Python 3 is usually available under the name python3:

```
$ python3 --version
Python 3.8.0
```

You can also figure out the version of Python you're using at runtime by inspecting values in the sys built-in module:

```
import sys
print(sys.version_info)
print(sys.version)
```

```
>>>
sys.version_info(major=3, minor=8, micro=0,
➥releaselevel='final', serial=0)
3.8.0 (default, Oct 21 2019, 12:51:32)
[Clang 6.0 (clang-600.0.57)]
```

Python 3 is actively maintained by the Python core developers and community, and it is constantly being improved. Python 3 includes a variety of powerful new features that are covered in this book. The majority of Python's most common open source libraries are compatible with and focused on Python 3. I strongly encourage you to use Python 3 for all your Python projects.

Python 2 is scheduled for *end of life* after January 1, 2020, at which point all forms of bug fixes, security patches, and backports of features will cease. Using Python 2 after that date is a liability because it will no longer be officially maintained. If you're still stuck working in a Python 2 codebase, you should consider using helpful tools like 2to3 (preinstalled with Python) and six (available as a community package; see Item 82: "Know Where to Find Community-Built Modules") to help you make the transition to Python 3.

Things to Remember

- ✦ Python 3 is the most up-to-date and well-supported version of Python, and you should use it for your projects.
- ✦ Be sure that the command-line executable for running Python on your system is the version you expect it to be.
- ✦ Avoid Python 2 because it will no longer be maintained after January 1, 2020.

Item 2: Follow the PEP 8 Style Guide

Python Enhancement Proposal #8, otherwise known as PEP 8, is the style guide for how to format Python code. You are welcome to write Python code any way you want, as long as it has valid syntax. However, using a consistent style makes your code more approachable and easier to read. Sharing a common style with other Python programmers in the larger community facilitates collaboration on projects. But even if you are the only one who will ever read your code, following the style guide will make it easier for you to change things later, and can help you avoid many common errors.

PEP 8 provides a wealth of details about how to write clear Python code. It continues to be updated as the Python language evolves. It's worth reading the whole guide online (https://www.python.org/dev/peps/pep-0008/). Here are a few rules you should be sure to follow.

Whitespace

In Python, whitespace is syntactically significant. Python programmers are especially sensitive to the effects of whitespace on code clarity. Follow these guidelines related to whitespace:

- Use spaces instead of tabs for indentation.
- Use four spaces for each level of syntactically significant indenting.
- Lines should be 79 characters in length or less.
- Continuations of long expressions onto additional lines should be indented by four extra spaces from their normal indentation level.
- In a file, functions and classes should be separated by two blank lines.
- In a class, methods should be separated by one blank line.
- In a dictionary, put no whitespace between each key and colon, and put a single space before the corresponding value if it fits on the same line.
- Put one—and only one—space before and after the = operator in a variable assignment.
- For type annotations, ensure that there is no separation between the variable name and the colon, and use a space before the type information.

Naming

PEP 8 suggests unique styles of naming for different parts in the language. These conventions make it easy to distinguish which type corresponds to each name when reading code. Follow these guidelines related to naming:

- Functions, variables, and attributes should be in `lowercase_underscore` format.
- Protected instance attributes should be in `_leading_underscore` format.
- Private instance attributes should be in `__double_leading_underscore` format.
- Classes (including exceptions) should be in `CapitalizedWord` format.
- Module-level constants should be in `ALL_CAPS` format.
- Instance methods in classes should use `self`, which refers to the object, as the name of the first parameter.
- Class methods should use `cls`, which refers to the class, as the name of the first parameter.

Expressions and Statements

The Zen of Python states: "There should be one—and preferably only one—obvious way to do it." PEP 8 attempts to codify this style in its guidance for expressions and statements:

- Use inline negation (`if a is not b`) instead of negation of positive expressions (`if not a is b`).
- Don't check for empty containers or sequences (like `[]` or `''`) by comparing the length to zero (`if len(somelist) == 0`). Use `if not somelist` and assume that empty values will implicitly evaluate to `False`.
- The same thing goes for non-empty containers or sequences (like `[1]` or `'hi'`). The statement `if somelist` is implicitly `True` for non-empty values.
- Avoid single-line `if` statements, `for` and `while` loops, and `except` compound statements. Spread these over multiple lines for clarity.
- If you can't fit an expression on one line, surround it with parentheses and add line breaks and indentation to make it easier to read.
- Prefer surrounding multiline expressions with parentheses over using the \ line continuation character.

Imports

PEP 8 suggests some guidelines for how to import modules and use them in your code:

- Always put `import` statements (including `from x import y`) at the top of a file.
- Always use absolute names for modules when importing them, not names relative to the current module's own path. For example, to import the `foo` module from within the `bar` package, you should use `from bar import foo`, not just `import foo`.
- If you must do relative imports, use the explicit syntax `from . import foo`.
- Imports should be in sections in the following order: standard library modules, third-party modules, your own modules. Each subsection should have imports in alphabetical order.

Note

The Pylint tool (https://www.pylint.org) is a popular static analyzer for Python source code. Pylint provides automated enforcement of the PEP 8 style guide and detects many other types of common errors in Python programs. Many IDEs and editors also include linting tools or support similar plug-ins.

Things to Remember

✦ Always follow the Python Enhancement Proposal #8 (PEP 8) style guide when writing Python code.

✦ Sharing a common style with the larger Python community facilitates collaboration with others.

✦ Using a consistent style makes it easier to modify your own code later.

Item 3: Know the Differences Between bytes and str

In Python, there are two types that represent sequences of character data: `bytes` and `str`. Instances of `bytes` contain raw, unsigned 8-bit values (often displayed in the ASCII encoding):

```
a = b'h\x65llo'
print(list(a))
print(a)
```

```
>>>
[104, 101, 108, 108, 111]
b'hello'
```

Instances of str contain Unicode *code points* that represent textual characters from human languages:

```
a = 'a\u0300 propos'
print(list(a))
print(a)
```

```
>>>
['a', '`', ' ', 'p', 'r', 'o', 'p', 'o', 's']
à propos
```

Importantly, str instances do not have an associated binary encoding, and bytes instances do not have an associated text encoding. To convert Unicode data to binary data, you must call the encode method of str. To convert binary data to Unicode data, you must call the decode method of bytes. You can explicitly specify the encoding you want to use for these methods, or accept the system default, which is commonly *UTF-8* (but not always—see more on that below).

When you're writing Python programs, it's important to do encoding and decoding of Unicode data at the furthest boundary of your interfaces; this approach is often called the *Unicode sandwich*. The core of your program should use the str type containing Unicode data and should not assume anything about character encodings. This approach allows you to be very accepting of alternative text encodings (such as *Latin-1*, *Shift JIS*, and *Big5*) while being strict about your output text encoding (ideally, UTF-8).

The split between character types leads to two common situations in Python code:

- You want to operate on raw 8-bit sequences that contain UTF-8-encoded strings (or some other encoding).
- You want to operate on Unicode strings that have no specific encoding.

You'll often need two helper functions to convert between these cases and to ensure that the type of input values matches your code's expectations.

The first function takes a bytes or str instance and always returns a str:

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of str
```

```
print(repr(to_str(b'foo')))
print(repr(to_str('bar')))
```

```
>>>
'foo'
'bar'
```

The second function takes a bytes or str instance and always returns a bytes:

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value  # Instance of bytes

print(repr(to_bytes(b'foo')))
print(repr(to_bytes('bar')))
```

There are two big gotchas when dealing with raw 8-bit values and Unicode strings in Python.

The first issue is that bytes and str seem to work the same way, but their instances are not compatible with each other, so you must be deliberate about the types of character sequences that you're passing around.

By using the + operator, you can add bytes to bytes and str to str, respectively:

```
print(b'one' + b'two')
print('one' + 'two')
```

```
>>>
b'onetwo'
onetwo
```

But you can't add str instances to bytes instances:

```
b'one' + 'two'
```

```
>>>
Traceback ...
TypeError: can't concat str to bytes
```

Nor can you add bytes instances to str instances:

```
'one' + b'two'
```

```
>>>
Traceback ...
TypeError: can only concatenate str (not "bytes") to str
```

By using binary operators, you can compare bytes to bytes and str to str, respectively:

```
assert b'red' > b'blue'
assert 'red' > 'blue'
```

But you can't compare a str instance to a bytes instance:

```
assert 'red' > b'blue'
```

```
>>>
Traceback ...
TypeError: '>' not supported between instances of 'str' and
➥'bytes'
```

Nor can you compare a bytes instance to a str instance:

```
assert b'blue' < 'red'
```

```
>>>
Traceback ...
TypeError: '<' not supported between instances of 'bytes'
➥and 'str'
```

Comparing bytes and str instances for equality will always evaluate to False, even when they contain exactly the same characters (in this case, ASCII-encoded "foo"):

```
print(b'foo' == 'foo')
```

```
>>>
False
```

The % operator works with format strings for each type, respectively:

```
print(b'red %s' % b'blue')
print('red %s' % 'blue')
```

```
>>>
b'red blue'
red blue
```

But you can't pass a str instance to a bytes format string because Python doesn't know what binary text encoding to use:

```
print(b'red %s' % 'blue')
```

```
>>>
Traceback ...
TypeError: %b requires a bytes-like object, or an object that
➥implements __bytes__, not 'str'
```

You *can* pass a bytes instance to a str format string using the % operator, but it doesn't do what you'd expect:

```
print('red %s' % b'blue')
```

```
>>>
red b'blue'
```

This code actually invokes the `__repr__` method (see Item 75: "Use repr Strings for Debugging Output") on the bytes instance and substitutes that in place of the %s, which is why b'blue' remains escaped in the output.

The second issue is that operations involving file handles (returned by the open built-in function) default to requiring Unicode strings instead of raw bytes. This can cause surprising failures, especially for programmers accustomed to Python 2. For example, say that I want to write some binary data to a file. This seemingly simple code breaks:

```
with open('data.bin', 'w') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

```
>>>
Traceback ...
TypeError: write() argument must be str, not bytes
```

The cause of the exception is that the file was opened in write text mode ('w') instead of write binary mode ('wb'). When a file is in text mode, write operations expect str instances containing Unicode data instead of bytes instances containing binary data. Here, I fix this by changing the open mode to 'wb':

```
with open('data.bin', 'wb') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

A similar problem also exists for reading data from files. For example, here I try to read the binary file that was written above:

```
with open('data.bin', 'r') as f:
    data = f.read()
```

```
>>>
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in
➥position 0: invalid continuation byte
```

This fails because the file was opened in read text mode ('r') instead of read binary mode ('rb'). When a handle is in text mode, it uses the system's default text encoding to interpret binary data using the bytes.encode (for writing) and str.decode (for reading) methods. On most systems, the default encoding is UTF-8, which can't accept the binary data b'\xf1\xf2\xf3\xf4\xf5', thus causing the error above. Here, I solve this problem by changing the open mode to 'rb':

```
with open('data.bin', 'rb') as f:
    data = f.read()

assert data == b'\xf1\xf2\xf3\xf4\xf5'
```

Alternatively, I can explicitly specify the encoding parameter to the open function to make sure that I'm not surprised by any platform-specific behavior. For example, here I assume that the binary data in the file was actually meant to be a string encoded as 'cp1252' (a legacy Windows encoding):

```
with open('data.bin', 'r', encoding='cp1252') as f:
    data = f.read()

assert data == 'ñòóôõ'
```

The exception is gone, and the string interpretation of the file's contents is very different from what was returned when reading raw bytes. The lesson here is that you should check the default encoding on your system (using python3 -c 'import locale; print(locale.getpreferredencoding())') to understand how it differs from your expectations. When in doubt, you should explicitly pass the encoding parameter to open.

Things to Remember

- ✦ bytes contains sequences of 8-bit values, and str contains sequences of Unicode code points.
- ✦ Use helper functions to ensure that the inputs you operate on are the type of character sequence that you expect (8-bit values, UTF-8-encoded strings, Unicode code points, etc).
- ✦ bytes and str instances can't be used together with operators (like >, ==, +, and %).
- ✦ If you want to read or write binary data to/from a file, always open the file using a binary mode (like 'rb' or 'wb').
- ✦ If you want to read or write Unicode data to/from a file, be careful about your system's default text encoding. Explicitly pass the encoding parameter to open if you want to avoid surprises.

Item 4: Prefer Interpolated F-Strings Over C-style Format Strings and `str.format`

Strings are present throughout Python codebases. They're used for rendering messages in user interfaces and command-line utilities. They're used for writing data to files and sockets. They're used for specifying what's gone wrong in `Exception` details (see Item 27: "Use Comprehensions Instead of `map` and `filter`"). They're used in debugging (see Item 80: "Consider Interactive Debugging with `pdb`" and Item 75: "Use `repr` Strings for Debugging Output").

Formatting is the process of combining predefined text with data values into a single human-readable message that's stored as a string. Python has four different ways of formatting strings that are built into the language and standard library. All but one of them, which is covered last in this item, have serious shortcomings that you should understand and avoid.

The most common way to format a string in Python is by using the `%` formatting operator. The predefined text template is provided on the left side of the operator in a *format string*. The values to insert into the template are provided as a single value or `tuple` of multiple values on the right side of the format operator. For example, here I use the `%` operator to convert difficult-to-read binary and hexadecimal values to integer strings:

```
a = 0b10111011
b = 0xc5f
print('Binary is %d, hex is %d' % (a, b))
```

```
>>>
Binary is 187, hex is 3167
```

The format string uses format specifiers (like `%d`) as placeholders that will be replaced by values from the right side of the formatting expression. The syntax for format specifiers comes from C's `printf` function, which has been inherited by Python (as well as by other programming languages). Python supports all the usual options you'd expect from `printf`, such as `%s`, `%x`, and `%f` format specifiers, as well as control over decimal places, padding, fill, and alignment. Many programmers who are new to Python start with C-style format strings because they're familiar and simple to use.

There are four problems with C-style format strings in Python.

The first problem is that if you change the type or order of data values in the `tuple` on the right side of a formatting expression, you can get errors due to type conversion incompatibility. For example, this simple formatting expression works:

```
key = 'my_var'
value = 1.234
formatted = '%-10s = %.2f' % (key, value)
print(formatted)

>>>
my_var     = 1.23
```

But if you swap key and value, you get an exception at runtime:

```
reordered_tuple = '%-10s = %.2f' % (value, key)

>>>
Traceback ...
TypeError: must be real number, not str
```

Similarly, leaving the right side parameters in the original order but changing the format string results in the same error:

```
reordered_string = '%.2f = %-10s' % (key, value)

>>>
Traceback ...
TypeError: must be real number, not str
```

To avoid this gotcha, you need to constantly check that the two sides of the % operator are in sync; this process is error prone because it must be done manually for every change.

The second problem with C-style formatting expressions is that they become difficult to read when you need to make small modifications to values before formatting them into a string—and this is an extremely common need. Here, I list the contents of my kitchen pantry without making inline changes:

```
pantry = [
    ('avocados', 1.25),
    ('bananas', 2.5),
    ('cherries', 15),
]
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %.2f' % (i, item, count))

>>>
#0: avocados   = 1.25
#1: bananas    = 2.50
#2: cherries   = 15.00
```

Now, I make a few modifications to the values that I'm formatting to make the printed message more useful. This causes the `tuple` in the formatting expression to become so long that it needs to be split across multiple lines, which hurts readability:

```
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count)))
```

```
>>>
#1: Avocados   = 1
#2: Bananas    = 2
#3: Cherries   = 15
```

The third problem with formatting expressions is that if you want to use the same value in a format string multiple times, you have to repeat it in the right side `tuple`:

```
template = '%s loves food. See %s cook.'
name = 'Max'
formatted = template % (name, name)
print(formatted)
```

```
>>>
Max loves food. See Max cook.
```

This is especially annoying and error prone if you have to repeat small modifications to the values being formatted. For example, here I remembered to call the `title()` method multiple times, but I could have easily added the method call to one reference to `name` and not the other, which would cause mismatched output:

```
name = 'brad'
formatted = template % (name.title(), name.title())
print(formatted)
```

```
>>>
Brad loves food. See Brad cook.
```

To help solve some of these problems, the `%` operator in Python has the ability to also do formatting with a dictionary instead of a `tuple`. The keys from the dictionary are matched with format specifiers with the corresponding name, such as `%(key)s`. Here, I use this functionality to change the order of values on the right side of the formatting expression with no effect on the output, thus solving problem #1 from above:

```
key = 'my_var'
value = 1.234
```

```
old_way = '%-10s = %.2f' % (key, value)

new_way = '%(key)-10s = %(value).2f' % {
    'key': key, 'value': value}  # Original

reordered = '%(key)-10s = %(value).2f' % {
    'value': value, 'key': key}  # Swapped

assert old_way == new_way == reordered
```

Using dictionaries in formatting expressions also solves problem #3 from above by allowing multiple format specifiers to reference the same value, thus making it unnecessary to supply that value more than once:

```
name = 'Max'

template = '%s loves food. See %s cook.'
before = template % (name, name)   # Tuple

template = '%(name)s loves food. See %(name)s cook.'
after = template % {'name': name}  # Dictionary

assert before == after
```

However, dictionary format strings introduce and exacerbate other issues. For problem #2 above, regarding small modifications to values before formatting them, formatting expressions become longer and more visually noisy because of the presence of the dictionary key and colon operator on the right side. Here, I render the same string with and without dictionaries to show this problem:

```
for i, (item, count) in enumerate(pantry):
    before = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    after = '#%(loop)d: %(item)-10s = %(count)d' % {
        'loop': i + 1,
        'item': item.title(),
        'count': round(count),
    }

    assert before == after
```

Using dictionaries in formatting expressions also increases verbosity, which is problem #4 with C-style formatting expressions in Python. Each key must be specified at least twice—once in the format specifier, once in the dictionary as a key, and potentially once more for the variable name that contains the dictionary value:

```
soup = 'lentil'
formatted = 'Today\'s soup is %(soup)s.' % {'soup': soup}
print(formatted)
```

```
>>>
Today's soup is lentil.
```

Besides the duplicative characters, this redundancy causes formatting expressions that use dictionaries to be long. These expressions often must span multiple lines, with the format strings being concatenated across multiple lines and the dictionary assignments having one line per value to use in formatting:

```
menu = {
    'soup': 'lentil',
    'oyster': 'kumamoto',
    'special': 'schnitzel',
}
template = ('Today\'s soup is %(soup)s, '
            'buy one get two %(oyster)s oysters, '
            'and our special entrée is %(special)s.')
formatted = template % menu
print(formatted)
```

```
>>>
Today's soup is lentil, buy one get two kumamoto oysters, and
➥our special entrée is schnitzel.
```

To understand what this formatting expression is going to produce, your eyes have to keep going back and forth between the lines of the format string and the lines of the dictionary. This disconnect makes it hard to spot bugs, and readability gets even worse if you need to make small modifications to any of the values before formatting.

There must be a better way.

The `format` Built-in and `str.format`

Python 3 added support for *advanced string formatting* that is more expressive than the old C-style format strings that use the `%` operator. For individual Python values, this new functionality can be accessed through the `format` built-in function. For example, here I use some of the new options (`,` for thousands separators and `^` for centering) to format values:

```
a = 1234.5678
formatted = format(a, ',.2f')
print(formatted)

b = 'my string'
formatted = format(b, '^20s')
print('*', formatted, '*')
```

```
>>>
1,234.57
*      my string       *
```

You can use this functionality to format multiple values together by calling the new `format` method of the `str` type. Instead of using C-style format specifiers like `%d`, you can specify placeholders with `{}`. By default the placeholders in the format string are replaced by the corresponding positional arguments passed to the `format` method in the order in which they appear:

```
key = 'my_var'
value = 1.234

formatted = '{} = {}'.format(key, value)
print(formatted)
```

```
>>>
my_var = 1.234
```

Within each placeholder you can optionally provide a colon character followed by format specifiers to customize how values will be converted into strings (see `help('FORMATTING')` for the full range of options):

```
formatted = '{:<10} = {:.2f}'.format(key, value)
print(formatted)
```

```
>>>
my_var     = 1.23
```

The way to think about how this works is that the format specifiers will be passed to the `format` built-in function along with the value (`format(value, '.2f')` in the example above). The result of that function call is what replaces the placeholder in the overall formatted string. The formatting behavior can be customized per class using the `__format__` special method.

With C-style format strings, you need to escape the % character (by doubling it) so it's not interpreted as a placeholder accidentally. With the str.format method you need to similarly escape braces:

```
print('%.2f%%' % 12.5)
print('{} replaces {{}}'.format(1.23))

>>>
12.50%
1.23 replaces {}
```

Within the braces you may also specify the positional index of an argument passed to the format method to use for replacing the placeholder. This allows the format string to be updated to reorder the output without requiring you to also change the right side of the formatting expression, thus addressing problem #1 from above:

```
formatted = '{1} = {0}'.format(key, value)
print(formatted)

>>>
1.234 = my_var
```

The same positional index may also be referenced multiple times in the format string without the need to pass the value to the format method more than once, which solves problem #3 from above:

```
formatted = '{0} loves food. See {0} cook.'.format(name)
print(formatted)

>>>
Max loves food. See Max cook.
```

Unfortunately, the new format method does nothing to address problem #2 from above, leaving your code difficult to read when you need to make small modifications to values before formatting them. There's little difference in readability between the old and new options, which are similarly noisy:

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{}: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))

    assert old_style == new_style
```

There are even more advanced options for the specifiers used with the str.format method, such as using combinations of dictionary keys and list indexes in placeholders, and coercing values to Unicode and repr strings:

```
formatted = 'First letter is {menu[oyster][0]!r}'.format(
    menu=menu)
print(formatted)
```

```
>>>
First letter is 'k'
```

But these features don't help reduce the redundancy of repeated keys from problem #4 above. For example, here I compare the verbosity of using dictionaries in C-style formatting expressions to the new style of passing keyword arguments to the format method:

```
old_template = (
    'Today\'s soup is %(soup)s, '
    'buy one get two %(oyster)s oysters, '
    'and our special entrée is %(special)s.')
old_formatted = template % {
    'soup': 'lentil',
    'oyster': 'kumamoto',
    'special': 'schnitzel',
}

new_template = (
    'Today\'s soup is {soup}, '
    'buy one get two {oyster} oysters, '
    'and our special entrée is {special}.')
new_formatted = new_template.format(
    soup='lentil',
    oyster='kumamoto',
    special='schnitzel',
)

assert old_formatted == new_formatted
```

This style is slightly less noisy because it eliminates some quotes in the dictionary and a few characters in the format specifiers, but it's hardly compelling. Further, the advanced features of using dictionary keys and indexes within placeholders only provides a tiny subset of Python's expression functionality. This lack of expressiveness is so limiting that it undermines the value of the format method from str overall.

Given these shortcomings and the problems from C-style formatting expressions that remain (problems #2 and #4 from above), I suggest that you avoid the `str.format` method in general. It's important to know about the new mini language used in format specifiers (everything after the colon) and how to use the `format` built-in function. But the rest of the `str.format` method should be treated as a historical artifact to help you understand how Python's new *f-strings* work and why they're so great.

Interpolated Format Strings

Python 3.6 added *interpolated format strings—f-strings* for short—to solve these issues once and for all. This new language syntax requires you to prefix format strings with an `f` character, which is similar to how byte strings are prefixed with a `b` character and raw (unescaped) strings are prefixed with an `r` character.

F-strings take the expressiveness of format strings to the extreme, solving problem #4 from above by completely eliminating the redundancy of providing keys and values to be formatted. They achieve this pithiness by allowing you to reference all names in the current Python scope as part of a formatting expression:

```
key = 'my_var'
value = 1.234

formatted = f'{key} = {value}'
print(formatted)
```

```
>>>
my_var = 1.234
```

All of the same options from the new `format` built-in mini language are available after the colon in the placeholders within an f-string, as is the ability to coerce values to Unicode and `repr` strings similar to the `str.format` method:

```
formatted = f'{key!r:<10} = {value:.2f}'
print(formatted)
```

```
>>>
'my_var'   = 1.23
```

Formatting with f-strings is shorter than using C-style format strings with the `%` operator and the `str.format` method in all cases. Here, I show all these options together in order of shortest to longest, and line up the left side of the assignment so you can easily compare them:

```
f_string = f'{key:<10} = {value:.2f}'

c_tuple  = '%-10s = %.2f' % (key, value)

str_args = '{:<10} = {:.2f}'.format(key, value)

str_kw   = '{key:<10} = {value:.2f}'.format(key=key,
                                            value=value)

c_dict   = '%(key)-10s = %(value).2f' % {'key': key,
                                         'value': value}

assert c_tuple == c_dict == f_string
assert str_args == str_kw == f_string
```

F-strings also enable you to put a full Python expression within the placeholder braces, solving problem #2 from above by allowing small modifications to the values being formatted with concise syntax. What took multiple lines with C-style formatting and the str.format method now easily fits on a single line:

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{}: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))

    f_string = f'#{i+1}: {item.title():<10s} = {round(count)}'

    assert old_style == new_style == f_string
```

Or, if it's clearer, you can split an f-string over multiple lines by relying on adjacent-string concatenation (similar to C). Even though this is longer than the single-line version, it's still much clearer than any of the other multiline approaches:

```
for i, (item, count) in enumerate(pantry):
    print(f'#{i+1}: '
          f'{item.title():<10s} = '
          f'{round(count)}')
```

```
>>>
#1: Avocados   = 1
#2: Bananas    = 2
#3: Cherries   = 15
```

Python expressions may also appear within the format specifier options. For example, here I parameterize the number of digits to print by using a variable instead of hard-coding it in the format string:

```
places = 3
number = 1.23456
print(f'My number is {number:.{places}f}')
```

```
>>>
My number is 1.235
```

The combination of expressiveness, terseness, and clarity provided by f-strings makes them the best built-in option for Python programmers. Any time you find yourself needing to format values into strings, choose f-strings over the alternatives.

Things to Remember

- ✦ C-style format strings that use the % operator suffer from a variety of gotchas and verbosity problems.
- ✦ The str.format method introduces some useful concepts in its formatting specifiers mini language, but it otherwise repeats the mistakes of C-style format strings and should be avoided.
- ✦ F-strings are a new syntax for formatting values into strings that solves the biggest problems with C-style format strings.
- ✦ F-strings are succinct yet powerful because they allow for arbitrary Python expressions to be directly embedded within format specifiers.

Item 5: Write Helper Functions Instead of Complex Expressions

Python's pithy syntax makes it easy to write single-line expressions that implement a lot of logic. For example, say that I want to decode the query string from a URL. Here, each query string parameter represents an integer value:

```
from urllib.parse import parse_qs
```

```
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))
```

```
>>>
{'red': ['5'], 'blue': ['0'], 'green': ['']}
```

Some query string parameters may have multiple values, some may have single values, some may be present but have blank values, and some may be missing entirely. Using the get method on the result dictionary will return different values in each circumstance:

```
print('Red:     ', my_values.get('red'))
print('Green:   ', my_values.get('green'))
print('Opacity: ', my_values.get('opacity'))
```

```
>>>
Red:      ['5']
Green:    ['']
Opacity:  None
```

It'd be nice if a default value of 0 were assigned when a parameter isn't supplied or is blank. I might choose to do this with Boolean expressions because it feels like this logic doesn't merit a whole if statement or helper function quite yet.

Python's syntax makes this choice all too easy. The trick here is that the empty string, the empty list, and zero all evaluate to False implicitly. Thus, the expressions below will evaluate to the subexpression after the or operator when the first subexpression is False:

```
# For query string 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print(f'Red:     {red!r}')
print(f'Green:   {green!r}')
print(f'Opacity: {opacity!r}')
```

```
>>>
Red:     '5'
Green:   0
Opacity: 0
```

The red case works because the key is present in the my_values dictionary. The value is a list with one member: the string '5'. This string implicitly evaluates to True, so red is assigned to the first part of the or expression.

The green case works because the value in the my_values dictionary is a list with one member: an empty string. The empty string implicitly evaluates to False, causing the or expression to evaluate to 0.

The opacity case works because the value in the my_values dictionary is missing altogether. The behavior of the get method is to return its second argument if the key doesn't exist in the dictionary (see Item 16: "Prefer get Over in and KeyError to Handle Missing Dictionary Keys"). The default value in this case is a list with one member: an empty string. When opacity isn't found in the dictionary, this code does exactly the same thing as the green case.

However, this expression is difficult to read, and it still doesn't do everything I need. I'd also want to ensure that all the parameter values are converted to integers so I can immediately use them in mathematical expressions. To do that, I'd wrap each expression with the int built-in function to parse the string as an integer:

```
red = int(my_values.get('red', [''])[0] or 0)
```

This is now extremely hard to read. There's so much visual noise. The code isn't approachable. A new reader of the code would have to spend too much time picking apart the expression to figure out what it actually does. Even though it's nice to keep things short, it's not worth trying to fit this all on one line.

Python has if/else conditional—or ternary—expressions to make cases like this clearer while keeping the code short:

```
red_str = my_values.get('red', [''])
red = int(red_str[0]) if red_str[0] else 0
```

This is better. For less complicated situations, if/else conditional expressions can make things very clear. But the example above is still not as clear as the alternative of a full if/else statement over multiple lines. Seeing all of the logic spread out like this makes the dense version seem even more complex:

```
green_str = my_values.get('green', [''])
if green_str[0]:
    green = int(green_str[0])
else:
    green = 0
```

If you need to reuse this logic repeatedly—even just two or three times, as in this example—then writing a helper function is the way to go:

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
        return int(found[0])
    return default
```

The calling code is much clearer than the complex expression using or and the two-line version using the if/else expression:

```
green = get_first_int(my_values, 'green')
```

As soon as expressions get complicated, it's time to consider splitting them into smaller pieces and moving logic into helper functions. What you gain in readability always outweighs what brevity may have afforded you. Avoid letting Python's pithy syntax for complex expressions from getting you into a mess like this. Follow the *DRY principle*: Don't repeat yourself.

Things to Remember

✦ Python's syntax makes it easy to write single-line expressions that are overly complicated and difficult to read.

✦ Move complex expressions into helper functions, especially if you need to use the same logic repeatedly.

✦ An if/else expression provides a more readable alternative to using the Boolean operators or and and in expressions.

Item 6: Prefer Multiple Assignment Unpacking Over Indexing

Python has a built-in tuple type that can be used to create immutable, ordered sequences of values. In the simplest case, a tuple is a pair of two values, such as keys and values from a dictionary:

```
snack_calories = {
    'chips': 140,
    'popcorn': 80,
    'nuts': 190,
}
items = tuple(snack_calories.items())
print(items)
```

```
>>>
(('chips', 140), ('popcorn', 80), ('nuts', 190))
```

The values in tuples can be accessed through numerical indexes:

```
item = ('Peanut butter', 'Jelly')
first = item[0]
second = item[1]
print(first, 'and', second)
```

```
>>>
Peanut butter and Jelly
```

Once a tuple is created, you can't modify it by assigning a new value to an index:

```
pair = ('Chocolate', 'Peanut butter')
pair[0] = 'Honey'
```

```
>>>
Traceback ...
TypeError: 'tuple' object does not support item assignment
```

Python also has syntax for *unpacking*, which allows for assigning multiple values in a single statement. The patterns that you specify in unpacking assignments look a lot like trying to mutate tuples—which isn't allowed—but they actually work quite differently. For example, if you know that a tuple is a pair, instead of using indexes to access its values, you can assign it to a tuple of two variable names:

```
item = ('Peanut butter', 'Jelly')
first, second = item  # Unpacking
print(first, 'and', second)
```

```
>>>
Peanut butter and Jelly
```

Unpacking has less visual noise than accessing the tuple's indexes, and it often requires fewer lines. The same pattern matching syntax of unpacking works when assigning to lists, sequences, and multiple levels of arbitrary iterables within iterables. I don't recommend doing the following in your code, but it's important to know that it's possible and how it works:

```
favorite_snacks = {
    'salty': ('pretzels', 100),
    'sweet': ('cookies', 180),
    'veggie': ('carrots', 20),
}

((type1, (name1, cals1)),
 (type2, (name2, cals2)),
 (type3, (name3, cals3))) = favorite_snacks.items()
```

```
print(f'Favorite {type1} is {name1} with {cals1} calories')
print(f'Favorite {type2} is {name2} with {cals2} calories')
print(f'Favorite {type3} is {name3} with {cals3} calories')

>>>
Favorite salty is pretzels with 100 calories
Favorite sweet is cookies with 180 calories
Favorite veggie is carrots with 20 calories
```

Newcomers to Python may be surprised to learn that unpacking can even be used to swap values in place without the need to create temporary variables. Here, I use typical syntax with indexes to swap the values between two positions in a `list` as part of an ascending order sorting algorithm:

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                temp = a[i]
                a[i] = a[i-1]
                a[i-1] = temp

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']
```

However, with unpacking syntax, it’s possible to swap indexes in a single line:

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                a[i-1], a[i] = a[i], a[i-1]  # Swap

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']
```

The way this swap works is that the right side of the assignment (`a[i], a[i-1]`) is evaluated first, and its values are put into a new temporary, unnamed `tuple` (such as `('carrots', 'pretzels')` on the first iteration of the loops). Then, the unpacking pattern from the left side of the assignment (a[i-1], a[i]) is used to receive that tuple value and assign it to the variable names a[i-1] and a[i], respectively. This replaces 'pretzels' with 'carrots' at index 0 and 'carrots' with 'pretzels' at index 1. Finally, the temporary unnamed tuple silently goes away.

Another valuable application of unpacking is in the target list of for loops and similar constructs, such as comprehensions and generator expressions (see Item 27: "Use Comprehensions Instead of map and filter" for those). As an example for contrast, here I iterate over a list of snacks without using unpacking:

```
snacks = [('bacon', 350), ('donut', 240), ('muffin', 190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

This works, but it's noisy. There are a lot of extra characters required in order to index into the various levels of the snacks structure. Here, I achieve the same output by using unpacking along with the enumerate built-in function (see Item 7: "Prefer enumerate Over range"):

```
for rank, (name, calories) in enumerate(snacks, 1):
    print(f'#{rank}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

This is the Pythonic way to write this type of loop; it's short and easy to understand. There's usually no need to access anything using indexes.

Python provides additional unpacking functionality for list construction (see Item 13: "Prefer Catch-All Unpacking Over Slicing"), function arguments (see Item 22: "Reduce Visual Noise with Variable Positional Arguments"), keyword arguments (see Item 23: "Provide Optional Behavior with Keyword Arguments"), multiple return values (see Item 19: "Never Unpack More Than Three Variables When Functions Return Multiple Values"), and more.

Using unpacking wisely will enable you to avoid indexing when possible, resulting in clearer and more Pythonic code.

Things to Remember

- ✦ Python has special syntax called unpacking for assigning multiple values in a single statement.
- ✦ Unpacking is generalized in Python and can be applied to any iterable, including many levels of iterables within iterables.
- ✦ Reduce visual noise and increase code clarity by using unpacking to avoid explicitly indexing into sequences.

Item 7: Prefer enumerate Over range

The range built-in function is useful for loops that iterate over a set of integers:

```
from random import randint

random_bits = 0
for i in range(32):
    if randint(0, 1):
        random_bits |= 1 << i

print(bin(random_bits))
```

```
>>>
0b11101000100100000111000010000001
```

When you have a data structure to iterate over, like a list of strings, you can loop directly over the sequence:

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print(f'{flavor} is delicious')
```

```
>>>
vanilla is delicious
chocolate is delicious
pecan is delicious
strawberry is delicious
```

Often, you'll want to iterate over a list and also know the index of the current item in the list. For example, say that I want to print the ranking of my favorite ice cream flavors. One way to do it is by using range:

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print(f'{i + 1}: {flavor}')
```

```
>>>
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

This looks clumsy compared with the other examples of iterating over `flavor_list` or `range`. I have to get the length of the `list`. I have to index into the array. The multiple steps make it harder to read.

Python provides the `enumerate` built-in function to address this situation. `enumerate` wraps any iterator with a lazy generator (see Item 30: "Consider Generators Instead of Returning Lists"). `enumerate` yields pairs of the loop index and the next value from the given iterator. Here, I manually advance the returned iterator with the `next` built-in function to demonstrate what it does:

```
it = enumerate(flavor_list)
print(next(it))
print(next(it))
```

```
>>>
(0, 'vanilla')
(1, 'chocolate')
```

Each pair yielded by `enumerate` can be succinctly unpacked in a `for` statement (see Item 6: "Prefer Multiple Assignment Unpacking Over Indexing" for how that works). The resulting code is much clearer:

```
for i, flavor in enumerate(flavor_list):
    print(f'{i + 1}: {flavor}')
```

```
>>>
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

I can make this even shorter by specifying the number from which `enumerate` should begin counting (1 in this case) as the second parameter:

```
for i, flavor in enumerate(flavor_list, 1):
    print(f'{i}: {flavor}')
```

Things to Remember

✦ `enumerate` provides concise syntax for looping over an iterator and getting the index of each item from the iterator as you go.

- ✦ Prefer enumerate instead of looping over a range and indexing into a sequence.
- ✦ You can supply a second parameter to enumerate to specify the number from which to begin counting (zero is the default).

Item 8: Use zip to Process Iterators in Parallel

Often in Python you find yourself with many lists of related objects. List comprehensions make it easy to take a source list and get a derived list by applying an expression (see Item 27: “Use Comprehensions Instead of map and filter”):

```
names = ['Cecilia', 'Lise', 'Marie']
counts = [len(n) for n in names]
print(counts)
```

```
>>>
[7, 4, 5]
```

The items in the derived list are related to the items in the source list by their indexes. To iterate over both lists in parallel, I can iterate over the length of the names source list:

```
longest_name = None
max_count = 0

for i in range(len(names)):
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count

print(longest_name)
```

```
>>>
Cecilia
```

The problem is that this whole loop statement is visually noisy. The indexes into names and counts make the code hard to read. Indexing into the arrays by the loop index i happens twice. Using enumerate (see Item 7: “Prefer enumerate Over range”) improves this slightly, but it’s still not ideal:

```
for i, name in enumerate(names):
    count = counts[i]
    if count > max_count:
        longest_name = name
        max_count = count
```

To make this code clearer, Python provides the zip built-in function. zip wraps two or more iterators with a lazy generator. The zip generator yields tuples containing the next value from each iterator. These tuples can be unpacked directly within a for statement (see Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”). The resulting code is much cleaner than the code for indexing into multiple lists:

```
for name, count in zip(names, counts):
    if count > max_count:
        longest_name = name
        max_count = count
```

zip consumes the iterators it wraps one item at a time, which means it can be used with infinitely long inputs without risk of a program using too much memory and crashing.

However, beware of zip’s behavior when the input iterators are of different lengths. For example, say that I add another item to names above but forget to update counts. Running zip on the two input lists will have an unexpected result:

```
names.append('Rosalind')
for name, count in zip(names, counts):
    print(name)
```

```
>>>
Cecilia
Lise
Marie
```

The new item for 'Rosalind' isn’t there. Why not? This is just how zip works. It keeps yielding tuples until any one of the wrapped iterators is exhausted. Its output is as long as its shortest input. This approach works fine when you know that the iterators are of the same length, which is often the case for derived lists created by list comprehensions.

But in many other cases, the truncating behavior of zip is surprising and bad. If you don’t expect the lengths of the lists passed to zip to be equal, consider using the zip_longest function from the itertools built-in module instead:

```
import itertools


for name, count in itertools.zip_longest(names, counts):
    print(f'{name}: {count}')
```

```
>>>
Cecilia: 7
Lise: 4
Marie: 5
Rosalind: None
```

`zip_longest` replaces missing values—the length of the string `'Rosalind'` in this case—with whatever `fillvalue` is passed to it, which defaults to `None`.

Things to Remember

- ✦ The `zip` built-in function can be used to iterate over multiple iterators in parallel.
- ✦ `zip` creates a lazy generator that produces tuples, so it can be used on infinitely long inputs.
- ✦ `zip` truncates its output silently to the shortest iterator if you supply it with iterators of different lengths.
- ✦ Use the `zip_longest` function from the `itertools` built-in module if you want to use `zip` on iterators of unequal lengths without truncation.

Item 9: Avoid `else` Blocks After `for` and `while` Loops

Python loops have an extra feature that is not available in most other programming languages: You can put an `else` block immediately after a loop's repeated interior block:

```
for i in range(3):
    print('Loop', i)
else:
    print('Else block!')
```

```
>>>
Loop 0
Loop 1
Loop 2
Else block!
```

Surprisingly, the `else` block runs immediately after the loop finishes. Why is the clause called "else"? Why not "and"? In an `if`/`else` statement, `else` means "Do this if the block before this doesn't happen." In a `try`/`except` statement, `except` has the same definition: "Do this if trying the block before this failed."

Similarly, else from try/except/else follows this pattern (see Item 65: “Take Advantage of Each Block in try/except/else/finally”) because it means “Do this if there was no exception to handle.” try/finally is also intuitive because it means “Always do this after trying the block before.”

Given all the uses of else, except, and finally in Python, a new programmer might assume that the else part of for/else means “Do this if the loop wasn’t completed.” In reality, it does exactly the opposite. Using a break statement in a loop actually skips the else block:

```
for i in range(3):
    print('Loop', i)
    if i == 1:
        break
else:
    print('Else block!')
```

```
>>>
Loop 0
Loop 1
```

Another surprise is that the else block runs immediately if you loop over an empty sequence:

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')
```

```
>>>
For Else block!
```

The else block also runs when while loops are initially False:

```
while False:
    print('Never runs')
else:
    print('While Else block!')
```

```
>>>
While Else block!
```

The rationale for these behaviors is that else blocks after loops are useful when using loops to search for something. For example, say that I want to determine whether two numbers are coprime (that is, their only common divisor is 1). Here, I iterate through every possible common divisor and test the numbers. After every option has been tried, the loop ends. The else block runs when the numbers are coprime because the loop doesn't encounter a break:

```
a = 4
b = 9

for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')

>>>
Testing 2
Testing 3
Testing 4
Coprime
```

In practice, I wouldn't write the code this way. Instead, I'd write a helper function to do the calculation. Such a helper function is written in two common styles.

The first approach is to return early when I find the condition I'm looking for. I return the default outcome if I fall through the loop:

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True

assert coprime(4, 9)
assert not coprime(3, 6)
```

The second way is to have a result variable that indicates whether I've found what I'm looking for in the loop. I break out of the loop as soon as I find something:

```
def coprime_alternate(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
```

```
assert coprime_alternate(4, 9)
assert not coprime_alternate(3, 6)
```

Both approaches are much clearer to readers of unfamiliar code. Depending on the situation, either may be a good choice. However, the expressivity you gain from the `else` block doesn't outweigh the burden you put on people (including yourself) who want to understand your code in the future. Simple constructs like loops should be self-evident in Python. You should avoid using `else` blocks after loops entirely.

Things to Remember

- ✦ Python has special syntax that allows `else` blocks to immediately follow `for` and `while` loop interior blocks.
- ✦ The `else` block after a loop runs only if the loop body did not encounter a `break` statement.
- ✦ Avoid using `else` blocks after loops because their behavior isn't intuitive and can be confusing.

Item 10: Prevent Repetition with Assignment Expressions

An assignment expression—also known as the *walrus operator*—is a new syntax introduced in Python 3.8 to solve a long-standing problem with the language that can cause code duplication. Whereas normal assignment statements are written `a = b` and pronounced "a equals b," these assignments are written `a := b` and pronounced "a *walrus* b" (because := looks like a pair of eyeballs and tusks).

Assignment expressions are useful because they enable you to assign variables in places where assignment statements are disallowed, such as in the conditional expression of an `if` statement. An assignment expression's value evaluates to whatever was assigned to the identifier on the left side of the walrus operator.

For example, say that I have a basket of fresh fruit that I'm trying to manage for a juice bar. Here, I define the contents of the basket:

```
fresh_fruit = {
    'apple': 10,
    'banana': 8,
    'lemon': 5,
}
```

When a customer comes to the counter to order some lemonade, I need to make sure there is at least one lemon in the basket to squeeze. Here, I do this by retrieving the count of lemons and then using an `if` statement to check for a non-zero value:

```
def make_lemonade(count):
    ...

def out_of_stock():
    ...

count = fresh_fruit.get('lemon', 0)
if count:
    make_lemonade(count)
else:
    out_of_stock()
```

The problem with this seemingly simple code is that it's noisier than it needs to be. The `count` variable is used only within the first block of the `if` statement. Defining `count` above the `if` statement causes it to appear to be more important than it really is, as if all code that follows, including the `else` block, will need to access the `count` variable, when in fact that is not the case.

This pattern of fetching a value, checking to see if it's non-zero, and then using it is extremely common in Python. Many programmers try to work around the multiple references to `count` with a variety of tricks that hurt readability (see Item 5: "Write Helper Functions Instead of Complex Expressions" for an example). Luckily, assignment expressions were added to the language to streamline exactly this type of code. Here, I rewrite this example using the walrus operator:

```
if count := fresh_fruit.get('lemon', 0):
    make_lemonade(count)
else:
    out_of_stock()
```

Though this is only one line shorter, it's a lot more readable because it's now clear that `count` is only relevant to the first block of the `if` statement. The assignment expression is first assigning a value to the `count` variable, and then evaluating that value in the context of the `if` statement to determine how to proceed with flow control. This two-step behavior—assign and then evaluate—is the fundamental nature of the walrus operator.

Lemons are quite potent, so only one is needed for my lemonade recipe, which means a non-zero check is good enough. If a customer orders a cider, though, I need to make sure that I have at least four apples. Here, I do this by fetching the count from the fruit_basket dictionary, and then using a comparison in the if statement conditional expression:

```
def make_cider(count):
    ...

count = fresh_fruit.get('apple', 0)
if count >= 4:
    make_cider(count)
else:
    out_of_stock()
```

This has the same problem as the lemonade example, where the assignment of count puts distracting emphasis on that variable. Here, I improve the clarity of this code by also using the walrus operator:

```
if (count := fresh_fruit.get('apple', 0)) >= 4:
    make_cider(count)
else:
    out_of_stock()
```

This works as expected and makes the code one line shorter. It's important to note how I needed to surround the assignment expression with parentheses to compare it with 4 in the if statement. In the lemonade example, no surrounding parentheses were required because the assignment expression stood on its own as a non-zero check; it wasn't a subexpression of a larger expression. As with other expressions, you should avoid surrounding assignment expressions with parentheses when possible.

Another common variation of this repetitive pattern occurs when I need to assign a variable in the enclosing scope depending on some condition, and then reference that variable shortly afterward in a function call. For example, say that a customer orders some banana smoothies. In order to make them, I need to have at least two bananas' worth of slices, or else an OutOfBananas exception will be raised. Here, I implement this logic in a typical way:

```
def slice_bananas(count):
    ...

class OutOfBananas(Exception):
    pass
```

```
def make_smoothies(count):
    ...

pieces = 0
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

The other common way to do this is to put the `pieces = 0` assignment in the `else` block:

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

This second approach can feel odd because it means that the `pieces` variable has two different locations—in each block of the `if` statement—where it can be initially defined. This split definition technically works because of Python's scoping rules (see Item 21: "Know How Closures Interact with Variable Scope"), but it isn't easy to read or discover, which is why many people prefer the construct above, where the `pieces = 0` assignment is first.

The walrus operator can again be used to shorten this example by one line of code. This small change removes any emphasis on the `count` variable. Now, it's clearer that `pieces` will be important beyond the `if` statement:

```
pieces = 0
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Using the walrus operator also improves the readability of splitting the definition of pieces across both parts of the if statement. It's easier to trace the pieces variable when the count definition no longer precedes the if statement:

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

One frustration that programmers who are new to Python often have is the lack of a flexible switch/case statement. The general style for approximating this type of functionality is to have a deep nesting of multiple if, elif, and else statements.

For example, imagine that I want to implement a system of precedence so that each customer automatically gets the best juice available and doesn't have to order. Here, I define logic to make it so banana smoothies are served first, followed by apple cider, and then finally lemonade:

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
else:
    count = fresh_fruit.get('apple', 0)
    if count >= 4:
        to_enjoy = make_cider(count)
    else:
        count = fresh_fruit.get('lemon', 0)
        if count:
            to_enjoy = make_lemonade(count)
        else:
            to_enjoy = 'Nothing'
```

Ugly constructs like this are surprisingly common in Python code. Luckily, the walrus operator provides an elegant solution that can feel nearly as versatile as dedicated syntax for switch/case statements:

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
```

```
elif (count := fresh_fruit.get('apple', 0)) >= 4:
    to_enjoy = make_cider(count)
elif count := fresh_fruit.get('lemon', 0):
    to_enjoy = make_lemonade(count)
else:
    to_enjoy = 'Nothing'
```

The version that uses assignment expressions is only five lines shorter than the original, but the improvement in readability is vast due to the reduction in nesting and indentation. If you ever see such ugly constructs emerge in your code, I suggest that you move them over to using the walrus operator if possible.

Another common frustration of new Python programmers is the lack of a do/while loop construct. For example, say that I want to bottle juice as new fruit is delivered until there's no fruit remaining. Here, I implement this logic with a `while` loop:

```
def pick_fruit():
    ...

def make_juice(fruit, count):
    ...

bottles = []
fresh_fruit = pick_fruit()
while fresh_fruit:
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
    fresh_fruit = pick_fruit()
```

This is repetitive because it requires two separate `fresh_fruit = pick_fruit()` calls: one before the loop to set initial conditions, and another at the end of the loop to replenish the `list` of delivered fruit.

A strategy for improving code reuse in this situation is to use the *loop-and-a-half* idiom. This eliminates the redundant lines, but it also undermines the `while` loop's contribution by making it a dumb infinite loop. Now, all of the flow control of the loop depends on the conditional `break` statement:

```
bottles = []
while True:                     # Loop
    fresh_fruit = pick_fruit()
    if not fresh_fruit:         # And a half
        break
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

The walrus operator obviates the need for the loop-and-a-half idiom by allowing the `fresh_fruit` variable to be reassigned and then conditionally evaluated each time through the `while` loop. This solution is short and easy to read, and it should be the preferred approach in your code:

```
bottles = []
while fresh_fruit := pick_fruit():
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

There are many other situations where assignment expressions can be used to eliminate redundancy (see Item 29: "Avoid Repeated Work in Comprehensions by Using Assignment Expressions" for another). In general, when you find yourself repeating the same expression or assignment multiple times within a grouping of lines, it's time to consider using assignment expressions in order to improve readability.

Things to Remember

✦ Assignment expressions use the walrus operator (:=) to both assign and evaluate variable names in a single expression, thus reducing repetition.

✦ When an assignment expression is a subexpression of a larger expression, it must be surrounded with parentheses.

✦ Although switch/case statements and do/while loops are not available in Python, their functionality can be emulated much more clearly by using assignment expressions.

This page intentionally left blank

2 Lists and Dictionaries

Many programs are written to automate repetitive tasks that are better suited to machines than to humans. In Python, the most common way to organize this kind of work is by using a sequence of values stored in a `list` type. Lists are extremely versatile and can be used to solve a variety of problems.

A natural complement to lists is the `dict` type, which stores lookup keys mapped to corresponding values (in what is often called an *associative array* or a *hash table*). Dictionaries provide constant time (amortized) performance for assignments and accesses, which means they are ideal for bookkeeping dynamic information.

Python has special syntax and built-in modules that enhance readability and extend the capabilities of lists and dictionaries beyond what you might expect from simple array, vector, and hash table types in other languages.

Item 11: Know How to Slice Sequences

Python includes syntax for *slicing* sequences into pieces. Slicing allows you to access a subset of a sequence's items with minimal effort. The simplest uses for slicing are the built-in types `list`, `str`, and `bytes`. Slicing can be extended to any Python class that implements the `__getitem__` and `__setitem__` special methods (see Item 43: "Inherit from `collections.abc` for Custom Container Types").

The basic form of the slicing syntax is `somelist[start:end]`, where start is inclusive and end is exclusive:

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('Middle two:  ', a[3:5])
print('All but ends:', a[1:7])
```

```
>>>
Middle two:   ['d', 'e']
All but ends: ['b', 'c', 'd', 'e', 'f', 'g']
```