-
- Python Cookbook

- Comments Off
- Chapters

Table of Contents

- Log In / Sign Up
- Search book...

Chapter 7. Functions

Defining functions using the `def` statement is a cornerstone of all programs. The goal of this chapter is to present some more advanced and unusual function definition and usage patterns. Topics include default arguments, functions that take any number of arguments, keyword-only arguments, annotations, and closures. In addition, some tricky control flow and data passing problems involving callback functions are addressed.

Writing Functions That Accept Any Number of Arguments

Problem

You want to write a function that accepts any number of input arguments.

Solution

To write a function that accepts any number of positional arguments, use a * argument. For example:

```
def avg(first, *rest):
    return (first + sum(rest)) / (1 + len(rest))

# Sample use
avg(1, 2)          # 1.5
avg(1, 2, 3, 4)    # 2.5
```

In this example, `rest` is a tuple of all the extra positional arguments passed. The code treats it as a sequence in performing subsequent calculations.

To accept any number of keyword arguments, use an argument that starts with **. For example:

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
                  name=name,
                  attrs=attr_str,
                  value=html.escape(value))
    return element

# Example
# Creates '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Creates '<p>&lt;spam&gt;</p>'
make_element('p', '<spam>')
```

Here, `attrs` is a dictionary that holds the passed keyword arguments (if any).

If you want a function that can accept both any number of positional and keyword-only arguments, use * and ** together. For example:

```
def anyargs(*args, **kwargs):
    print(args)      # A tuple
    print(kwargs)    # A dict
```

With this function, all of the positional arguments are placed into a tuple `args`, and all of the keyword arguments are placed into a dictionary `kwargs`.

Discussion

A * argument can only appear as the last positional argument in a function definition. A ** argument can only appear as the last argument. A subtle aspect of function definitions is that arguments can still appear after a * argument.

```
def a(x, *args, y):
    pass

def b(x, *args, y, **kwargs):
    pass
```

Such arguments are known as keyword-only arguments, and are discussed further in "Writing Functions That Only Accept Keyword Arguments".

Writing Functions That Only Accept Keyword Arguments

Problem

You want a function to only accept certain arguments by keyword.

Solution

This feature is easy to implement if you place the keyword arguments after a * argument or a single unnamed *. For example:

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True)        #  TypeError
recv(1024, block=True)  # Ok
```

This technique can also be used to specify keyword arguments for functions that accept a varying number of positional arguments. For example:

```
def mininum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

minimum(1, 5, 2, -5, 10)          # Returns -5
minimum(1, 5, 2, -5, 10, clip=0)  # Returns 0
```

Discussion

Keyword-only arguments are often a good way to enforce greater code clarity when specifying optional function arguments. For example, consider a call like this:

```
msg = recv(1024, False)
```

If someone is not intimately familiar with the workings of the `recv()`, they may have no idea what the `False` argument means. On the other hand, it is much clearer if the call is written like this:

```
msg = recv(1024, block=False)
```

The use of keyword-only arguments is also often preferrable to tricks involving `**kwargs`, since they show up properly when the user asks for help:

```
>>> help(recv)
Help on function recv in module __main__:

recv(maxsize, *, block)
    Receives a message
```

Keyword-only arguments also have utility in more advanced contexts. For example, they can be used to inject arguments into functions that make use of the `*args` and `**kwargs` convention for accepting all inputs. See "Writing Decorators That Add Arguments to Wrapped Functions" for an example.

Attaching Informational Metadata to Function Arguments

Problem

You've written a function, but would like to attach some additional information to the arguments so that others know more about how a function is supposed to be used.

Solution

Function argument annotations can be a useful way to give programmers hints about how a function is supposed to be used. For example, consider the following annotated function:

```
def add(x:int, y:int) -> int:
    return x + y
```

The Python interpreter does not attach any semantic meaning to the attached annotations. They are not type checks, nor do they make Python behave any differently than it did before. However, they might give useful hints to others reading the source code about what you had in mind. Third-party tools and frameworks might also attach semantic meaning to the annotations. They also appear in documentation:

```
>>> help(add)
Help on function add in module __main__:

add(x: int, y: int) -> int
>>>
```

Although you can attach any kind of object to a function as an annotation (e.g., numbers, strings, instances, etc.), classes or strings often seem to make the most sense.

Discussion

Function annotations are merely stored in a function’s `__annotations__` attribute. For example:

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

Although there are many potential uses of annotations, their primary utility is probably just documentation. Because Python doesn’t have type declarations, it can often be difficult to know what you’re supposed to pass into a function if you’re simply reading its source code in isolation. An annotation gives someone more of a hint.

See “Implementing Multiple Dispatch with Function Annotations” for an advanced example showing how to use annotations to implement multiple dispatch (i.e., overloaded functions).

Returning Multiple Values from a Function

Problem

You want to return multiple values from a function.

Solution

To return multiple values from a function, simply return a tuple. For example:

```
>>> def myfun():
...     return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

Discussion

Although it looks like `myfun()` returns multiple values, a tuple is actually being created. It looks a bit peculiar, but it’s actually the comma that forms a tuple, not the parentheses. For example:

```
>>> a = (1, 2)     # With parentheses
>>> a
(1, 2)
>>> b = 1, 2       # Without parentheses
>>> b
(1, 2)
>>>
```

When calling functions that return a tuple, it is common to assign the result to multiple variables, as shown. This is simply tuple unpacking, as described in "Unpacking a Sequence into Separate Variables". The return value could also have been assigned to a single variable:

```
>>> x = myfun()
>>> x
(1, 2, 3)
>>>
```

Defining Functions with Default Arguments

Problem

You want to define a function or method where one or more of the arguments are optional and have a default value.

Solution

On the surface, defining a function with optional arguments is easy—simply assign values in the definition and make sure that default arguments appear last. For example:

```
def spam(a, b=42):
    print(a, b)

spam(1)      # Ok. a=1, b=42
spam(1, 2)   # Ok. a=1, b=2
```

If the default value is supposed to be a mutable container, such as a list, set, or dictionary, use `None` as the default and write code like this:

```
# Using a list as a default value
def spam(a, b=None):
    if b is None:
        b = []
    ...
```

If, instead of providing a default value, you want to write code that merely tests whether an optional argument was given an interesting value or not, use this idiom:

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...
```

Here's how this function behaves:

```
>>> spam(1)
No b value supplied
>>> spam(1, 2)     # b = 2
>>> spam(1, None)  # b = None
>>>
```

Carefully observe that there is a distinction between passing no value at all and passing a value of `None`.

Discussion

Defining functions with default arguments is easy, but there is a bit more to it than meets the eye.

First, the values assigned as a default are bound only once at the time of function definition. Try this example to see it:

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42
>>> x = 23     # Has no effect
>>> spam(1)
1 42
>>>
```

Notice how changing the variable `x` (which was used as a default value) has no effect whatsoever. This is because the default value was fixed at function definition time.

Second, the values assigned as defaults should always be immutable objects, such as `None`, `True`, `False`, numbers, or strings. Specifically, never write code like this:

```
def spam(a, b=[]):     # NO!
    ...
```

If you do this, you can run into all sorts of trouble if the default value ever escapes the function and gets modified. Such changes will permanently alter the default value across future function calls. For example:

```
>>> def spam(a, b=[]):
...     print(b)
...     return b
...
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1)       # Modified list gets returned!
```

```
[99, 'Yow!']
>>>
```

That's probably not what you want. To avoid this, it's better to assign `None` as a default and add a check inside the function for it, as shown in the solution.

The use of the `is` operator when testing for `None` is a critical part of this recipe. Sometimes people make this mistake:

```
def spam(a, b=None):
    if not b:      # NO! Use 'b is None' instead
        b = []
    ...
```

The problem here is that although `None` evaluates to `False`, many other objects (e.g., zero-length strings, lists, tuples, dicts, etc.) do as well. Thus, the test just shown would falsely treat certain inputs as missing. For example:

```
>>> spam(1)        # OK
>>> x = []
>>> spam(1, x)     # Silent error. x value overwritten by default
>>> spam(1, 0)     # Silent error. 0 ignored
>>> spam(1, '')    # Silent error. '' ignored
>>>
```

The last part of this recipe is something that's rather subtle—a function that tests to see whether a value (any value) has been supplied to an optional argument or not. The tricky part here is that you can't use a default value of `None`, `0`, or `False` to test for the presence of a user-supplied argument (since all of these are perfectly valid values that a user might supply). Thus, you need something else to test against.

To solve this problem, you can create a unique private instance of `object`, as shown in the solution (the `_no_value` variable). In the function, you then check the identity of the supplied argument against this special value to see if an argument was supplied or not. The thinking here is that it would be extremely unlikely for a user to pass the `_no_value` instance in as an input value. Therefore, it becomes a safe value to check against if you're trying to determine whether an argument was supplied or not.

The use of `object()` might look rather unusual here. `object` is a class that serves as the common base class for almost all objects in Python. You can create instances of `object`, but they are wholly uninteresting, as they have no notable methods nor any instance data (because there is no underlying instance dictionary, you can't even set any attributes). About the only thing you can do is perform tests for identity. This makes them useful as special values, as shown in the solution.

Defining Anonymous or Inline Functions

Problem

You need to supply a short callback function for use with an operation such as `sort()`, but you don’t want to write a separate one-line function using the `def` statement. Instead, you’d like a shortcut that allows you to specify the function "in line."

Solution

Simple functions that do nothing more than evaluate an expression can be replaced by a `lambda` expression. For example:

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

The use of `lambda` here is the same as having typed this:

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

Typically, `lambda` is used in the context of some other operation, such as sorting or a data reduction:

```
>>> names = ['David Beazley', 'Brian Jones',
...          'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

Discussion

Although `lambda` allows you to define a simple function, its use is highly restricted. In particular, only a single expression can be specified, the result of which is the return value. This means that no other language features, including multiple statements, conditionals, iteration, and exception handling, can be included.

You can quite happily write a lot of Python code without ever using lambda. However, you’ll occasionally encounter it in programs where someone is writing a lot of tiny functions that evaluate various expressions, or in programs that require users to supply callback functions.

Capturing Variables in Anonymous Functions

Problem

You’ve defined an anonymous function using `lambda`, but you also need to capture the values of certain variables at the time of definition.

Solution

Consider the behavior of the following code:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

Now ask yourself a question. What are the values of `a(10)` and `b(10)`? If you think the results might be 20 and 30, you would be wrong:

```
>>> a(10)
30
>>> b(10)
30
>>>
```

The problem here is that the value of `x` used in the `lambda` expression is a free variable that gets bound at runtime, not definition time. Thus, the value of `x` in the lambda expressions is whatever the value of the `x` variable happens to be at the time of execution. For example:

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

If you want an anonymous function to capture a value at the point of definition and keep it, include the value as a default value, like this:

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

Discussion

The problem addressed in this recipe is something that tends to come up in code that tries to be just a little bit too clever with the use of lambda functions. For example, creating a list of lambda expressions using a list comprehension or in a loop of some kind and expecting the lambda functions to remember the iteration variable at the time of definition. For example:

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
4
4
4
4
4
>>>
```

Notice how all functions think that `n` has the last value during iteration. Now compare to the following:

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
0
1
2
3
4
>>>
```

As you can see, the functions now capture the value of `n` at the time of definition.

Making an N-Argument Callable Work As a Callable with Fewer Arguments

Problem

You have a callable that you would like to use with some other Python code, possibly as a callback function or handler, but it takes too many arguments and causes an exception when called.

Solution

If you need to reduce the number of arguments to a function, you should use `functools.partial()`. The `partial()` function allows you to assign fixed values to one or more of the arguments, thus reducing the number of arguments that need to be supplied to subsequent calls. To illustrate, suppose you have this function:

```
def spam(a, b, c, d):
    print(a, b, c, d)
```

Now consider the use of `partial()` to fix certain argument values:

```
>>> from functools import partial
>>> s1 = partial(spam, 1)       # a = 1
>>> s1(2, 3, 4)
```

```
1 2 3 4
>>> s1(4, 5, 6)
1 4 5 6
>>> s2 = partial(spam, d=42)    # d = 42
>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42
>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
>>>
```

Observe that `partial()` fixes the values for certain arguments and returns a new callable as a result. This new callable accepts the still unassigned arguments, combines them with the arguments given to `partial()`, and passes everything to the original function.

Discussion

This recipe is really related to the problem of making seemingly incompatible bits of code work together. A series of examples will help illustrate.

As a first example, suppose you have a list of points represented as tuples of (x,y) coordinates. You could use the following function to compute the distance between two points:

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

Now suppose you want to sort all of the points according to their distance from some other point. The `sort()` method of lists accepts a `key` argument that can be used to customize sorting, but it only works with functions that take a single argument (thus, `distance()` is not suitable). Here's how you might use `partial()` to fix it:

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

As an extension of this idea, `partial()` can often be used to tweak the argument signatures of callback functions used in other libraries. For example, here's a bit of code that uses `multiprocessing` to asynchronously compute a result which is handed to a callback function that accepts both the result and an optional logging argument:

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# A sample function
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

When supplying the callback function using `apply_async()`, the extra logging argument is given using `partial()`. `multiprocessing` is none the wiser about all of this—it simply invokes the callback function with a single value.

As a similar example, consider the problem of writing network servers. The `socketserver` module makes it relatively easy. For example, here is a simple echo server:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

However, suppose you want to give the `EchoHandler` class an `__init__()` method that accepts an additional configuration argument. For example:

```
class EchoHandler(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are
    # any normal parameters supplied (which are passed on)
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)
    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)
```

If you make this change, you'll find there is no longer an obvious way to plug it into the

TCPServer class. In fact, you'll find that the code now starts generating exceptions like this:

```
Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
 ...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'
```

At first glance, it seems impossible to fix this code, short of modifying the source code to socketserver or coming up with some kind of weird workaround. However, it's easy to resolve using partial()—just use it to supply the value of the ack argument, like this:

```
from functools import partial
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVED:'))
serv.serve_forever()
```

In this example, the specification of the ack argument in the __init__() method might look a little funny, but it's being specified as a keyword-only argument. This is discussed further in "Writing Functions That Only Accept Keyword Arguments".

The functionality of partial() is sometimes replaced with a lambda expression. For example, the previous examples might use statements such as this:

```
points.sort(key=lambda p: distance(pt, p))

p.apply_async(add, (3, 4), callback=lambda result: output_result(result,log))

serv = TCPServer(('', 15000),
                 lambda *args, **kwargs: EchoHandler(*args,
                                                     ack=b'RECEIVED:',
                                                     **kwargs))
```

This code works, but it's more verbose and potentially a lot more confusing to someone reading it. Using partial() is a bit more explicit about your intentions (supplying values for some of the arguments).)

Replacing Single Method Classes with Functions

Problem

You have a class that only defines a single method besides __init__(). However, to simplify your code, you would much rather just have a simple function.

Solution

In many cases, single-method classes can be turned into functions using closures. Consider, as an example, the following class, which allows a user to fetch URLs using a kind of templating scheme.

```
from urllib.request import urlopen
```

```
class UrlTemplate:
    def __init__(self, template):
        self.template = template
    def open(self, **kwargs):
        return urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

The class could be replaced with a much simpler function:

```
def urltemplate(template):
    def opener(**kwargs):
        return urlopen(template.format_map(kwargs))
    return opener

# Example use
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

Discussion

In many cases, the only reason you might have a single-method class is to store additional state for use in the method. For example, the only purpose of the `UrlTemplate` class is to hold the `template` value someplace so that it can be used in the `open()` method.

Using an inner function or closure, as shown in the solution, is often more elegant. Simply stated, a closure is just a function, but with an extra environment of the variables that are used inside the function. A key feature of a closure is that it remembers the environment in which it was defined. Thus, in the solution, the `opener()` function remembers the value of the `template` argument, and uses it in subsequent calls.

Whenever you're writing code and you encounter the problem of attaching additional state to a function, think closures. They are often a more minimal and elegant solution than the alternative of turning your function into a full-fledged class.

Carrying Extra State with Callback Functions

Problem

You're writing code that relies on the use of callback functions (e.g., event handlers, completion callbacks, etc.), but you want to have the callback function carry extra state for use inside the callback function.

Solution

This recipe pertains to the use of callback functions that are found in many libraries and frameworks—especially those related to asynchronous processing. To illustrate and for the purposes of testing, define the following function, which invokes a callback:

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

In reality, such code might do all sorts of advanced processing involving threads, processes, and timers, but that's not the main focus here. Instead, we're simply focused on the invocation of the callback. Here's an example that shows how the preceding code gets used:

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

As you will notice, the `print_result()` function only accepts a single argument, which is the result. No other information is passed in. This lack of information can sometimes present problems when you want the callback to interact with other variables or parts of the environment.

One way to carry extra information in a callback is to use a bound-method instead of a simple function. For example, this class keeps an internal sequence number that is incremented every time a result is received:

```
class ResultHandler:
    def __init__(self):
        self.sequence = 0
    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

To use this class, you would create an instance and use the bound method `handler` as the callback:

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

As an alternative to a class, you can also use a closure to capture state. For example:

```
def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler
```

Here is an example of this variant:

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

As yet another variation on this theme, you can sometimes use a coroutine to accomplish the same thing:

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

For a coroutine, you would use its `send()` method as the callback, like this:

```
>>> handler = make_handler()
>>> next(handler)       # Advance to the yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

Last, but not least, you can also carry state into a callback using an extra argument and partial function application. For example:

```
>>> class SequenceNo:
...     def __init__(self):
...         self.sequence = 0
...
>>> def handler(result, seq):
...     seq.sequence += 1
...     print('[{}] Got: {}'.format(seq.sequence, result))
...
>>> seq = SequenceNo()
>>> from functools import partial
>>> apply_async(add, (2, 3), callback=partial(handler, seq=seq))
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=partial(handler, seq=seq))
[2] Got: helloworld
```

```
>>>
```

Discussion

Software based on callback functions often runs the risk of turning into a huge tangled mess. Part of the issue is that the callback function is often disconnected from the code that made the initial request leading to callback execution. Thus, the execution environment between making the request and handling the result is effectively lost. If you want the callback function to continue with a procedure involving multiple steps, you have to figure out how to save and restore the associated state.

There are really two main approaches that are useful for capturing and carrying state. You can carry it around on an instance (attached to a bound method perhaps) or you can carry it around in a closure (an inner function). Of the two techniques, closures are perhaps a bit more lightweight and natural in that they are simply built from functions. They also automatically capture all of the variables being used. Thus, it frees you from having to worry about the exact state needs to be stored (it's determined automatically from your code).

If using closures, you need to pay careful attention to mutable variables. In the solution, the `nonlocal` declaration is used to indicate that the `sequence` variable is being modified from within the callback. Without this declaration, you'll get an error.

The use of a coroutine as a callback handler is interesting in that it is closely related to the closure approach. In some sense, it's even cleaner, since there is just a single function. Moreover, variables can be freely modified without worrying about `nonlocal` declarations. The potential downside is that coroutines don't tend to be as well understood as other parts of Python. There are also a few tricky bits such as the need to call `next()` on a coroutine prior to using it. That's something that could be easy to forget in practice. Nevertheless, coroutines have other potential uses here, such as the definition of an inlined callback (covered in the next recipe).

The last technique involving `partial()` is useful if all you need to do is pass extra values into a callback. Instead of using `partial()`, you'll sometimes see the same thing accomplished with the use of a `lambda`:

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

For more examples, see "Making an N-Argument Callable Work As a Callable with Fewer Arguments", which shows how to use `partial()` to change argument signatures.

Inlining Callback Functions

Problem

You're writing code that uses callback functions, but you're concerned about the proliferation of small functions and mind boggling control flow. You would like some way to make the code look more like a normal sequence of procedural steps.

Solution

Callback functions can be inlined into a function using generators and coroutines. To illustrate, suppose you have a function that performs work and invokes a callback as follows (see "Carrying Extra State with Callback Functions"):

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

Now take a look at the following supporting code, which involves an `Async` class and an `inlined_async` decorator:

```
from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
        self.args = args

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
    return wrapper
```

These two fragments of code will allow you to inline the callback steps using `yield` statements. For example:

```
def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')
```

If you call `test()`, you’ll get output like this:

```
5
helloworld
0
2
4
6
8
10
12
14
16
18
Goodbye
```

Aside from the special decorator and use of `yield`, you will notice that no callback functions appear anywhere (except behind the scenes).

Discussion

This recipe will really test your knowledge of callback functions, generators, and control flow.

First, in code involving callbacks, the whole point is that the current calculation will suspend and resume at some later point in time (e.g., asynchronously). When the calculation resumes, the callback will get executed to continue the processing. The `apply_async()` function illustrates the essential parts of executing the callback, although in reality it might be much more complicated (involving threads, processes, event handlers, etc.).

The idea that a calculation will suspend and resume naturally maps to the execution model of a generator function. Specifically, the `yield` operation makes a generator function emit a value and suspend. Subsequent calls to the `__next__()` or `send()` method of a generator will make it start again.

With this in mind, the core of this recipe is found in the `inline_async()` decorator function. The key idea is that the decorator will step the generator function through all of its `yield` statements, one at a time. To do this, a result queue is created and initially populated with a value of `None`. A loop is then initiated in which a result is popped off the queue and sent into the generator. This advances to the next yield, at which point an instance of `Async` is received. The loop then looks at the function and arguments, and initiates the asynchronous calculation `apply_async()`. However, the sneakiest part of this calculation is that instead of using a normal callback function, the callback is set to the queue `put()` method.

At this point, it is left somewhat open as to precisely what happens. The main loop immediately goes back to the top and simply executes a `get()` operation on the queue. If data is present, it must be the result placed there by the `put()` callback. If nothing is there, the operation blocks, waiting for a result to arrive at some future time. How that might happen depends on the precise implementation of the `apply_async()` function.

If you're doubtful that anything this crazy would work, you can try it with the multiprocessing library and have async operations executed in separate processes:

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

Indeed, you'll find that it works, but unraveling the control flow might require more coffee.

Hiding tricky control flow behind generator functions is found elsewhere in the standard library and third-party packages. For example, the `@contextmanager` decorator in the `contextlib` performs a similar mind-bending trick that glues the entry and exit from a context manager together across a `yield` statement. The popular Twisted package has inlined callbacks that are also similar.

Accessing Variables Defined Inside a Closure

Problem

You would like to extend a closure with functions that allow the inner variables to be accessed and modified.

Solution

Normally, the inner variables of a closure are completely hidden to the outside world. However, you can provide access by writing accessor functions and attaching them to the closure as function attributes. For example:

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value
    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

Here is an example of using this code:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```

Discussion

There are two main features that make this recipe work. First, `nonlocal` declarations make it possible to write functions that change inner variables. Second, function attributes allow the accessor methods to be attached to the closure function in a straightforward manner where they work a lot like instance methods (even though no class is involved).

A slight extension to this recipe can be made to have closures emulate instances of a class. All you need to do is copy the inner functions over to the dictionary of an instance and return it. For example:

```
import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )
    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()

# Example use
def Stack():
    items = []

    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)
    return ClosureInstance()
```

Here’s an interactive session to show that it actually works:

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
>>> s.pop()
20
>>> s.pop()
10
>>>
```

Interestingly, this code runs a bit faster than using a normal class definition. For example, you might be inclined to test the performance against a class like this:

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

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

If you do, you’ll get results similar to the following:

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

As shown, the closure version runs about 8% faster. Most of that is coming from streamlined access to the instance variables. Closures are faster because there’s no extra `self` variable involved.

Raymond Hettinger has devised an even more diabolical variant of this idea. However, should you be inclined to do something like this in your code, be aware that it's still a rather weird substitute for a real class. For example, major features such as inheritance, properties, descriptors, or class methods don't work. You also have to play some tricks to get special methods to work (e.g., see the implementation of `__len__()` in `ClosureInstance`).

Lastly, you'll run the risk of confusing people who read your code and wonder why it doesn't look anything like a normal class definition (of course, they'll also wonder why it's faster). Nevertheless, it's an interesting example of what can be done by providing access to the internals of a closure.

In the big picture, adding methods to closures might have more utility in settings where you want to do things like reset the internal state, flush buffers, clear caches, or have some kind of feedback mechanism.

Prev
Chapter 6. Data Encoding and Processing

Home

Next
Chapter 8. Classes and Objects