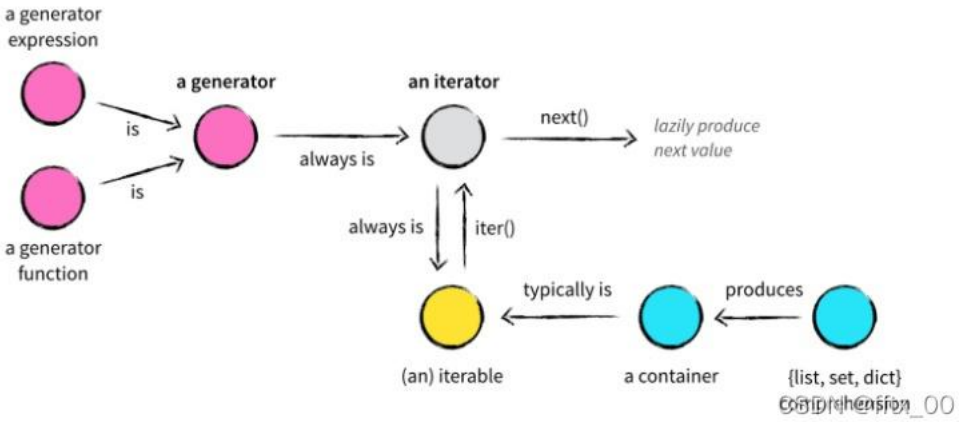
【Python】可迭代对象、迭代器对象、生成器对象、序列

生成器|迭代器关系，可以看下图：



1.迭代器 (iterator)

- 迭代器

```
# 迭代器类型的定义：
    1.当类中定义了 __iter__ 和 __next__ 两个方法。
    2.__iter__ 方法需要返回对象本身，即：self
    3. __next__ 方法，返回下一个数据，如果没有数据了，则需要抛出一个StopIteration的异常。
    官方文档：https://docs.python.org/3/library/stdtypes.html#iterator-types

# 创建 迭代器类型 ：
    class IT(object):
        def __init__(self):
            self.counter = 0

        def __iter__(self):
            return self

        def __next__(self):
            self.counter += 1
            if self.counter == 3:
                raise StopIteration()
            return self.counter

# 根据类实例化创建一个迭代器对象：
    obj1 = IT()

    # v1 = obj1.__next__()
    # v2 = obj1.__next__()
    # v3 = obj1.__next__() # 抛出异常

    v1 = next(obj1) # obj1.__next__()
    print(v1)

    v2 = next(obj1)
    print(v2)

    v3 = next(obj1)
    print(v3)


    obj2 = IT()
    for item in obj2:  # 首先会执行迭代器对象的__iter__方法并获取返回值，一直去反复的执行 next(对象)
        print(item)
```

迭代器对象支持通过next取值，如果取值结束则自动抛出StopIteration。
for循环内部在循环时，先执行__iter__方法，获取一个迭代器对象，然后不断执行的next取值（有异常StopIteration则终止循环）。

2.生成器(generator)

- 生成器

```python
# 创建生成器函数
    def func():
        yield 1
        yield 2

# 创建生成器对象（内部是根据生成器类generator创建的对象），生成器类的内部也声明了：__iter__、__next__ 方法。
    obj1 = func()

    v1 = next(obj1)
    print(v1)

    v2 = next(obj1)
    print(v2)

    v3 = next(obj1)
    print(v3)


    obj2 = func()
    for item in obj2:
        print(item)
```

如果按照迭代器的规定来看，其实生成器类也是一种特殊的迭代器类（生成器也是一个中特殊的迭代器）。

3.可迭代对象(iterable)

- 可迭代对象

```python
# 如果一个类中有__iter__方法且返回一个迭代器对象 ；则我们称以这个类创建的对象为可迭代对象。

class Foo(object):

    def __iter__(self):
        return 迭代器对象(生成器对象)

obj = Foo() # obj是 可迭代对象。

# 可迭代对象是可以使用for来进行循环，在循环的内部其实是先执行 __iter__ 方法，获取其迭代器对象，然后再在内部执行这个迭代器对象的next功能，逐步取值。
for item in obj:
    pass
```



```python
from collections.abc import Iterator, Iterable

v1 = [11, 22, 33]
print( isinstance(v1, Iterable) )  # True，判断是可迭代；判断依据是是否有__iter__且返回迭代器对象。
print( isinstance(v1, Iterator) )  # false，判断是否是迭代器；判断依据是__iter__ 和 __next__。

v2 = v1.__iter__()
print(isinstance(v2, Iterable))  # True
print(isinstance(v2, Iterator))  # True
```


暴龙科利尔 LV3

有了迭代器为什么还要可迭代对象？都用迭代器实现不就行了

2021-07-18 04:25 5 回复

永昼的星期天 LV6 因为可迭代对象可以通过增加类方法实现更多的功能，比如list，它是个可迭代对象，但是它的功能远远超出迭代器所有的，比如append，clear，copy等等。迭代器实质上只是一个强大的类的配件

2021-11-18 19:51 1 回复

4.序列(sequence)

一个通过特殊方法支持使用整数索引的有效元素访问的可迭代对象，`__getitem__()`并定义了一个 `__len__()` 返回序列长度的方法。一些内置序列类型`list`， `str`， `tuple`，和`bytes`。请注意，`dict`也支持 `__getitem__()` and `__len__()`，但被视为映射而不是序列，因为查找使用任意 不可变键而不是整数。

该`collections.abc.Sequence`抽象基类定义了更加丰富的接口，并不仅止于 `__getitem__()`和`__len__()`补充 `count()`， `index()`， `__contains__()`，和 `__reversed__()`。可以使用 显式注册实现此扩展接口的类型 `register()`。

其实这里需要引入一个概念，叫迭代器，常见的就是我们在使用`for`语句的时候，python内部其实是把`for`后面的对象上使用了内建函数`iter`，比如：

```
a = [1, 2, 3]
for i in a:
    do_something()
```

其实在python内部进行了类似如下的转换：

```
a = [1, 2, 3]
for i in iter(a):
    do_something()
```



```
>>> a=[1,2,3]
>>> b=a.__iter__()
>>> c=b.__next__()
>>> c
1
>>> c=b.__next__()
>>> c
2
>>> c=b.__next__()
>>> c
3
>>> c=b.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```


 # 每个迭代器只能被使用遍历一次

```
>>> a=[1,2,3,4]
>>> b=a.__iter__()
>>> b
<list_iterator object at 0x0000017439740A00>
>>> print(b)
<list_iterator object at 0x0000017439740A00>
>>> print(list(b))
[1, 2, 3, 4]
>>> b
<list_iterator object at 0x0000017439740A00>
>>> print(list(b))
[]
```


 # 遍历过之后迭代器不会还原，这样看起来像个空数组

```
>>> a=[1,2,3,4]
>>> b=a.__iter__()
>>> b=b.__next__()
>>> print(list(b))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
```


 # __next__()返回的是，**下一个对象的值**，不是下一个迭代器对象

参考资料：

1.Python之Iterable与Iterator - 知乎

2.Glossary — Python 3.10.1 documentation

3.『教程』Python中的迭代器_哔哩哔哩_bilibili

4.15分钟彻底搞懂迭代器、可迭代对象、生成器【python迭代器】_哔哩哔哩_bilibili