Demystifying Python

ABBAS WU

讲座内容

不讲：

- Python的基本语法和常用内置类型
- Python的常用第三方库（如numpy、pytorch、django）
- 如何借助于这些第三方库完成某些具体任务（如数值计算、神经网络、Web服务器后端）

讲 Python实用而有趣的函数式语言构造，包括：

- 函数
- 迭代器
  - （传统的）生成器
- （作为协程的）生成器

Python的内置类型

- None
- 数字
- **序列**
  - 字符串
  - 列表
  - 元组
  - range
  - collections.deque
- 字典
- 集合
- **函数**
- **迭代器**
  - **（传统的）生成器**
- **（作为协程的）生成器**
- 对象与类
- 异常
- 模块

序列

- 序列解包

序列解包（1）

在Python中，可以将序列对象（字符串、列表、元组、range、collections.deque）解包为一组变量，例如：

```
person = ['Zhang', 'San'] # 列表，[名, 姓]
address = ('www.python.org', 80) # 元组，(域名, 端口号)
address = 'www.google.com', 80 # 隐式地创建元组
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' # 字符串
```

解包列表

```
>>> person = ['Zhang', 'San']
>>> first_name, last_name = person
>>> first_name
'Zhang'
>>> last_name
'San'
```

解包元组

```
>>> address = ('www.python.org', 80)
>>> host, port = address
>>> host
'www.python.org'
>>> port
80
```

序列解包（2）

如果解包的变量数比序列中的元素数少，可以在一个变量名前加*，此时该变量将被解包为包含0个、1个或多个值的列表。

```
>>> alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> # 解包得到第一个字母、第二个字母和余下各字母
... first_letter, second_letter, *remaining = alphabet
>>> first_letter
'A'
>>> second_letter
'B'
>>> remaining
['C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

序列解包（2）

如果解包的变量数比序列中的元素数少，可以在一个变量名前加*，此时该变量将被解包为包含0个、1个或多个值的列表。

```
>>> alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> # 解包得到前面的字母和最后一个字母
... *letters_in_the_front, last_letter = alphabet
>>> letters_in_the_front
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']
>>> last_letter
'Z'
```

序列解包（2）

如果解包的变量数比序列中的元素数少，可以在一个变量名前加*，此时该变量将被解包为包含0个、1个或多个值的列表。

```
>>> alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> # 解包得到第一个字母、最后一个字母、中间各字母
... first_letter, *letters_in_the_middle, last_letter = alphabet
>>> first_letter
'A'
>>> letters_in_the_middle
['B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y']
>>> last_letter
'Z'
```

序列解包的重要应用：交换两个变量的值

如何交换两个变量first和second的值?

传统思路:

```
>>> first, second = 1, 2
>>> temp = first
>>> first = second
>>> second = temp
>>> first
2
>>> second
1
```

序列解包的重要应用：交换两个变量的值

传统思路的缺点：

- 麻烦
- 如果在代码中需要频繁交换两个变量的值，会导致一段代码反复出现，而在Python中，这段代码无法被封装成函数

序列解包的重要应用：交换两个变量的值

Python实现交换两个变量的值的最佳实践为first, second = second, first。如下所示：

```
>>> first, second = 1, 2
>>> first, second = second, first
>>> first
2
>>> second
1
```

序列解包的重要应用：交换两个变量的值

first, second = second, first的原理：

1. 赋值运算符右边的second, first隐式地创建一个包含second、first两个变量的值的元组
2. 赋值运算符左边，将该元组的值解包到first、second两个变量中

优点：简洁、可读性好

函数

- 作为对象与闭包的函数

作为对象与闭包的函数

函数在Python中是第一类对象。也就是说，可以把它们当作参数传递给其他的函数，放在数据结构中，以及作为函数的返回值。

下面的例子给出了一个函数，它接受另一个函数作为输入并调用它。

```
>>> def call_function(function):
...     return function()
...
>>> def return_hello_world():
...     return 'Hello World'
...
>>> call_function(return_hello_world)
'Hello World'
```

作为对象与闭包的函数

Python支持嵌套的函数定义。

1. **如果在内层函数需要修改外层函数的局部变量，需要在内层函数声明该变量为nonlocal。**

不使用nonlocal:

```
>>> def outer_function():
...     string = 'Hello'
...     def inner_function():
...         # 此时string为inner_function的局部变量
...         string = 'World'
...     inner_function()
...     return string
...
>>> outer_function()
'Hello'
```

作为对象与闭包的函数

Python支持嵌套的函数定义。

1. **如果在内层函数需要修改外层函数的局部变量，需要在内层函数声明该变量为nonlocal。**

使用nonlocal:

```
>>> def outer_function():
...     string = 'Hello'
...     def inner_function():
...         # 此时string就是外层函数的string
...         nonlocal string
...         string = 'World'
...     inner_function()
...     return string
...
>>> outer_function()
'World'
```

作为对象与闭包的函数

Python支持嵌套的函数定义。

2. 所有的函数都是**闭包**，包含组成函数的语句，以及这些语句的**执行环境**。**可以在外层函数中定义内层函数，并将内层函数返回。返回后的内层函数仍然可以在外层函数外被调用。**这是一种设计模式，常用来创建**定制化的函数**。

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

作为对象与闭包的函数

Python支持嵌套的函数定义。

2. 所有的函数都是**闭包**，包含组成函数的语句，以及这些语句的**执行环境**。**可以在外层函数中定义内层函数，并将内层函数返回。返回后的内层函数仍然可以在外层函数外被调用。**这是一种设计模式，常用来创建**定制化的函数**。

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

作为对象与闭包的函数

查看函数闭包中的内容:

```
>>> 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'
```

作为对象与闭包的函数

如果要**在一系列函数调用中记录某一个状态**，使用闭包是一个非常高效的方式。

传统（面向对象）思路：

```
# 倒数计数器
class Countdown:
    # 初始化倒数计数器的值
    def __init__(self, n):
        self.n = n
    # 获取倒数计数器的下一个值
    def next_value(self):
        old_value = self.n
        self.n -= 1
        return old_value
```

作为对象与闭包的函数

如果要**在一系列函数调用中记录某一个状态**，使用闭包是一个非常高效的方式。

函数式编程思路：

```
# 提供初始值，返回用于获取倒数计数下一个值的函数
def countdown(n):
    # 定义用于获取倒数计数下一个值的函数
    def get_next_value():
        # 此时n就是外层函数的n，这样可以实现对外层函数的n的修改
        nonlocal n
        old_value = n
        n -= 1
        return old_value
    # 返回定义的函数
    return get_next_value
```

作为对象与闭包的函数

速度测试

```
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
```

作为对象与闭包的函数

速度测试

```
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)
```

迭代器

- 迭代器和可迭代对象
- （传统的）生成器
- 从可迭代对象创建高效迭代器的函数

迭代器和可迭代对象

迭代器是一个可以记住遍历的位置的对象。它从第一个元素开始访问，直到所有的元素被访问完结束。在访问元素的过程中，迭代器只能前进，不能后退。

在Python中，可以**通过iter()函数，从提供__iter__()方法的对象中获取迭代器**，包括序列、字典、集合，以及用户自定义类型。如下所示：

```
>>> numbers = range(10)
>>> numbers_iterator = iter(numbers)
>>> # numbers_iterator = numbers.__iter__()也行
```

迭代器和可迭代对象

在Python中，**所有的迭代器对象实现了__next__()方法，用于返回迭代器当前指向的元素，并让迭代器前进。通常通过next()函数调用迭代器对象的__next__()方法：**

```
>>> numbers_iterator.__next__()
0
>>> next(numbers_iterator)
1
>>> next(numbers_iterator)
2
>>> numbers_iterator.__next__()
3
```

迭代器和可迭代对象

在Python中，除了通过手动调用next()函数获取迭代器当前指向的元素，并让迭代器前进之外，**还可以在for循环中一次次获取迭代器当前指向的元素，并让迭代器前进。不过，迭代器已经过了的元素不能再次被访问。**如下所示：

```
>>> for remaining_number in numbers_iterator:
...     print(remaining_number)
...
4
5
6
7
8
9
```

迭代器和可迭代对象

迭代器和可以获取迭代器的对象，统称“可迭代对象”。所有的可迭代对象，都可以作为for循环遍历的对象。如果for循环遍历的对象本身不是迭代器，Python会首先创建迭代器，然后针对迭代器运行for循环。

```
for number in numbers:
    print(number)
```

等效于：

```
numbers_iterator = iter(numbers)
for number in numbers_iterator:
    print(number)
```

（传统的）生成器

在定义函数时使用yield关键字可以定义**生成器**。生成器是一种迭代器，它允许用户以函数的形式，定义一个按照一定规则得到一个值的序列的迭代器。

如下所示:

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

（传统的）生成器

调用该生成器，我们可以创建一个生成器对象。可以对该生成器调用next函数，也可以将该生成器用于for循环:

```
In [2]: c = countdown(10)

In [3]: next(c)
Out[3]: 10

In [4]: next(c)
Out[4]: 9

In [5]: for value in c:
   ...:     print(value)
   ...:
8
7
6
5
4
3
2
1
```

（传统的）生成器

对生成器对象调用next()时，生成器函数将开始执行语句，直至遇到yield语句为止。yield语句将返回指定的值。

```
In [2]: c = countdown(10)

In [3]: next(c)
Out[3]: 10
```

此时执行的语句:

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

然后对外返回n。

（传统的）生成器

对生成器对象调用next()时，生成器函数将开始执行语句，直至遇到yield语句为止。yield语句将返回指定的值。

```
In [4]: next(c)
Out[4]: 9
```

此时执行的语句:

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

然后对外返回n。

（传统的）生成器

生成器的这种**按需返回值**的特性，被称为**惰性求值**。在某些应用中，这可能极大地提高性能和内存使用。如下所示的函数:

```
# 打开一个文件，从中返回所有注释
def get_comments_from_file(file):
    with open(file, 'r') as fp:
        for line in fp:
            # 删除前后空白
            stripped_line = line.strip()
            # 检查删除前后空白后该行是否为空
            if stripped_line:
                # 若不为空，通过第一个字符判断是否为注释
                if stripped_line[0] == '#':
                    # 若为注释，返回该行
                    yield stripped_line
```

（传统的）生成器

在这个例子中，生成器表达式提取文件各行并判断该行是否为注释，如果为注释，就将该行返回。

但实际上，该生成器没有将整个文件读到内存中。只有当用户对生成器调用next函数，或使用生成器的循环进入下一次迭代时，生成器才会继续逐行读文件，在遇到注释行时返回。

这是一种从GB级大小的Python源文件中提取注释的高效方法。

从可迭代对象创建高效迭代器的函数

Python包含从可迭代对象创建高效迭代器的函数：

1. 过滤
   - filter(predicate, iterable)
   - itertools.filterfalse(predicate, iterable)
   - itertools.dropwhile(predicate, iterable)
   - itertools.takewhile(predicate, iterable)
   - itertools.islice(predicate, [start,] stop [,step])
2. **排列组合**
   - itertools.permutations(iterable [, r])
   - itertools.combinations(iterable, r)
3. 合并
   - zip(iter1, iter2, ... iterN)
   - itertools.product(iter1, iter2, ... iterN, [repeat=1])
4. 枚举
   - enumerate(iterable, start=0)
5. 函数式编程
   - map(func, *iterables)
   - functools.reduce(function, sequence[, initial])

创建排列组合迭代器

itertools提供了如下创建排列组合迭代器的函数:

itertools.permutations(iterable [,r])

创建一个迭代器，返回iterable中所有长度为r的排列。如果省略了r，那么返回iterable中的所有全排列。

创建排列组合迭代器

```
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)

In [7]: next(permutations_of_two_numbers_iterator)
Out[7]: (1, 0)

In [8]: next(permutations_of_two_numbers_iterator)
Out[8]: (1, 2)
```

创建排列组合迭代器

itertools提供了如下创建排列组合迭代器的函数:

itertools.combinations(iterable ,r)

创建一个迭代器，返回iterable中所有长度为r的组合。

```
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
```

（作为协程的）生成器

Python 2.5允许yield语句作为生成器内的赋值语句的**右值**出现，如下所示:

```
captured_input = yield value_to_yield
```

这样的生成器除了对外返回值之外，**还可以接受外界输入**。这样的生成器被称为**协程**。

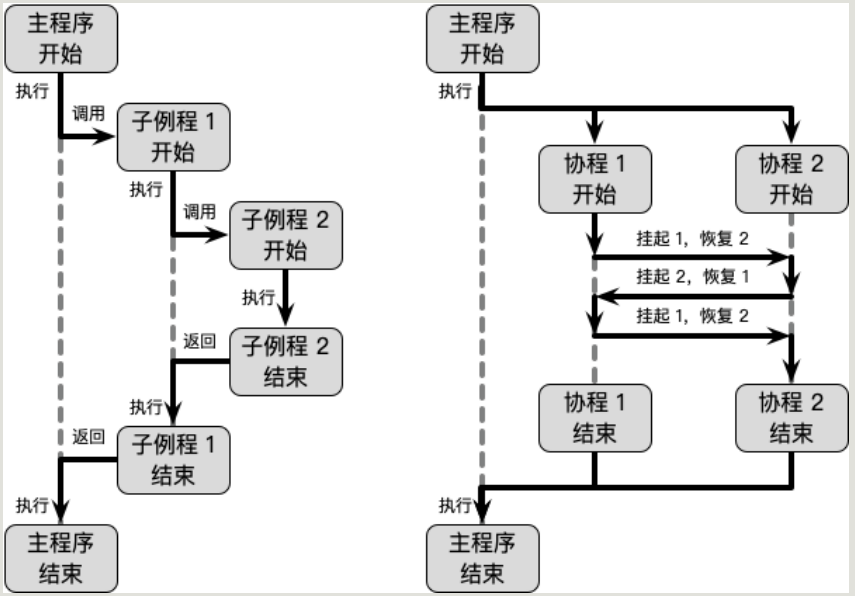
（作为协程的）生成器

协程（coroutine）的概念早在上世纪60年代就被提出来了，但直到最近几年才在某些语言中得到广泛应用。

协程可以视为**函数**与**线程**的结合体。

- 与函数不同，**协程在执行过程中可中断，在适当的时候再返回来接着执行。**
- 多个协作的协程有点像多线程，但:
  1. 协程切换由**程序自身**而不是操作系统控制。
  2. 协程都在一个线程内，在微观上串行，不存在同时写变量冲突，不需要多线程的锁机制。

（作为协程的）生成器



函数与协程

（作为协程的）生成器

以下的协程针对流式数据实时更新并返回平均值和标准差：

```
import math


# 针对流式数据实时更新并返回均值和标准差（协程）
def update_mean_and_standard_deviation():
    # 从外界接受第一个输入
    current_input = yield

    # 所有数据的和
    sum = current_input
    # 所有数的平方之和
    sum_of_squares = current_input * current_input
    # 所有数据的个数
    count = 1
    # 均值
    mean = current_input
    # 标准差
    standard_deviation = 0

    while True:
        # 返回均值与标准差，并接受下一个输入
        current_input = yield mean, standard_deviation

        sum += current_input
        sum_of_squares += current_input * current_input
        count += 1
        mean = sum / count
        standard_deviation = math.sqrt(sum_of_squares / count - mean * mean)
```

（作为协程的）生成器

首先，创建并初始化协程：

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

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

执行协程内语句：

```
current_input = yield
```

此时，协程等待外界提供值完成current_input = yield的赋值。

（作为协程的）生成器

其次，提供第一个数据：

```
In [5]: updater.send(2)
Out[5]: (2, 0)
```

此时，协程完成current_input = yield的赋值，并执行：

```
    sum = current_input
    sum_of_squares = current_input * current_input
    count = 1
    mean = current_input
    standard_deviation = 0
    while True:
        current_input = yield mean, standard_deviation
```

此后，协程等待外界提供值完成current_input = yield mean, standard_deviation的赋值。

（作为协程的）生成器

再其次，再提供一个数据：

```
In [6]: updater.send(4)
Out[6]: (3.0, 1.0)
```

此时，协程完成current_input = yield mean, standard_deviation的赋值，并执行：

```
        sum += current_input
        sum_of_squares += current_input * current_input
        count += 1
        mean = sum / count
        standard_deviation = math.sqrt(sum_of_squares / count - mean * mean)
    while True:
        current_input = yield mean, standard_deviation
```

此后，协程等待外界提供值完成current_input = yield mean, standard_deviation的赋值。

谢谢大家!