input `list` when the `key` function returns values that are equal to each other. This means that I can call `sort` multiple times on the same `list` to combine different criteria together. Here, I produce the same sort ordering of `weight` descending and `name` ascending as I did above but by using two separate calls to `sort`:

```
power_tools.sort(key=lambda x: x.name)   # Name ascending

power_tools.sort(key=lambda x: x.weight, # Weight descending
                 reverse=True)

print(power_tools)
```

```
>>>
[Tool('jackhammer',   40),
 Tool('circular saw', 5),
 Tool('drill',        4),
 Tool('sander',       4)]
```

To understand why this works, note how the first call to `sort` puts the names in alphabetical order:

```
power_tools.sort(key=lambda x: x.name)
print(power_tools)
```

```
>>>
[Tool('circular saw', 5),
 Tool('drill',        4),
 Tool('jackhammer',   40),
 Tool('sander',       4)]
```

When the second `sort` call by `weight` descending is made, it sees that both `'sander'` and `'drill'` have a weight of `4`. This causes the `sort` method to put both items into the final result `list` in the same order that they appeared in the original `list`, thus preserving their relative ordering by `name` ascending:

```
power_tools.sort(key=lambda x: x.weight,
                 reverse=True)
print(power_tools)
```

```
>>>
[Tool('jackhammer',   40),
 Tool('circular saw', 5),
 Tool('drill',        4),
 Tool('sander',       4)]
```

This same approach can be used to combine as many different types of sorting criteria as you'd like in any direction, respectively. You just need to make sure that you execute the sorts in the opposite sequence of what you want the final `list` to contain. In this example, I wanted the sort order to be by `weight` descending and then by `name` ascending, so I had to do the `name` sort first, followed by the `weight` sort.

That said, the approach of having the `key` function return a `tuple`, and using unary negation to mix sort orders, is simpler to read and requires less code. I recommend only using multiple calls to `sort` if it's absolutely necessary.

Things to Remember

- ✦ The sort method of the list type can be used to rearrange a list's contents by the natural ordering of built-in types like strings, integers, tuples, and so on.
- ✦ The sort method doesn't work for objects unless they define a natural ordering using special methods, which is uncommon.
- ✦ The key parameter of the sort method can be used to supply a helper function that returns the value to use for sorting in place of each item from the list.
- ✦ Returning a tuple from the key function allows you to combine multiple sorting criteria together. The unary minus operator can be used to reverse individual sort orders for types that allow it.
- ✦ For types that can't be negated, you can combine many sorting criteria together by calling the sort method multiple times using different key functions and reverse values, in the order of lowest rank sort call to highest rank sort call.

Item 15: Be Cautious When Relying on dict Insertion Ordering

In Python 3.5 and before, iterating over a dict would return keys in arbitrary order. The order of iteration would not match the order in which the items were inserted. For example, here I create a dictionary mapping animal names to their corresponding baby names and then print it out (see Item 75: "Use repr Strings for Debugging Output" for how this works):

```
# Python 3.5
baby_names = {
    'cat': 'kitten',
    'dog': 'puppy',
}
print(baby_names)
```

```
>>>
{'dog': 'puppy', 'cat': 'kitten'}
```

When I created the dictionary the keys were in the order 'cat', 'dog', but when I printed it the keys were in the reverse order 'dog', 'cat'. This behavior is surprising, makes it harder to reproduce test cases, increases the difficulty of debugging, and is especially confusing to newcomers to Python.

This happened because the dictionary type previously implemented its hash table algorithm with a combination of the hash built-in function and a random seed that was assigned when the Python interpreter started. Together, these behaviors caused dictionary orderings to not match insertion order and to randomly shuffle between program executions.

Starting with Python 3.6, and officially part of the Python specification in version 3.7, dictionaries will preserve insertion order. Now, this code will always print the dictionary in the same way it was originally created by the programmer:

```
baby_names = {
    'cat': 'kitten',
    'dog': 'puppy',
}
print(baby_names)
```

```
>>>
{'cat': 'kitten', 'dog': 'puppy'}
```

With Python 3.5 and earlier, all methods provided by dict that relied on iteration order, including keys, values, items, and popitem, would similarly demonstrate this random-looking behavior:

```
# Python 3.5
print(list(baby_names.keys()))
print(list(baby_names.values()))
print(list(baby_names.items()))
print(baby_names.popitem())  # Randomly chooses an item
```

```
>>>
['dog', 'cat']
['puppy', 'kitten']
[('dog', 'puppy'), ('cat', 'kitten')]
('dog', 'puppy')
```

These methods now provide consistent insertion ordering that you can rely on when you write your programs:

```
print(list(baby_names.keys()))
print(list(baby_names.values()))
print(list(baby_names.items()))
print(baby_names.popitem())  # Last item inserted
```

```
>>>
['cat', 'dog']
['kitten', 'puppy']
[('cat', 'kitten'), ('dog', 'puppy')]
('dog', 'puppy')
```

There are many repercussions of this change on other Python features that are dependent on the dict type and its specific implementation.

Keyword arguments to functions—including the **kwargs catch-all parameter; see Item 23: "Provide Optional Behavior with Keyword Arguments"—previously would come through in seemingly random order, which can make it harder to debug function calls:

```
# Python 3.5
def my_func(**kwargs):
    for key, value in kwargs.items():
        print('%s = %s' % (key, value))

my_func(goose='gosling', kangaroo='joey')
```

```
>>>
kangaroo = joey
goose = gosling
```

Now, the order of keyword arguments is always preserved to match how the programmer originally called the function:

```
def my_func(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')

my_func(goose='gosling', kangaroo='joey')
```

```
>>>
goose = gosling
kangaroo = joey
```

Classes also use the dict type for their instance dictionaries. In previous versions of Python, object fields would show the randomizing behavior:

```
# Python 3.5
class MyClass:
    def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'

a = MyClass()
for key, value in a.__dict__.items():
    print('%s = %s' % (key, value))
```

```
>>>
elephant = calf
alligator = hatchling
```

Again, you can now assume that the order of assignment for these instance fields will be reflected in `__dict__`:

```
class MyClass:
    def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'

a = MyClass()
for key, value in a.__dict__.items():
    print(f'{key} = {value}')
```

```
>>>
alligator = hatchling
elephant = calf
```

The way that dictionaries preserve insertion ordering is now part of the Python language specification. For the language features above, you can rely on this behavior and even make it part of the APIs you design for your classes and functions.

> **Note**
>
> For a long time the `collections` built-in module has had an `OrderedDict` class that preserves insertion ordering. Although this class's behavior is similar to that of the standard `dict` type (since Python 3.7), the performance characteristics of `OrderedDict` are quite different. If you need to handle a high rate of key insertions and `popitem` calls (e.g., to implement a least-recently-used cache), `OrderedDict` may be a better fit than the standard Python `dict` type (see Item 70: "Profile Before Optimizing" on how to make sure you need this).

However, you shouldn't always assume that insertion ordering behavior will be present when you're handling dictionaries. Python makes it easy for programmers to define their own custom container types that emulate the standard *protocols* matching `list`, `dict`, and other types (see Item 43: "Inherit from `collections.abc` for Custom Container Types"). Python is not statically typed, so most code relies on *duck typing*—where an object's behavior is its de facto type—instead of rigid class hierarchies. This can result in surprising gotchas.

For example, say that I'm writing a program to show the results of a contest for the cutest baby animal. Here, I start with a dictionary containing the total vote count for each one:

```
votes = {
    'otter': 1281,
    'polar bear': 587,
    'fox': 863,
}
```

I define a function to process this voting data and save the rank of each animal name into a provided empty dictionary. In this case, the dictionary could be the data model that powers a UI element:

```
def populate_ranks(votes, ranks):
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)
    for i, name in enumerate(names, 1):
        ranks[name] = i
```

I also need a function that will tell me which animal won the contest. This function works by assuming that `populate_ranks` will assign the contents of the `ranks` dictionary in ascending order, meaning that the first key must be the winner:

```
def get_winner(ranks):
    return next(iter(ranks))
```

Here, I can confirm that these functions work as designed and deliver the result that I expected:

```
ranks = {}
populate_ranks(votes, ranks)
print(ranks)
winner = get_winner(ranks)
print(winner)
```

```
>>>
{'otter': 1, 'fox': 2, 'polar bear': 3}
otter
```

Now, imagine that the requirements of this program have changed. The UI element that shows the results should be in alphabetical order instead of rank order. To accomplish this, I can use the `collections.abc` built-in module to define a new dictionary-like class that iterates its contents in alphabetical order:

```
from collections.abc import MutableMapping


class SortedDict(MutableMapping):
    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        return self.data[key]

    def __setitem__(self, key, value):
        self.data[key] = value
    def __delitem__(self, key):
        del self.data[key]

    def __iter__(self):
        keys = list(self.data.keys())
        keys.sort()
        for key in keys:
            yield key

    def __len__(self):
        return len(self.data)
```

I can use a SortedDict instance in place of a standard dict with the functions from before and no errors will be raised since this class conforms to the protocol of a standard dictionary. However, the result is incorrect:

```
sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)
```

```
>>>
{'otter': 1, 'fox': 2, 'polar bear': 3}
fox
```

The problem here is that the implementation of get_winner assumes that the dictionary's iteration is in insertion order to match populate_ranks. This code is using SortedDict instead of dict, so that assumption is no longer true. Thus, the value returned for the winner is 'fox', which is alphabetically first.

There are three ways to mitigate this problem. First, I can reimplement the get_winner function to no longer assume that the ranks dictionary has a specific iteration order. This is the most conservative and robust solution:

```
def get_winner(ranks):
    for name, rank in ranks.items():
        if rank == 1:
            return name

winner = get_winner(sorted_ranks)
print(winner)
```

```
>>>
otter
```

The second approach is to add an explicit check to the top of the function to ensure that the type of ranks matches my expectations, and to raise an exception if not. This solution likely has better runtime performance than the more conservative approach:

```
def get_winner(ranks):
    if not isinstance(ranks, dict):
        raise TypeError('must provide a dict instance')
    return next(iter(ranks))

get_winner(sorted_ranks)
```

```
>>>
Traceback ...
TypeError: must provide a dict instance
```

The third alternative is to use type annotations to enforce that the value passed to get_winner is a dict instance and not a MutableMapping with dictionary-like behavior (see Item 90: "Consider Static Analysis via typing to Obviate Bugs"). Here, I run the mypy tool in strict mode on an annotated version of the code above:

```
from typing import Dict, MutableMapping


def populate_ranks(votes: Dict[str, int],
                   ranks: Dict[str, int]) -> None:
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)
    for i, name in enumerate(names, 1):
        ranks[name] = i

def get_winner(ranks: Dict[str, int]) -> str:
    return next(iter(ranks))

class SortedDict(MutableMapping[str, int]):
    ...

votes = {
    'otter': 1281,
    'polar bear': 587,
    'fox': 863,
}

sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)
```

```
$ python3 -m mypy --strict example.py
.../example.py:48: error: Argument 2 to "populate_ranks" has
➥incompatible type "SortedDict"; expected "Dict[str, int]"
.../example.py:50: error: Argument 1 to "get_winner" has
➥incompatible type "SortedDict"; expected "Dict[str, int]"
```

This correctly detects the mismatch between the `dict` and `MutableMapping` types and flags the incorrect usage as an error. This solution provides the best mix of static type safety and runtime performance.

Things to Remember

✦ Since Python 3.7, you can rely on the fact that iterating a `dict` instance's contents will occur in the same order in which the keys were initially added.

✦ Python makes it easy to define objects that act like dictionaries but that aren't `dict` instances. For these types, you can't assume that insertion ordering will be preserved.

✦ There are three ways to be careful about dictionary-like classes: Write code that doesn't rely on insertion ordering, explicitly check for the `dict` type at runtime, or require `dict` values using type annotations and static analysis.

Item 16: Prefer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys

The three fundamental operations for interacting with dictionaries are accessing, assigning, and deleting keys and their associated values. The contents of dictionaries are dynamic, and thus it's entirely possible—even likely—that when you try to access or delete a key, it won't already be present.

For example, say that I'm trying to determine people's favorite type of bread to devise the menu for a sandwich shop. Here, I define a dictionary of counters with the current votes for each style:

```
counters = {
    'pumpernickel': 2,
    'sourdough': 1,
}
```

To increment the counter for a new vote, I need to see if the key exists, insert the key with a default counter value of zero if it's missing, and then increment the counter's value. This requires accessing the key two times and assigning it once. Here, I accomplish this task using an `if` statement with an `in` expression that returns `True` when the key is present:

```
key = 'wheat'

if key in counters:
    count = counters[key]
else:
    count = 0

counters[key] = count + 1
```

Another way to accomplish the same behavior is by relying on how dictionaries raise a `KeyError` exception when you try to get the value for a key that doesn't exist. This approach is more efficient because it requires only one access and one assignment:

```
try:
    count = counters[key]
except KeyError:
    count = 0

counters[key] = count + 1
```

This flow of fetching a key that exists or returning a default value is so common that the `dict` built-in type provides the `get` method to accomplish this task. The second parameter to `get` is the default value to return in the case that the key—the first parameter—isn't present. This also requires only one access and one assignment, but it's much shorter than the `KeyError` example:

```
count = counters.get(key, 0)
counters[key] = count + 1
```

It's possible to shorten the `in` expression and `KeyError` approaches in various ways, but all of these alternatives suffer from requiring code duplication for the assignments, which makes them less readable and worth avoiding:

```
if key not in counters:
    counters[key] = 0
counters[key] += 1

if key in counters:
    counters[key] += 1
else:
    counters[key] = 1
```

```
try:
    counters[key] += 1
except KeyError:
    counters[key] = 1
```

Thus, for a dictionary with simple types, using the `get` method is the shortest and clearest option.

> Note
>
> If you're maintaining dictionaries of counters like this, it's worth considering the `Counter` class from the `collections` built-in module, which provides most of the facilities you are likely to need.

What if the values of the dictionary are a more complex type, like a `list`? For example, say that instead of only counting votes, I also want to know who voted for each type of bread. Here, I do this by associating a `list` of names with each key:

```
votes = {
    'baguette': ['Bob', 'Alice'],
    'ciabatta': ['Coco', 'Deb'],
}
key = 'brioche'
who = 'Elmer'

if key in votes:
    names = votes[key]
else:
    votes[key] = names = []

names.append(who)
print(votes)
```

```
>>>
{'baguette': ['Bob', 'Alice'],
 'ciabatta': ['Coco', 'Deb'],
 'brioche': ['Elmer']}
```

Relying on the `in` expression requires two accesses if the key is present, or one access and one assignment if the key is missing. This example is different from the counters example above because the value for each key can be assigned blindly to the default value of an empty `list` if the key doesn't already exist. The triple assignment statement (`votes[key] = names = []`) populates the key in one line instead of two. Once the default value has been inserted into the dictionary, I don't need to assign it again because the `list` is modified by reference in the later call to `append`.

It's also possible to rely on the KeyError exception being raised when the dictionary value is a list. This approach requires one key access if the key is present, or one key access and one assignment if it's missing, which makes it more efficient than the in condition:

```
try:
    names = votes[key]
except KeyError:
    votes[key] = names = []

names.append(who)
```

Similarly, you can use the get method to fetch a list value when the key is present, or do one fetch and one assignment if the key isn't present:

```
names = votes.get(key)
if names is None:
    votes[key] = names = []

names.append(who)
```

The approach that involves using get to fetch list values can further be shortened by one line if you use an assignment expression (introduced in Python 3.8; see Item 10: "Prevent Repetition with Assignment Expressions") in the if statement, which improves readability:

```
if (names := votes.get(key)) is None:
    votes[key] = names = []

names.append(who)
```

The dict type also provides the setdefault method to help shorten this pattern even further. setdefault tries to fetch the value of a key in the dictionary. If the key isn't present, the method assigns that key to the default value provided. And then the method returns the value for that key: either the originally present value or the newly inserted default value. Here, I use setdefault to implement the same logic as in the get example above:

```
names = votes.setdefault(key, [])
names.append(who)
```

This works as expected, and it is shorter than using get with an assignment expression. However, the readability of this approach isn't ideal. The method name setdefault doesn't make its purpose immediately obvious. Why is it set when what it's doing is getting a value? Why not call it get_or_set? I'm arguing about the color of the bike shed here, but the point is that if you were a new reader of the code and not completely familiar with Python, you might have trouble understanding what this code is trying to accomplish because setdefault isn't self-explanatory.

There's also one important gotcha: The default value passed to setdefault is assigned directly into the dictionary when the key is missing instead of being copied. Here, I demonstrate the effect of this when the value is a list:

```
data = {}
key = 'foo'
value = []
data.setdefault(key, value)
print('Before:', data)
value.append('hello')
print('After: ', data)
```

```
>>>
Before: {'foo': []}
After:  {'foo': ['hello']}
```

This means that I need to make sure that I'm always constructing a new default value for each key I access with setdefault. This leads to a significant performance overhead in this example because I have to allocate a list instance for each call. If I reuse an object for the default value—which I might try to do to increase efficiency or readability—I might introduce strange behavior and bugs (see Item 24: "Use None and Docstrings to Specify Dynamic Default Arguments" for another example of this problem).

Going back to the earlier example that used counters for dictionary values instead of lists of who voted: Why not also use the setdefault method in that case? Here, I reimplement the same example using this approach:

```
count = counters.setdefault(key, 0)
counters[key] = count + 1
```

The problem here is that the call to setdefault is superfluous. You always need to assign the key in the dictionary to a new value after you increment the counter, so the extra assignment done by setdefault is unnecessary. The earlier approach of using get for counter updates requires only one access and one assignment, whereas using setdefault requires one access and two assignments.

There are only a few circumstances in which using setdefault is the shortest way to handle missing dictionary keys, such as when the default values are cheap to construct, mutable, and there's no potential for raising exceptions (e.g., list instances). In these very specific cases, it may seem worth accepting the confusing method name setdefault instead of having to write more characters and lines to use get. However, often what you really should do in these situations is to use defaultdict instead (see Item 17: "Prefer defaultdict Over setdefault to Handle Missing Items in Internal State").

Things to Remember

- ✦ There are four common ways to detect and handle missing keys in dictionaries: using in expressions, KeyError exceptions, the get method, and the setdefault method.
- ✦ The get method is best for dictionaries that contain basic types like counters, and it is preferable along with assignment expressions when creating dictionary values has a high cost or may raise exceptions.
- ✦ When the setdefault method of dict seems like the best fit for your problem, you should consider using defaultdict instead.

Item 17: Prefer defaultdict Over setdefault to Handle Missing Items in Internal State

When working with a dictionary that you didn't create, there are a variety of ways to handle missing keys (see Item 16: "Prefer get Over in and KeyError to Handle Missing Dictionary Keys"). Although using the get method is a better approach than using in expressions and KeyError exceptions, for some use cases setdefault appears to be the shortest option.

For example, say that I want to keep track of the cities I've visited in countries around the world. Here, I do this by using a dictionary that maps country names to a set instance containing corresponding city names:

```
visits = {
    'Mexico': {'Tulum', 'Puerto Vallarta'},
    'Japan': {'Hakone'},
}
```

I can use the setdefault method to add new cities to the sets, whether the country name is already present in the dictionary or not. This approach is much shorter than achieving the same behavior with the get method and an assignment expression (which is available as of Python 3.8):

```
visits.setdefault('France', set()).add('Arles')  # Short


if (japan := visits.get('Japan')) is None:       # Long
    visits['Japan'] = japan = set()
japan.add('Kyoto')

print(visits)
```

```
>>>
{'Mexico': {'Tulum', 'Puerto Vallarta'},
 'Japan': {'Kyoto', 'Hakone'},
 'France': {'Arles'}}
```

What about the situation when you *do* control creation of the dictionary being accessed? This is generally the case when you're using a dictionary instance to keep track of the internal state of a class, for example. Here, I wrap the example above in a class with helper methods to access the dynamic inner state stored in a dictionary:

```
class Visits:
    def __init__(self):
        self.data = {}

    def add(self, country, city):
        city_set = self.data.setdefault(country, set())
        city_set.add(city)
```

This new class hides the complexity of calling setdefault correctly, and it provides a nicer interface for the programmer:

```
visits = Visits()
visits.add('Russia', 'Yekaterinburg')
visits.add('Tanzania', 'Zanzibar')
print(visits.data)
```

```
>>>
{'Russia': {'Yekaterinburg'}, 'Tanzania': {'Zanzibar'}}
```

However, the implementation of the Visits.add method still isn't ideal. The setdefault method is still confusingly named, which makes it more difficult for a new reader of the code to immediately understand what's happening. And the implementation isn't efficient because it constructs a new set instance on every call, regardless of whether the given country was already present in the data dictionary.

Luckily, the `defaultdict` class from the `collections` built-in module simplifies this common use case by automatically storing a default value when a key doesn't exist. All you have to do is provide a function that will return the default value to use each time a key is missing (an example of Item 38: "Accept Functions Instead of Classes for Simple Interfaces"). Here, I rewrite the `Visits` class to use `defaultdict`:

```
from collections import defaultdict


class Visits:
    def __init__(self):
        self.data = defaultdict(set)

    def add(self, country, city):
        self.data[country].add(city)

visits = Visits()
visits.add('England', 'Bath')
visits.add('England', 'London')
print(visits.data)
```

```
>>>
defaultdict(<class 'set'>, {'England': {'London', 'Bath'}})
```

Now, the implementation of `add` is short and simple. The code can assume that accessing any key in the `data` dictionary will always result in an existing `set` instance. No superfluous `set` instances will be allocated, which could be costly if the `add` method is called a large number of times.

Using `defaultdict` is much better than using `setdefault` for this type of situation (see Item 37: "Compose Classes Instead of Nesting Many Levels of Built-in Types" for another example). There are still cases in which `defaultdict` will fall short of solving your problems, but there are even more tools available in Python to work around those limitations (see Item 18: "Know How to Construct Key-Dependent Default Values with `__missing__`," Item 43: "Inherit from `collections.abc` for Custom Container Types," and the `collections.Counter` built-in class).

Things to Remember

- ✦ If you're creating a dictionary to manage an arbitrary set of potential keys, then you should prefer using a `defaultdict` instance from the `collections` built-in module if it suits your problem.
- ✦ If a dictionary of arbitrary keys is passed to you, and you don't control its creation, then you should prefer the `get` method to access its items. However, it's worth considering using the `setdefault` method for the few situations in which it leads to shorter code.

Item 18: Know How to Construct Key-Dependent Default Values with `__missing__`

The built-in `dict` type's `setdefault` method results in shorter code when handling missing keys in some specific circumstances (see Item 16: "Prefer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys" for examples). For many of those situations, the better tool for the job is the `defaultdict` type from the `collections` built-in module (see Item 17: "Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State" for why). However, there are times when neither `setdefault` nor `defaultdict` is the right fit.

For example, say that I'm writing a program to manage social network profile pictures on the filesystem. I need a dictionary to map profile picture pathnames to open file handles so I can read and write those images as needed. Here, I do this by using a normal `dict` instance and checking for the presence of keys using the `get` method and an assignment expression (introduced in Python 3.8; see Item 10: "Prevent Repetition with Assignment Expressions"):

```python
pictures = {}
path = 'profile_1234.png'

if (handle := pictures.get(path)) is None:
    try:
        handle = open(path, 'a+b')
    except OSError:
        print(f'Failed to open path {path}')
        raise
    else:
        pictures[path] = handle

handle.seek(0)
image_data = handle.read()
```

When the file handle already exists in the dictionary, this code makes only a single dictionary access. In the case that the file handle doesn't exist, the dictionary is accessed once by `get`, and then it is assigned in the `else` clause of the `try`/`except` block. (This approach also works with `finally`; see Item 65: "Take Advantage of Each Block in `try`/`except`/`else`/`finally`.") The call to the `read` method stands clearly separate from the code that calls `open` and handles exceptions.

Although it's possible to use the `in` expression or `KeyError` approaches to implement this same logic, those options require more dictionary accesses and levels of nesting. Given that these other options work, you might also assume that the `setdefault` method would work, too:

```
try:
    handle = pictures.setdefault(path, open(path, 'a+b'))
except OSError:
    print(f'Failed to open path {path}')
    raise
else:
    handle.seek(0)
    image_data = handle.read()
```

This code has many problems. The `open` built-in function to create the file handle is always called, even when the path is already present in the dictionary. This results in an additional file handle that may conflict with existing open handles in the same program. Exceptions may be raised by the `open` call and need to be handled, but it may not be possible to differentiate them from exceptions that may be raised by the `setdefault` call on the same line (which is possible for other dictionary-like implementations; see Item 43: "Inherit from `collections.abc` for Custom Container Types").

If you're trying to manage internal state, another assumption you might make is that a `defaultdict` could be used for keeping track of these profile pictures. Here, I attempt to implement the same logic as before but now using a helper function and the `defaultdict` class:

```
from collections import defaultdict


def open_picture(profile_path):
    try:
        return open(profile_path, 'a+b')
    except OSError:
        print(f'Failed to open path {profile_path}')
        raise

pictures = defaultdict(open_picture)
handle = pictures[path]
handle.seek(0)
image_data = handle.read()
```

```
>>>
Traceback ...
TypeError: open_picture() missing 1 required positional
argument: 'profile_path'
```

The problem is that `defaultdict` expects that the function passed to its constructor doesn't require any arguments. This means that the helper function that `defaultdict` calls doesn't know which specific key is being accessed, which eliminates my ability to call open. In this situation, both setdefault and defaultdict fall short of what I need.

Fortunately, this situation is common enough that Python has another built-in solution. You can subclass the dict type and implement the __missing__ special method to add custom logic for handling missing keys. Here, I do this by defining a new class that takes advantage of the same open_picture helper method defined above:

```
class Pictures(dict):
    def __missing__(self, key):
        value = open_picture(key)
        self[key] = value
        return value

pictures = Pictures()
handle = pictures[path]
handle.seek(0)
image_data = handle.read()
```

When the pictures[path] dictionary access finds that the path key isn't present in the dictionary, the __missing__ method is called. This method must create the new default value for the key, insert it into the dictionary, and return it to the caller. Subsequent accesses of the same path will not call __missing__ since the corresponding item is already present (similar to the behavior of __getattr__; see Item 47: "Use __getattr__, __getattribute__, and __setattr__ for Lazy Attributes").

Things to Remember

- ✦ The setdefault method of dict is a bad fit when creating the default value has high computational cost or may raise exceptions.
- ✦ The function passed to defaultdict must not require any arguments, which makes it impossible to have the default value depend on the key being accessed.
- ✦ You can define your own dict subclass with a __missing__ method in order to construct default values that must know which key was being accessed.

This page intentionally left blank

3 Functions

The first organizational tool programmers use in Python is the *function*. As in other programming languages, functions enable you to break large programs into smaller, simpler pieces with names to represent their intent. They improve readability and make code more approachable. They allow for reuse and refactoring.

Functions in Python have a variety of extra features that make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. These extras can make a function's purpose more obvious. They can eliminate noise and clarify the intention of callers. They can significantly reduce subtle bugs that are difficult to find.

Item 19: Never Unpack More Than Three Variables When Functions Return Multiple Values

One effect of the unpacking syntax (see Item 6: "Prefer Multiple Assignment Unpacking Over Indexing") is that it allows Python functions to seemingly return more than one value. For example, say that I'm trying to determine various statistics for a population of alligators. Given a `list` of lengths, I need to calculate the minimum and maximum lengths in the population. Here, I do this in a single function that appears to return two values:

```python
def get_stats(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    return minimum, maximum

lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]

minimum, maximum = get_stats(lengths)  # Two return values

print(f'Min: {minimum}, Max: {maximum}')
```