Python in a Functional Style: Closures, Generators, and Coroutines

Jifeng Wu

2022-10-28

Contents

- Closures 👈
- Generators
- Coroutines

Closures

All Python functions are **closures**.

- Function code.
- Execution environment of function code (variables it depend on).

A nested function can be returned. This is a common design pattern for creating **tailored functions**.

```python
def get_greeting_function(name):
    def greeting_function():
        print(f'Hello, {name}')
    return greeting_function
```

Closures

All Python functions are **closures**.

- Function code.
- Execution environment of function code (variables it depend on).

A nested function can be returned. This is a common design pattern for creating **tailored functions**.

```
>>> function_greeting_a = get_greeting_function('A')
>>> function_greeting_a()
Hello, A
>>>
>>> function_greeting_b = get_greeting_function('B')
>>> function_greeting_b()
Hello, B
```

Closures

Look into a closure's `cell_contents`:

```
>>> function_greeting_a.__closure__
(<cell at 0x7f3c81849ca8: str object at 0x7f3c8185ac70>,)
>>> function_greeting_a.__closure__[0]
<cell at 0x7f3c81849ca8: str object at 0x7f3c8185ac70>
>>> function_greeting_a.__closure__[0].cell_contents
'A'
>>>
>>> function_greeting_b.__closure__
(<cell at 0x7f3c81849c18: str object at 0x7f3c82f18e30>,)
>>> function_greeting_b.__closure__[0]
<cell at 0x7f3c81849c18: str object at 0x7f3c82f18e30>
>>> function_greeting_b.__closure__[0].cell_contents
'B'
```

Closures

Should an inner function **use an outer function's local variable** (instead of **shadowing it**), that local variable should be declared `nonlocal` within the inner function. Not using `nonlocal`:

```
def outer_function():
    string = 'Hello'
    def inner_function():
        # Shadows the local variable `string` of `outer_function`
        string = 'World'
    inner_function()
    return string
```

```
>>> outer_function()
'Hello'
```

Closures

Should an inner function **use an outer function's local variable** (instead of **shadowing it**), that local variable should be declared `nonlocal` within the inner function. Using `nonlocal`:

```
def outer_function():
    string = 'Hello'
    def inner_function():
        # Uses the local variable `string` of `outer_function`
        nonlocal string
        string = 'World'
    inner_function()
    return string
```

```
>>> outer_function()
'World'
```

Closures

Creating and returning a nested function based on a function argument is widely used in Python, called **decorating a function**.

```python
def cached(function):
    cache = {}
    def cached_function(*args):
        nonlocal function, cache
        if args in cache:
            print(f'Cache hit with args: {args}')
            return cache[args]
        else:
            print(f'Cache miss with args: {args}')
            result = function(*args)
            print(f'Writing f({args}) => {result} to cache')
            cache[args] = result
            return result
    return cached_function
```

Closures

Python even has special syntatical support for this.

```
@cached
def fib(n):
    if n < 1:
        return 0
    elif n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Closures

```
In [4]: fib(5)
Cache miss with args: (5,)
Cache miss with args: (4,)
Cache miss with args: (3,)
Cache miss with args: (2,)
Cache miss with args: (1,)
Writing f((1,)) => 1 to cache
Cache miss with args: (0,)
Writing f((0,)) => 0 to cache
Writing f((2,)) => 1 to cache
Cache hit with args: (1,)
Writing f((3,)) => 2 to cache
Cache hit with args: (2,)
Writing f((4,)) => 3 to cache
Cache hit with args: (3,)
Writing f((5,)) => 5 to cache
Out[4]: 5
```

$O(n)$ time complexity.

Closures

LeetCode problem: Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

mdaines.github.io/grammophone/

Edit | Transform | Analyze

```
# Type a grammar here:

S -> S S' | S' .
S' -> ( S ) | ( ) .
```

Analysis

Sanity Checks

- All nonterminals are reachable.
- All nonterminals are realizable.
- The grammar contains no cycles.
- The grammar is null unambiguous.

Example Sentences

- ( )
- (( ))
- ( ) ( )
- ((( )))
- ( ) (( ))
- (( ) ( ))
- (( )) ( )
- ( ) ( ) ( )
- (((( ))))
- ( ) ((( )))

More example sentences

Nonterminals

Symbol	Nullable?	Endable?	First set	Follow set
S		Endable	(	(,), $
S'		Endable	(	(,), $

Parsing Algorithms

LL(1)	Not LL(1) — it contains a first set clash.	Parsing table
LR(0)	Not LR(0) — it contains a shift-reduce conflict.	Automaton, Parsing table
SLR(1)	The grammar is SLR(1).	Parsing table
LR(1)	The grammar is LR(1).	Automaton, Parsing table
LALR(1)	The grammar is LALR(1).	Automaton, Parsing table

Closures

We write a Context Free Grammar and analyze it:

```
S -> S S' | S' .
S' -> ( S ) | ( ) .
```

https://mdaines.github.io/grammophone/#

Closures

```python
@cached
def s_generator(number_of_parenthesis):
    print(f's_generator({number_of_parenthesis})')

    return_value = []

    # s -> ss .
    if number_of_parenthesis >= 1:
        for ss_string in ss_generator(number_of_parenthesis):
            return_value.append(ss_string)

    # s -> s ss .
    if number_of_parenthesis >= 2:
        for i in range(1, number_of_parenthesis):
            for s_string, ss_string in itertools.product(
                s_generator(i),
                ss_generator(number_of_parenthesis - i)
            ):
                return_value.append(s_string + ss_string)

    return return_value
```

Closures

```
@cached
def ss_generator(number_of_parenthesis):
    print(f'ss_generator({number_of_parenthesis})')

    return_value = []

    # ss -> ( ) .
    if number_of_parenthesis == 1:
        return_value.append('()')
    # ss -> ( s ) .
    if number_of_parenthesis > 1:
        for s_string in s_generator(number_of_parenthesis - 1):
            return_value.append('(' + s_string + ')')

    return return_value
```

Closures

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

```
In [4]: s_generator(3)
s_generator(3)
ss_generator(3)
s_generator(2)
ss_generator(2)
s_generator(1)
ss_generator(1)
Out[4]: ['((()))', '(()())', '()(())', '(())()', '()()()']

In [5]: s_generator.cache_info()
Out[5]: CacheInfo(hits=3, misses=3, maxsize=None, currsize=3)

In [6]: ss_generator.cache_info()
Out[6]: CacheInfo(hits=3, misses=3, maxsize=None, currsize=3)
```

Closures

Closures also provide an efficient mechanism for **maintaining state between several calls**. Traditional (OOP) approach:

```
class Countdown:
    def __init__(self, n):
        self.n = n

    def next_value(self):
        old_value = self.n
        self.n -= 1
        return old_value
```

Closures

Closure-based approach:

```python
def countdown(n):
    def get_next_value():
        nonlocal n
        old_value = n
        n -= 1
        return old_value

    return get_next_value
```

Closures

This is not only clean but also **fast**.

```
def test_object_oriented_approach():
    c = Countdown(1_000_000)
    while True:
        value = c.next_value()
        if value == 0:
            break

def test_functional_approach():
    get_next_value = countdown(1_000_000)
    while True:
        value = get_next_value()
        if value == 0:
            break
```

Closures

```
In [5]: %timeit test_object_oriented_approach()
182 ms ± 2.61 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [6]: %timeit test_functional_approach()
96.8 ms ± 1.18 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Why?

Closures

```
In [9]: c = Countdown(1_000_000)
In [10]: dis(c.next_value)
  6           0 LOAD_FAST                0 (self)
              2 LOAD_ATTR                0 (n)
              4 STORE_FAST               1 (old_value)

  7           6 LOAD_FAST                0 (self)
              8 DUP_TOP
             10 LOAD_ATTR                0 (n)
             12 LOAD_CONST               1 (1)
             14 INPLACE_SUBTRACT
             16 ROT_TWO
             18 STORE_ATTR               0 (n)

  8          20 LOAD_FAST                1 (old_value)
             22 RETURN_VALUE
```

12 instructions, 2 `LOAD_ATTR` instructions, 1 `STORE_ATTR` instruction.

Closures

```
In [11]: get_next_value = countdown(1_000_000)
In [12]: dis(get_next_value)
  4           0 LOAD_DEREF               0 (n)
              2 STORE_FAST               0 (old_value)

  5           4 LOAD_DEREF               0 (n)
              6 LOAD_CONST               1 (1)
              8 INPLACE_SUBTRACT
             10 STORE_DEREF              0 (n)

  6          12 LOAD_FAST                0 (old_value)
             14 RETURN_VALUE
```

8 instructions, NO `LOAD_ATTR`, `STORE_ATTR` instructions.

Contents

- Closures
- Generators👈
- Coroutines

Generators

When we define a function containing the `yield` keyword, we define a generator. Defining a generator allows the user to define a **custom iterator** in the style of defining a function.

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

Generators

We create a **generator object** when we call a generator definition. The generator object can be used like any iterator:

```
In [2]: c = countdown(5)

In [3]: next(c)
Out[3]: 5

In [4]: next(c)
Out[4]: 4

In [5]: for value in c:
   ...:     print(value)
   ...:
3
2
1
```

Generators

When we call `next()` on a generator object, it will execute code, until it encounters a `yield` statement. The `yield` statement tells the generator object to **return a value, and continue execution from here when `next()` is called again**.

```
In [2]: c = countdown(5)

In [3]: next(c)
Out[3]: 5
```

This executes:

```
    while n > 0:
        yield n
```

Generators

When we call `next()` on a generator object, it will execute code, until it encounters a `yield` statement. The `yield` statement tells the generator object to **return a value, and continue execution from here when `next()` is called again**.

```
In [4]: next(c)
Out[4]: 4
```

This executes:

```
        n -= 1
    while n > 0:
        yield n
```

Generators

This is called **lazy evaluation**. This can dramatically boost performance and reduce memory usage in some applications. For example:

```
def get_comments_from_file(file):
    with open(file, 'r') as fp:
        for line in fp:
            # strip whitespace
            stripped_line = line.strip()
            # check if the line is empty after stripping whitespace
            if stripped_line:
                # check if the line is a comment
                if stripped_line[0] == '#':
                    # if it is, yield it
                    yield stripped_line
```

Generators

This will **NOT** read the whole file into memory. Only when the user calls `next()` on the generator object, will the generator read the file **LINE BY LINE** (with only **ONE LINE** of the file in memory at once), and return the next comment line.

This is an efficient way of extracting comments from GB-sized files (such as logs).

itertools

Python provides many functions for creating an iterator from another iterator. For example:

- `itertools.permutations(iterable [, r])`
- `itertools.combinations(iterable, r)`
- `itertools.product(iter1, iter2, iterN, [repeat=1])`

Generators

Widely used in algorithms: `itertools.permutations(iterable [,r])`

```
In [1]: import itertools


In [2]: numbers = range(4)

In [3]: permutations_of_two_numbers_iterator = itertools.permutations(numbers, r=2)

In [4]: next(permutations_of_two_numbers_iterator)
Out[4]: (0, 1)

In [5]: next(permutations_of_two_numbers_iterator)
Out[5]: (0, 2)

In [6]: next(permutations_of_two_numbers_iterator)
Out[6]: (0, 3)
```

Generators

Widely used in algorithms: `itertools.combinations(iterable ,r)`

```
In [1]: import itertools

In [2]: numbers = range(4)

In [3]: for first, second in itertools.combinations(numbers, 2):
   ...:     print(first, second)
   ...:
0 1
0 2
0 3
1 2
1 3
2 3
```

Generators

Widely used in algorithms:

`itertools.product(iter1, iter2, iterN, [repeat=1])`

```
In [1]: import itertools

In [2]: first_list = [1,2,3]
In [3]: second_list = ['a','b','c']
In [4]: third_list = [True,False]

In [5]: it = itertools.product(first_list, second_list, third_list)

In [6]: next(it)
Out[6]: (1, 'a', True)
In [7]: next(it)
Out[7]: (1, 'a', False)
In [8]: next(it)
Out[8]: (1, 'b', True)
```

Contents

- Closures
- Generators
- Coroutines👈

Coroutines

Starting from Python 2.5, the `yield` statement can be used as an **right value**:

```
captured_input = yield value_to_yield
```

Generators defined like this can **accept sent input** while providing output. These generators are called **coroutines**.

Coroutines

The concept of coroutines was proposed in the 60s, but only gained traction in recent years.

Coroutines can be seen as a combination of **subroutines** and **threads**.

- Can **pause and restart** during execution.
- Controlled by **itself** instead of the operating system.
- Different coroutines run within a thread are **concurrent** instead of **parallel**.

Coroutines

Simple example:

```python
import math


def update_mean():
    current_input = yield

    sum = current_input
    count = 1

    while True:
        current_input = yield sum / count

        sum += current_input
        count += 1
```

Coroutines

Simple example:

```
In [3]: updater = update_mean()

In [4]: next(updater)
```

This executes:

```
    current_input = yield
```

And the coroutine waits for an input to be sent.

Coroutines

Send an input:

```
In [5]: updater.send(2)
Out[5]: 2.0
```

The coroutine receives the input, and executes:

```
    sum = current_input
    count = 1
    while True:
        current_input = yield sum / count
```

And the coroutine waits for an input to be sent.

Coroutines

Send an input:

```
In [6]: updater.send(4)
Out[6]: 3.0
```

The coroutine receives the input, and executes:

```
        sum += current_input
        count += 1
    while True:
        current_input = yield sum / count
```

And the coroutine waits again for an input to be sent.

Coroutines

More complicated example: set-associative cache simulation

- `number_of_cache_sets` * Set
    - `number_of_ways_of_associativity` * Block
        - `block_size_in_bytes` * Byte
- The whole set-associative cache is a coroutine receiving `(address, is_write)` tuples as input, and calculating `(cache_hit, writeback_address)` tuples as output.
    - It models **each set** as a coroutine receiving `(tag, is_write)` tuples as input, and calculating `(cache_hit, writeback_address)` tuples as output.
        - Different coroutine definitions for round-robin, LRU, etc.

Coroutines

The whole set-associative cache

```
def cache_coroutine(cache_set_coroutine_function, block_size_in_bytes, number_of_ways_of_associativity, number_of_cache_sets):
    # create cache_set_coroutine_list and activate each cache_set_coroutine
    cache_set_coroutine_list = [ cache_set_coroutine_function(number_of_ways_of_associativity) for _ in range(number_of_cache_sets) ]
    for cache_set_coroutine in cache_set_coroutine_list:
        next(cache_set_coroutine)

    # get function_to_split_address and function_to_merge_address
    function_to_split_address, function_to_merge_address = get_functions_to_split_and_merge_address(
        block_size_in_bytes,
        number_of_cache_sets
    )

    # receive address, is_write
    # yields nothing
    address, is_write = yield

    while True:
        # splits address
        tag, cache_set_index, offset = function_to_split_address(address)

        # send (tag, is_write) to the appropriate cache_set_coroutine
        cache_hit, victim_tag, writeback_required = cache_set_coroutine_list[cache_set_index].send((tag, is_write))

        # create writeback_address if (victim_tag is not None) and writeback_required
        if (victim_tag is not None) and writeback_required:
            writeback_address = function_to_merge_address(victim_tag, cache_set_index, 0)
        else:
            writeback_address = None

        # receive address, is_write
        # yield cache_hit, writeback_address
        address, is_write = yield cache_hit, writeback_address
```

Coroutines

Cache Set with LRU replacement policy

```python
def lru_cache_set_coroutine(associativity):
    tag_list = [ None for _ in range(associativity) ]
    dirty_bit_list = [ False for _ in range(associativity) ]

    indices_in_lru_order = OrderedDict()
    for index in range(associativity - 1, -1, -1):
        indices_in_lru_order[index] = None

    # receive first tag and is_write
    tag, is_write = yield

    while True:
        cache_hit = False
        victim_tag = None
        writeback_required = False

        try:
            # find tag_index
            tag_index = tag_list.index(tag)

            # tag_index found
            cache_hit = True

            if is_write:
                dirty_bit_list[tag_index] = True

            # move tag_index to the end of indices_in_lru_order
            indices_in_lru_order.move_to_end(tag_index)

        except ValueError:
            # tag_index not found
            # get index_of_victim from indices_in_lru_order
            index_of_victim, _ = indices_in_lru_order.popitem(last=False)

            victim_tag = tag_list[index_of_victim]

            if dirty_bit_list[index_of_victim]:
                writeback_required = True

            tag_list[index_of_victim] = tag

            if is_write:
                dirty_bit_list[index_of_victim] = True
            else:
                dirty_bit_list[index_of_victim] = False

            # insert index_of_victim to the end of indices_in_lru_order
            indices_in_lru_order[index_of_victim] = None

        # receive tag and is_write
        # yield (cache_hit, victim_tag, writeback_required)
        tag, is_write = yield (cache_hit, victim_tag, writeback_required)
```

Coroutines

- Suppose our cache has only *eight* blocks and each block contains *four* words.
- The cache is *2-way* set associative, so there are four sets of two blocks.
- The write policy is *write-back* and write-allocate.
- *LRU replacement* is used.

---

https://courses.cs.washington.edu/courses/cse378/02sp/sections/section9-3.html

Coroutines

```
In [3]: cache = cache_coroutine(lru_cache_set_coroutine, block_size_in_bytes=4 *
   ...:  2, number_of_ways_of_associativity=2, number_of_cache_sets=4)

In [4]: next(cache)

In [5]: cache.send((0, True))
Out[5]: (False, None)

In [6]: cache.send((64, False))
Out[6]: (False, None)

In [7]: cache.send((4, True))
Out[7]: (True, None)

In [8]: cache.send((40, True))
Out[8]: (False, None)

In [9]: cache.send((68, False))
Out[9]: (True, None)

In [10]: cache.send((128, True))
Out[10]: (False, 0)

In [11]: cache.send((0, False))
Out[11]: (False, None)
```