```
>>>
Min: 60, Max: 73
```

The way this works is that multiple values are returned together in a two-item `tuple`. The calling code then unpacks the returned `tuple` by assigning two variables. Here, I use an even simpler example to show how an unpacking statement and multiple-return function work the same way:

```
first, second = 1, 2
assert first == 1
assert second == 2

def my_function():
    return 1, 2

first, second = my_function()
assert first == 1
assert second == 2
```

Multiple return values can also be received by starred expressions for catch-all unpacking (see Item 13: "Prefer Catch-All Unpacking Over Slicing"). For example, say I need another function that calculates how big each alligator is relative to the population average. This function returns a `list` of ratios, but I can receive the longest and shortest items individually by using a starred expression for the middle portion of the `list`:

```
def get_avg_ratio(numbers):
    average = sum(numbers) / len(numbers)
    scaled = [x / average for x in numbers]
    scaled.sort(reverse=True)
    return scaled

longest, *middle, shortest = get_avg_ratio(lengths)

print(f'Longest:  {longest:>4.0%}')
print(f'Shortest: {shortest:>4.0%}')
```

```
>>>
Longest:  108%
Shortest:  89%
```

Now, imagine that the program's requirements change, and I need to also determine the average length, median length, and total population size of the alligators. I can do this by expanding the `get_stats`

function to also calculate these statistics and return them in the result tuple that is unpacked by the caller:

```
def get_stats(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    count = len(numbers)
    average = sum(numbers) / count

    sorted_numbers = sorted(numbers)
    middle = count // 2
    if count % 2 == 0:
        lower = sorted_numbers[middle - 1]
        upper = sorted_numbers[middle]
        median = (lower + upper) / 2
    else:
        median = sorted_numbers[middle]

    return minimum, maximum, average, median, count

minimum, maximum, average, median, count = get_stats(lengths)

print(f'Min: {minimum}, Max: {maximum}')
print(f'Average: {average}, Median: {median}, Count {count}')
```

```
>>>
Min: 60, Max: 73
Average: 67.5, Median: 68.5, Count 10
```

There are two problems with this code. First, all the return values are numeric, so it is all too easy to reorder them accidentally (e.g., swapping average and median), which can cause bugs that are hard to spot later. Using a large number of return values is extremely error prone:

```
# Correct:
minimum, maximum, average, median, count = get_stats(lengths)

# Oops! Median and average swapped:
minimum, maximum, median, average, count = get_stats(lengths)
```

Second, the line that calls the function and unpacks the values is long, and it likely will need to be wrapped in one of a variety of ways (due to PEP8 style; see Item 2: "Follow the PEP 8 Style Guide"), which hurts readability:

```
minimum, maximum, average, median, count = get_stats(
    lengths)
```

```
minimum, maximum, average, median, count = \
    get_stats(lengths)

(minimum, maximum, average,
 median, count) = get_stats(lengths)

(minimum, maximum, average, median, count
    ) = get_stats(lengths)
```

To avoid these problems, you should never use more than three variables when unpacking the multiple return values from a function. These could be individual values from a three-tuple, two variables and one catch-all starred expression, or anything shorter. If you need to unpack more return values than that, you're better off defining a lightweight class or `namedtuple` (see Item 37: "Compose Classes Instead of Nesting Many Levels of Built-in Types") and having your function return an instance of that instead.

Things to Remember

- You can have functions return multiple values by putting them in a `tuple` and having the caller take advantage of Python's unpacking syntax.
- Multiple return values from a function can also be unpacked by catch-all starred expressions.
- Unpacking into four or more variables is error prone and should be avoided; instead, return a small class or `namedtuple` instance.

Item 20: Prefer Raising Exceptions to Returning `None`

When writing utility functions, there's a draw for Python programmers to give special meaning to the return value of `None`. It seems to make sense in some cases. For example, say I want a helper function that divides one number by another. In the case of dividing by zero, returning `None` seems natural because the result is undefined:

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Code using this function can interpret the return value accordingly:

```
x, y = 1, 0
result = careful_divide(x, y)
if result is None:
    print('Invalid inputs')
```

What happens with the careful_divide function when the numerator is zero? If the denominator is not zero, the function returns zero. The problem is that a zero return value can cause issues when you evaluate the result in a condition like an if statement. You might accidentally look for any False-equivalent value to indicate errors instead of only looking for None (see Item 5: "Write Helper Functions Instead of Complex Expressions" for a similar situation):

```
x, y = 0, 5
result = careful_divide(x, y)
if not result:
    print('Invalid inputs')  # This runs! But shouldn't
```

```
>>>
Invalid inputs
```

This misinterpretation of a False-equivalent return value is a common mistake in Python code when None has special meaning. This is why returning None from a function like careful_divide is error prone. There are two ways to reduce the chance of such errors.

The first way is to split the return value into a two-tuple (see Item 19: "Never Unpack More Than Three Variables When Functions Return Multiple Values" for background). The first part of the tuple indicates that the operation was a success or failure. The second part is the actual result that was computed:

```
def careful_divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Callers of this function have to unpack the tuple. That forces them to consider the status part of the tuple instead of just looking at the result of division:

```
success, result = careful_divide(x, y)
if not success:
    print('Invalid inputs')
```

The problem is that callers can easily ignore the first part of the tuple (using the underscore variable name, a Python convention for unused variables). The resulting code doesn't look wrong at first glance, but this can be just as error prone as returning None:

```
_, result = careful_divide(x, y)
if not result:
    print('Invalid inputs')
```

The second, better way to reduce these errors is to never return `None` for special cases. Instead, raise an `Exception` up to the caller and have the caller deal with it. Here, I turn a `ZeroDivisionError` into a `ValueError` to indicate to the caller that the input values are bad (see Item 87: "Define a Root `Exception` to Insulate Callers from APIs" on when you should use `Exception` subclasses):

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')
```

The caller no longer requires a condition on the return value of the function. Instead, it can assume that the return value is always valid and use the results immediately in the `else` block after `try` (see Item 65: "Take Advantage of Each Block in `try/except/else/finally`" for details):

```
x, y = 5, 2
try:
    result = careful_divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)
```

```
>>>
Result is 2.5
```

This approach can be extended to code using type annotations (see Item 90: "Consider Static Analysis via `typing` to Obviate Bugs" for background). You can specify that a function's return value will always be a `float` and thus will never be `None`. However, Python's gradual typing purposefully doesn't provide a way to indicate when exceptions are part of a function's interface (also known as *checked exceptions*). Instead, you have to document the exception-raising behavior and expect callers to rely on that in order to know which `Exceptions` they should plan to catch (see Item 84: "Write Docstrings for Every Function, Class, and Module").

Pulling it all together, here's what this function should look like when using type annotations and docstrings:

```
def careful_divide(a: float, b: float) -> float:
    """Divides a by b.

    Raises:
        ValueError: When the inputs cannot be divided.
    """
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')
```

Now the inputs, outputs, and exceptional behavior is clear, and the chance of a caller doing the wrong thing is extremely low.

Things to Remember

- Functions that return `None` to indicate special meaning are error prone because `None` and other values (e.g., zero, the empty string) all evaluate to `False` in conditional expressions.
- Raise exceptions to indicate special situations instead of returning `None`. Expect the calling code to handle exceptions properly when they're documented.
- Type annotations can be used to make it clear that a function will never return the value `None`, even in special situations.

Item 21: Know How Closures Interact with Variable Scope

Say that I want to sort a `list` of numbers but prioritize one group of numbers to come first. This pattern is useful when you're rendering a user interface and want important messages or exceptional events to be displayed before everything else.

A common way to do this is to pass a helper function as the `key` argument to a list's `sort` method (see Item 14: "Sort by Complex Criteria Using the `key` Parameter" for details). The helper's return value will be used as the value for sorting each item in the `list`. The helper can check whether the given item is in the important group and can vary the sorting value accordingly:

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
        return (1, x)
    values.sort(key=helper)
```

This function works for simple inputs:

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
```

```
>>>
[2, 3, 5, 7, 1, 4, 6, 8]
```

There are three reasons this function operates as expected:

- Python supports *closures*—that is, functions that refer to variables from the scope in which they were defined. This is why the `helper` function is able to access the `group` argument for `sort_priority`.
- Functions are *first-class* objects in Python, which means you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and `if` statements, and so on. This is how the `sort` method can accept a closure function as the `key` argument.
- Python has specific rules for comparing sequences (including tuples). It first compares items at index zero; then, if those are equal, it compares items at index one; if they are still equal, it compares items at index two, and so on. This is why the return value from the `helper` closure causes the sort order to have two distinct groups.

It'd be nice if this function returned whether higher-priority items were seen at all so the user interface code can act accordingly. Adding such behavior seems straightforward. There's already a closure function for deciding which group each number is in. Why not also use the closure to flip a flag when high-priority items are seen? Then, the function can return the flag value after it's been modified by the closure.

Here, I try to do that in a seemingly obvious way:

```
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True  # Seems simple
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

I can run the function on the same inputs as before:

```
found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)
```

```
>>>
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]
```

The sorted results are correct, which means items from `group` were definitely found in `numbers`. Yet the `found` result returned by the function is `False` when it should be `True`. How could this happen?

When you reference a variable in an expression, the Python interpreter traverses the scope to resolve the reference in this order:

1. The current function's scope.
2. Any enclosing scopes (such as other containing functions).
3. The scope of the module that contains the code (also called the *global scope*).
4. The built-in scope (that contains functions like `len` and `str`).

If none of these places has defined a variable with the referenced name, then a `NameError` exception is raised:

```
foo = does_not_exist * 5
```

```
>>>
Traceback ...
NameError: name 'does_not_exist' is not defined
```

Assigning a value to a variable works differently. If the variable is already defined in the current scope, it will just take on the new value. If the variable doesn't exist in the current scope, Python treats the assignment as a variable definition. Critically, the scope of the newly defined variable is the function that contains the assignment.

This assignment behavior explains the wrong return value of the `sort_priority2` function. The `found` variable is assigned to `True` in the `helper` closure. The closure's assignment is treated as a new variable definition within `helper`, not as an assignment within `sort_priority2`:

```
def sort_priority2(numbers, group):
    found = False         # Scope: 'sort_priority2'
    def helper(x):
        if x in group:
            found = True  # Scope: 'helper' -- Bad!
            return (0, x)
        return (1, x)
```

```
    numbers.sort(key=helper)
    return found
```

This problem is sometimes called the *scoping bug* because it can be so surprising to newbies. But this behavior is the intended result: It prevents local variables in a function from polluting the containing module. Otherwise, every assignment within a function would put garbage into the global module scope. Not only would that be noise, but the interplay of the resulting global variables could cause obscure bugs.

In Python, there is special syntax for getting data out of a closure. The `nonlocal` statement is used to indicate that scope traversal should happen upon assignment for a specific variable name. The only limit is that `nonlocal` won't traverse up to the module-level scope (to avoid polluting globals).

Here, I define the same function again, now using `nonlocal`:

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found  # Added
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

The `nonlocal` statement makes it clear when data is being assigned out of a closure and into another scope. It's complementary to the `global` statement, which indicates that a variable's assignment should go directly into the module scope.

However, much as with the anti-pattern of global variables, I'd caution against using `nonlocal` for anything beyond simple functions. The side effects of `nonlocal` can be hard to follow. It's especially hard to understand in long functions where the `nonlocal` statements and assignments to associated variables are far apart.

When your usage of `nonlocal` starts getting complicated, it's better to wrap your state in a helper class. Here, I define a class that achieves the same result as the `nonlocal` approach; it's a little longer but much easier to read (see Item 38: "Accept Functions Instead of Classes for Simple Interfaces" for details on the `__call__` special method):

```
class Sorter:
    def __init__(self, group):
```

```
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

Things to Remember

- Closure functions can refer to variables from any of the scopes in which they were defined.
- By default, closures can't affect enclosing scopes by assigning variables.
- Use the `nonlocal` statement to indicate when a closure can modify a variable in its enclosing scopes.
- Avoid using `nonlocal` statements for anything beyond simple functions.

Item 22: Reduce Visual Noise with Variable Positional Arguments

Accepting a variable number of positional arguments can make a function call clearer and reduce visual noise. (These positional arguments are often called *varargs* for short, or *star args*, in reference to the conventional name for the parameter `*args`.) For example, say that I want to log some debugging information. With a fixed number of arguments, I would need a function that takes a message and a `list` of values:

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')

log('My numbers are', [1, 2])
log('Hi there', [])
```

```
>>>
My numbers are: 1, 2
Hi there
```

Having to pass an empty `list` when I have no values to log is cumbersome and noisy. It'd be better to leave out the second argument entirely. I can do this in Python by prefixing the last positional parameter name with *. The first parameter for the log message is required, whereas any number of subsequent positional arguments are optional. The function body doesn't need to change; only the callers do:

```
def log(message, *values):  # The only difference
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')

log('My numbers are', 1, 2)
log('Hi there')  # Much better
```

```
>>>
My numbers are: 1, 2
Hi there
```

You might notice that this syntax works very similarly to the starred expressions used in unpacking assignment statements (see Item 13: "Prefer Catch-All Unpacking Over Slicing").

If I already have a sequence (like a `list`) and want to call a variadic function like `log`, I can do this by using the * operator. This instructs Python to pass items from the sequence as positional arguments to the function:

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)
```

```
>>>
Favorite colors: 7, 33, 99
```

There are two problems with accepting a variable number of positional arguments.

The first issue is that these optional positional arguments are always turned into a `tuple` before they are passed to a function. This means that if the caller of a function uses the * operator on a generator, it will be iterated until it's exhausted (see Item 30: "Consider Generators Instead of Returning Lists" for background). The resulting `tuple`

includes every value from the generator, which could consume a lot of memory and cause the program to crash:

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)
```

```
>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Functions that accept `*args` are best for situations where you know the number of inputs in the argument list will be reasonably small. `*args` is ideal for function calls that pass many literals or variable names together. It's primarily for the convenience of the programmer and the readability of the code.

The second issue with `*args` is that you can't add new positional arguments to a function in the future without migrating every caller. If you try to add a positional argument in the front of the argument list, existing callers will subtly break if they aren't updated:

```
def log(sequence, message, *values):
    if not values:
        print(f'{sequence} - {message}')
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{sequence} - {message}: {values_str}')

log(1, 'Favorites', 7, 33)      # New with *args OK
log(1, 'Hi there')              # New message only OK
log('Favorite numbers', 7, 33)  # Old usage breaks
```

```
>>>
1 - Favorites: 7, 33
1 - Hi there
Favorite numbers - 7: 33
```

The problem here is that the third call to `log` used `7` as the `message` parameter because a `sequence` argument wasn't given. Bugs like this are hard to track down because the code still runs without raising exceptions. To avoid this possibility entirely, you should use keyword-only arguments when you want to extend functions that

accept *args (see Item 25: "Enforce Clarity with Keyword-Only and Positional-Only Arguments"). To be even more defensive, you could also consider using type annotations (see Item 90: "Consider Static Analysis via typing to Obviate Bugs").

Things to Remember

- Functions can accept a variable number of positional arguments by using *args in the def statement.
- You can use the items from a sequence as the positional arguments for a function with the * operator.
- Using the * operator with a generator may cause a program to run out of memory and crash.
- Adding new positional parameters to functions that accept *args can introduce hard-to-detect bugs.

Item 23: Provide Optional Behavior with Keyword Arguments

As in most other programming languages, in Python you may pass arguments by position when calling a function:

```
def remainder(number, divisor):
    return number % divisor

assert remainder(20, 7) == 6
```

All normal arguments to Python functions can also be passed by keyword, where the name of the argument is used in an assignment within the parentheses of a function call. The keyword arguments can be passed in any order as long as all of the required positional arguments are specified. You can mix and match keyword and positional arguments. These calls are equivalent:

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Positional arguments must be specified before keyword arguments:

```
remainder(number=20, 7)
```

```
>>>
Traceback ...
SyntaxError: positional argument follows keyword argument
```

Each argument can be specified only once:

```
remainder(20, number=7)
```

```
>>>
Traceback ...
TypeError: remainder() got multiple values for argument
➥'number'
```

If you already have a dictionary, and you want to use its contents to call a function like `remainder`, you can do this by using the `**` operator. This instructs Python to pass the values from the dictionary as the corresponding keyword arguments of the function:

```
my_kwargs = {
    'number': 20,
    'divisor': 7,
}
assert remainder(**my_kwargs) == 6
```

You can mix the `**` operator with positional arguments or keyword arguments in the function call, as long as no argument is repeated:

```
my_kwargs = {
    'divisor': 7,
}
assert remainder(number=20, **my_kwargs) == 6
```

You can also use the `**` operator multiple times if you know that the dictionaries don't contain overlapping keys:

```
my_kwargs = {
    'number': 20,
}
other_kwargs = {
    'divisor': 7,
}
assert remainder(**my_kwargs, **other_kwargs) == 6
```

And if you'd like for a function to receive any named keyword argument, you can use the `**kwargs` catch-all parameter to collect those arguments into a `dict` that you can then process (see Item 26: "Define Function Decorators with `functools.wraps`" for when this is especially useful):

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

print_parameters(alpha=1.5, beta=9, gamma=4)
```

```
>>>
alpha = 1.5
beta = 9
gamma = 4
```

The flexibility of keyword arguments provides three significant benefits.

The first benefit is that keyword arguments make the function call clearer to new readers of the code. With the call `remainder(20, 7)`, it's not evident which argument is `number` and which is `divisor` unless you look at the implementation of the `remainder` method. In the call with keyword arguments, `number=20` and `divisor=7` make it immediately obvious which parameter is being used for each purpose.

The second benefit of keyword arguments is that they can have default values specified in the function definition. This allows a function to provide additional capabilities when you need them, but you can accept the default behavior most of the time. This eliminates repetitive code and reduces noise.

For example, say that I want to compute the rate of fluid flowing into a vat. If the vat is also on a scale, then I could use the difference between two weight measurements at two different times to determine the flow rate:

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff

weight_diff = 0.5
time_diff = 3
flow = flow_rate(weight_diff, time_diff)
print(f'{flow:.3} kg per second')
```

```
>>>
0.167 kg per second
```

In the typical case, it's useful to know the flow rate in kilograms per second. Other times, it'd be helpful to use the last sensor measurements to approximate larger time scales, like hours or days. I can provide this behavior in the same function by adding an argument for the time period scaling factor:

```
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

The problem is that now I need to specify the `period` argument every time I call the function, even in the common case of flow rate per second (where the period is 1):

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

To make this less noisy, I can give the `period` argument a default value:

```
def flow_rate(weight_diff, time_diff, period=1):
    return (weight_diff / time_diff) * period
```

The `period` argument is now optional:

```
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

This works well for simple default values; it gets tricky for complex default values (see Item 24: "Use `None` and Docstrings to Specify Dynamic Default Arguments" for details).

The third reason to use keyword arguments is that they provide a powerful way to extend a function's parameters while remaining backward compatible with existing callers. This means you can provide additional functionality without having to migrate a lot of existing code, which reduces the chance of introducing bugs.

For example, say that I want to extend the `flow_rate` function above to calculate flow rates in weight units besides kilograms. I can do this by adding a new optional parameter that provides a conversion rate to alternative measurement units:

```
def flow_rate(weight_diff, time_diff,
              period=1, units_per_kg=1):
    return ((weight_diff * units_per_kg) / time_diff) * period
```

The default argument value for `units_per_kg` is `1`, which makes the returned weight units remain kilograms. This means that all existing callers will see no change in behavior. New callers to `flow_rate` can specify the new keyword argument to see the new behavior:

```
pounds_per_hour = flow_rate(weight_diff, time_diff,
                            period=3600, units_per_kg=2.2)
```

Providing backward compatibility using optional keyword arguments like this is also crucial for functions that accept `*args` (see Item 22: "Reduce Visual Noise with Variable Positional Arguments").

The only problem with this approach is that optional keyword arguments like `period` and `units_per_kg` may still be specified as positional arguments:

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

Supplying optional arguments positionally can be confusing because it isn't clear what the values `3600` and `2.2` correspond to. The best practice is to always specify optional arguments using the keyword

names and never pass them as positional arguments. As a function author, you can also require that all callers use this more explicit keyword style to minimize potential errors (see Item 25: "Enforce Clarity with Keyword-Only and Positional-Only Arguments").

Things to Remember

- Function arguments can be specified by position or by keyword.
- Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
- Keyword arguments with default values make it easy to add new behaviors to a function without needing to migrate all existing callers.
- Optional keyword arguments should always be passed by keyword instead of by position.

Item 24: Use None and Docstrings to Specify Dynamic Default Arguments

Sometimes you need to use a non-static type as a keyword argument's default value. For example, say I want to print logging messages that are marked with the time of the logged event. In the default case, I want the message to include the time when the function was called. I might try the following approach, assuming that the default arguments are reevaluated each time the function is called:

```
from time import sleep
from datetime import datetime

def log(message, when=datetime.now()):
    print(f'{when}: {message}')

log('Hi there!')
sleep(0.1)
log('Hello again!')
```

```
>>>
2019-07-06 14:06:15.120124: Hi there!
2019-07-06 14:06:15.120124: Hello again!
```

This doesn't work as expected. The timestamps are the same because `datetime.now` is executed only a single time: when the function is defined. A default argument value is evaluated only once per module

load, which usually happens when a program starts up. After the module containing this code is loaded, the `datetime.now()` default argument will never be evaluated again.

The convention for achieving the desired result in Python is to provide a default value of `None` and to document the actual behavior in the docstring (see Item 84: "Write Docstrings for Every Function, Class, and Module" for background). When your code sees the argument value `None`, you allocate the default value accordingly:

```
def log(message, when=None):
    """Log a message with a timestamp.

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

Now the timestamps will be different:

```
log('Hi there!')
sleep(0.1)
log('Hello again!')
```

```
>>>
2019-07-06 14:06:15.222419: Hi there!
2019-07-06 14:06:15.322555: Hello again!
```

Using `None` for default argument values is especially important when the arguments are mutable. For example, say that I want to load a value encoded as JSON data; if decoding the data fails, I want an empty dictionary to be returned by default:

```
import json

def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

The problem here is the same as in the `datetime.now` example above. The dictionary specified for `default` will be shared by all calls to

decode because default argument values are evaluated only once (at module load time). This can cause extremely surprising behavior:

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
```

```
>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

You might expect two different dictionaries, each with a single key and value. But modifying one seems to also modify the other. The culprit is that foo and bar are both equal to the default parameter. They are the same dictionary object:

```
assert foo is bar
```

The fix is to set the keyword argument default value to None and then document the behavior in the function's docstring:

```
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
            Defaults to an empty dictionary.
    """
    try:
        return json.loads(data)
    except ValueError:
        if default is None:
            default = {}
        return default
```

Now, running the same test code as before produces the expected result:

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
assert foo is not bar
```

```
>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

This approach also works with type annotations (see Item 90: "Consider Static Analysis via `typing` to Obviate Bugs"). Here, the `when` argument is marked as having an `Optional` value that is a `datetime`. Thus, the only two valid choices for `when` are `None` or a `datetime` object:

```
from typing import Optional

def log_typed(message: str,
              when: Optional[datetime]=None) -> None:
    """Log a message with a timestamp.

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

Things to Remember

- A default argument value is evaluated only once: during function definition at module load time. This can cause odd behaviors for dynamic values (like `{}`, `[]`, or `datetime.now()`).
- Use `None` as the default value for any keyword argument that has a dynamic value. Document the actual default behavior in the function's docstring.
- Using `None` to represent keyword argument default values also works correctly with type annotations.

Item 25: Enforce Clarity with Keyword-Only and Positional-Only Arguments

Passing arguments by keyword is a powerful feature of Python functions (see Item 23: "Provide Optional Behavior with Keyword Arguments"). The flexibility of keyword arguments enables you to write functions that will be clear to new readers of your code for many use cases.

For example, say I want to divide one number by another but know that I need to be very careful about special cases. Sometimes, I want

to ignore ZeroDivisionError exceptions and return infinity instead. Other times, I want to ignore OverflowError exceptions and return zero instead:

```
def safe_division(number, divisor,
                  ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Using this function is straightforward. This call ignores the float overflow from division and returns zero:

```
result = safe_division(1.0, 10**500, True, False)
print(result)
```

```
>>>
0
```

This call ignores the error from dividing by zero and returns infinity:

```
result = safe_division(1.0, 0, False, True)
print(result)
```

```
>>>
inf
```

The problem is that it's easy to confuse the position of the two Boolean arguments that control the exception-ignoring behavior. This can easily cause bugs that are hard to track down. One way to improve the readability of this code is to use keyword arguments. By default, the function can be overly cautious and can always re-raise exceptions:

```
def safe_division_b(number, divisor,
                    ignore_overflow=False,        # Changed
                    ignore_zero_division=False):  # Changed
    ...
```

Then, callers can use keyword arguments to specify which of the ignore flags they want to set for specific operations, overriding the default behavior:

```
result = safe_division_b(1.0, 10**500, ignore_overflow=True)
print(result)

result = safe_division_b(1.0, 0, ignore_zero_division=True)
print(result)
```

```
>>>
0
inf
```

The problem is, since these keyword arguments are optional behavior, there's nothing forcing callers to use keyword arguments for clarity. Even with the new definition of `safe_division_b`, you can still call it the old way with positional arguments:

```
assert safe_division_b(1.0, 10**500, True, False) == 0
```

With complex functions like this, it's better to require that callers are clear about their intentions by defining functions with *keyword-only arguments*. These arguments can only be supplied by keyword, never by position.

Here, I redefine the `safe_division` function to accept keyword-only arguments. The `*` symbol in the argument list indicates the end of positional arguments and the beginning of keyword-only arguments:

```
def safe_division_c(number, divisor, *,  # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

Now, calling the function with positional arguments for the keyword arguments won't work:

```
safe_division_c(1.0, 10**500, True, False)
```

```
>>>
Traceback ...
TypeError: safe_division_c() takes 2 positional arguments but 4
➥were given
```

But keyword arguments and their default values will work as expected (ignoring an exception in one case and raising it in another):

```
result = safe_division_c(1.0, 0, ignore_zero_division=True)
assert result == float('inf')
```

```
try:
    result = safe_division_c(1.0, 0)
except ZeroDivisionError:
    pass  # Expected
```

However, a problem still remains with the `safe_division_c` version of this function: Callers may specify the first two required arguments (`number` and `divisor`) with a mix of positions and keywords:

```
assert safe_division_c(number=2, divisor=5) == 0.4
assert safe_division_c(divisor=5, number=2) == 0.4
assert safe_division_c(2, divisor=5) == 0.4
```

Later, I may decide to change the names of these first two arguments because of expanding needs or even just because my style preferences change:

```
def safe_division_c(numerator, denominator, *,  # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

Unfortunately, this seemingly superficial change breaks all the existing callers that specified the `number` or `divisor` arguments using keywords:

```
safe_division_c(number=2, divisor=5)
```

```
>>>
Traceback ...
TypeError: safe_division_c() got an unexpected keyword argument
➥'number'
```

This is especially problematic because I never intended for `number` and `divisor` to be part of an explicit interface for this function. These were just convenient parameter names that I chose for the implementation, and I didn't expect anyone to rely on them explicitly.

Python 3.8 introduces a solution to this problem, called *positional-only arguments*. These arguments can be supplied only by position and never by keyword (the opposite of the keyword-only arguments demonstrated above).

Here, I redefine the `safe_division` function to use positional-only arguments for the first two required parameters. The `/` symbol in the argument list indicates where positional-only arguments end:

```
def safe_division_d(numerator, denominator, /, *,  # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

I can verify that this function works when the required arguments are provided positionally:

```
assert safe_division_d(2, 5) == 0.4
```

But an exception is raised if keywords are used for the positional-only parameters:

```
safe_division_d(numerator=2, denominator=5)
```

```
>>>
Traceback ...
TypeError: safe_division_d() got some positional-only arguments
➥passed as keyword arguments: 'numerator, denominator'
```

Now, I can be sure that the first two required positional arguments in the definition of the `safe_division_d` function are decoupled from callers. I won't break anyone if I change the parameters' names again.

One notable consequence of keyword- and positional-only arguments is that any parameter name between the `/` and `*` symbols in the argument list may be passed either by position or by keyword (which is the default for all function arguments in Python). Depending on your API's style and needs, allowing both argument passing styles can increase readability and reduce noise. For example, here I've added another optional parameter to `safe_division` that allows callers to specify how many digits to use in rounding the result:

```
def safe_division_e(numerator, denominator, /,
                    ndigits=10, *,                # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    try:
        fraction = numerator / denominator        # Changed
        return round(fraction, ndigits)           # Changed
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Now, I can call this new version of the function in all these different ways, since `ndigits` is an optional parameter that may be passed either by position or by keyword:

```
result = safe_division_e(22, 7)
print(result)

result = safe_division_e(22, 7, 5)
print(result)

result = safe_division_e(22, 7, ndigits=2)
print(result)
```

```
>>>
3.1428571429
3.14286
3.14
```

Things to Remember

- Keyword-only arguments force callers to supply certain arguments by keyword (instead of by position), which makes the intention of a function call clearer. Keyword-only arguments are defined after a single * in the argument list.
- Positional-only arguments ensure that callers can't supply certain parameters using keywords, which helps reduce coupling. Positional-only arguments are defined before a single / in the argument list.
- Parameters between the / and * characters in the argument list may be supplied by position or keyword, which is the default for Python parameters.

Item 26: Define Function Decorators with `functools.wraps`

Python has special syntax for *decorators* that can be applied to functions. A decorator has the ability to run additional code before and after each call to a function it wraps. This means decorators can access and modify input arguments, return values, and raised exceptions. This functionality can be useful for enforcing semantics, debugging, registering functions, and more.

For example, say that I want to print the arguments and return value of a function call. This can be especially helpful when debugging

the stack of nested function calls from a recursive function. Here, I define such a decorator by using *args and **kwargs (see Item 22: "Reduce Visual Noise with Variable Positional Arguments" and Item 23: "Provide Optional Behavior with Keyword Arguments") to pass through all parameters to the wrapped function:

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f'{func.__name__}({args!r}, {kwargs!r}) '
              f'-> {result!r}')
        return result
    return wrapper
```

I can apply this decorator to a function by using the @ symbol:

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

Using the @ symbol is equivalent to calling the decorator on the function it wraps and assigning the return value to the original name in the same scope:

```
fibonacci = trace(fibonacci)
```

The decorated function runs the wrapper code before and after fibonacci runs. It prints the arguments and return value at each level in the recursive stack:

```
fibonacci(4)
```

```
>>>
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2
fibonacci((4,), {}) -> 3
```

This works well, but it has an unintended side effect. The value returned by the decorator—the function that's called above—doesn't think it's named `fibonacci`:

```
print(fibonacci)
```

```
>>>
<function trace.<locals>.wrapper at 0x108955dc0>
```

The cause of this isn't hard to see. The `trace` function returns the `wrapper` defined within its body. The `wrapper` function is what's assigned to the `fibonacci` name in the containing module because of the decorator. This behavior is problematic because it undermines tools that do introspection, such as debuggers (see Item 80: "Consider Interactive Debugging with `pdb`").

For example, the `help` built-in function is useless when called on the decorated `fibonacci` function. It should instead print out the docstring defined above (`'Return the n-th Fibonacci number'`):

```
help(fibonacci)
```

```
>>>
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

Object serializers (see Item 68: "Make `pickle` Reliable with `copyreg`") break because they can't determine the location of the original function that was decorated:

```
import pickle

pickle.dumps(fibonacci)
```

```
>>>
Traceback ...
AttributeError: Can't pickle local object 'trace.<locals>.
➥wrapper'
```

The solution is to use the `wraps` helper function from the `functools` built-in module. This is a decorator that helps you write decorators. When you apply it to the `wrapper` function, it copies all of the important metadata about the inner function to the outer function:

```
from functools import wraps
```

```
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ...
    return wrapper

@trace
def fibonacci(n):
    ...
```

Now, running the help function produces the expected result, even though the function is decorated:

```
help(fibonacci)
```

```
>>>
Help on function fibonacci in module __main__:

fibonacci(n)
    Return the n-th Fibonacci number
```

The pickle object serializer also works:

```
print(pickle.dumps(fibonacci))
```

```
>>>
b'\x80\x04\x95\x1a\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\
➥x94\x8c\tfibonacci\x94\x93\x94.'
```

Beyond these examples, Python functions have many other standard attributes (e.g., __name__, __module__, __annotations__) that must be preserved to maintain the interface of functions in the language. Using wraps ensures that you'll always get the correct behavior.

Things to Remember

- Decorators in Python are syntax to allow one function to modify another function at runtime.
- Using decorators can cause strange behaviors in tools that do introspection, such as debuggers.
- Use the wraps decorator from the functools built-in module when you define your own decorators to avoid issues.

This page intentionally left blank

4 Comprehensions and Generators

Many programs are built around processing lists, dictionary key/value pairs, and sets. Python provides a special syntax, called *comprehensions*, for succinctly iterating through these types and creating derivative data structures. Comprehensions can significantly increase the readability of code performing these common tasks and provide a number of other benefits.

This style of processing is extended to functions with *generators*, which enable a stream of values to be incrementally returned by a function. The result of a call to a generator function can be used anywhere an iterator is appropriate (e.g., `for` loops, starred expressions). Generators can improve performance, reduce memory usage, and increase readability.

Item 27: Use Comprehensions Instead of `map` and `filter`

Python provides compact syntax for deriving a new `list` from another sequence or iterable. These expressions are called *list comprehensions*. For example, say that I want to compute the square of each number in a `list`. Here, I do this by using a simple `for` loop:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []
for x in a:
    squares.append(x**2)
print(squares)
```

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```