14 迭代器协议

@(SIGAI课程录制)

本节课的OKR

Objective

- 深入理解Python中的迭代
- 写出更加Pythonic的数据迭代操作

Key Result

- 清楚什么是Python中的 `Iterator Protocol`
- 清楚 `for` 循环背后的实现逻辑
- 清楚 `Iterable` `Iterator` `Generator` 三者之间的关系
- 清楚 `__iter__()` 与 `__next__()` 分别是做什么的
- 清楚为什么要使用 `Iterator`
- 熟练使用 `Generator` ，并在代码中尽可能的用 `Generator` 替换 `List`

Iterator Protocol

- 迭代器是一个对象
- 迭代器可以被 `next()` 函数调用，并返回一个值
- 迭代器可以被 `iter()` 函数调用，并返回迭代器自己
- 连续被 `next()` 调用时依次返回一系列的值
- 如果到了迭代的末尾，则抛出 `StopIteration` 异常
- 迭代器也可以没有末尾，只要被 `next()` 调用，就一定会返回一个值
- Python中， `next()` 内置函数调用的是对象的 `__next__()` 方法
- Python中， `iter()` 内置函数调用的是对象的 `__iter__()` 方法
- 一个实现了迭代器协议的的对象可以被 `for` 语句循环迭代直到终止

Example - 1

```
class XIterator:
    def __next__(self):
        return "Hello World"


def main():
    x_it = XIterator()
    [print(next(x_it)) for i in range(3)]


if __name__ == "__main__":
    main()
```

输出：

```
Hello World
Hello World
Hello World
```

说明：

- 只要一个对象实现了`__next__()`方法，就可以被`next()`函数调用

Example - 2

```
class XIterator:
    def __init__(self):
        self.elements = list(range(5))

    def __next__(self):
        if self.elements:
            return self.elements.pop()


def main():
    x_it = XIterator()
    [print(next(x_it)) for i in range(10)]


if __name__ == "__main__":
    main()
```

输出：

```
4
3
2
1
0
None
None
None
None
None
```

说明：

- 此时由于没有实现`__iter__()`，若用`for`语句迭代，会报错

Example - 3

```
class XIterator:
    def __init__(self):
        self.elements = list(range(5))

    def __next__(self):
        if self.elements:
            return self.elements.pop()

    def __iter__(self):
        return self

def main():
    x_it = XIterator()
    for x in x_it:
        print(x)

if __name__ == "__main__":
    main()
```

说明：

- 实现了`__iter__()`和`__next__()`后便可以被`for`语句迭代了
- 但此时程序一旦开始执行便不再停止，Why?

`for`语句的内部实现

```
for element in iterable:
    # do something with element
```

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

说明：

- `for`语句里用的是`iterable`，而非`iterator`
- `for`语句执行的第一个操作是从一个`iterable`生成一个`iterator`
- `for`语句的循环体其实是靠检测`StopIteration`异常来中断的
- 要想被`for`语句迭代需要三个条件：`__iter__()` `__next__()` `StopIteration`

> 如果我们可以从一个对象里获得一个迭代器（Iterator），那么这个对象就是可迭代对象（Iterable） 迭代器都是可迭代对象（因为实现了`__iter__()`），但可迭代对象不一定是迭代器

Example - 4

```
class XIterator:
    def __init__(self):
        self.elements = list(range(5))

    def __next__(self):
        if self.elements:
            return self.elements.pop()
        else:
            raise StopIteration

    def __iter__(self):
        return self

def main():
    x_it = XIterator()
    for x in x_it:
        print(x)

if __name__ == "__main__":
    main()
```

输出：

```
4
3
2
1
0
```

Generator

- 迭代器协议很有用，但实现起来有些繁琐，没关系，生成器来帮你
- 生成器在保持代码简洁优雅的同时，自动实现了迭代器协议

实现生成器的方式一：`yeild` Expression

```python
def f():
    yield 1
    yield 2
    yield 3

def main():
    f_gen = f()
    [print(next(f_gen)) for i in range(5)]

if __name__ == "__main__":
    main()
```

输出：

```
1
2
3
Traceback (most recent call last):
  ...
StopIteration
```

说明：

- 使用`yeild`语句可以自动实现迭代器协议

> `yeild`与`return`有何区别：相比`return`是退出，`yeild`更像是暂停

既然实现了迭代器协议，就说明可以被`for`迭代：

```
def f():
    yield 1
    yield 2
    yield 3

def main():
    f_gen = f()
    for x in f_gen:
        print(x)

if __name__ == "__main__":
    main()
```

输出：

```
1
2
3
```

实现生成器的方式二：`Generator Expression`

```
>>> [print(x) for x in (x ** 2 for x in range(5))]
0
1
4
9
16
[None, None, None, None, None]
```

在Python中不要小瞧任何看似微小的区别

```
sum([x ** 2 for x in range(10000000)])
sum(x ** 2 for x in range(10000000))
```

为什么需要生成器

1. 相比迭代器协议，实现生成器的**代码量小**，**可读性更高**
2. 相比在`List`中操作元素，直接使用生成器能**节省大量内存**
3. 有时候我们会需要写出一个无法在内存中存放的**无限数据流**
4. 你可以建立**生成器管道**（多个生成器链式调用）

用生成器表示全部的斐波那契数列

```
def fibonacci():
    temp = [1, 1]
    while True:
        temp.append(sum(temp))
        yield temp.pop(0)
```

通过生成器管道模块化处理数据

```
def fibonacci():
    temp = [1, 1]
    while True:
        temp.append(sum(temp))
        yield temp.pop(0)

def dataflow():
    for x in fibonacci():
        yield x ** 2

if __name__ == "__main__":
    for x in dataflow():
        print(x)
        if x > 100000:
            break
```

总结：关于Python中的迭代思维

- Python中有两类运算：

1. 可以并行的矢量化的运算：Numpy
2. 必须一个个的操作的迭代式运算：Generator

- Python中有两类数据：

1. 内存中放得下的：数据量较小的数据
2. 内存中放不下的：数据量较大或者无穷大的数据

- Python中有两种思维：

1. Eager：着急派，需要的话，必须全都准备好
2. Lazy：懒惰派，“哎，干嘛那么着急，需要的时候再说呗”

- Python中处处是迭代器，缺的是发现迭代器的眼睛