迭代器和生成器

迭代器（Iterator）

可迭代（Iterables）

- 列表、元组、字典、Ranges、还有字符串（还有集合）都是可迭代的对象

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]

ranked_chocolates = ("Dark", "Milk", "White")

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}

best_topping = "pineapple"
```

迭代

我们可以对可迭代的对象迭代:

```
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
for item in my_order:
    print(item)
lowered = [item.lower() for item in my_order]

ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen":
19.99}
for product in prices:
    print(product, " costs ", prices[product])
discounted = { item: prices[item] * 0.75 for item in prices }

best_topping = "pineapple"
for letter in best_topping:
    print(letter)
```

迭代子

- 一个迭代子是一个可以提供序列化访问值的对象，其一次访问一个值！
    - iter(iterable) 返回一个在iterable对象之上的迭代器
    - next(iterator) 返回迭代器的下一个元素

迭代子

```
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]

topperator = iter(toppings)
next(iter) # 'pineapple'
next(iter) # 'pepper'
next(iter) # 'mushroom'
next(iter) # 'roasted red pepper'
next(iter) # ❌ StopIteration exception
```

迭代子是可变的吗？🤔

demo

处理 StopIteration

- StopIteration是一个会终止程序正常运行的"异常"（Exception）
- 处理异常应该使用 try/except

处理 StopIteration

```
ranked_chocolates = ("Dark", "Milk", "White")

chocolaterator = iter(ranked_chocolates)
print(next(chocolaterator))
print(next(chocolaterator))
print(next(chocolaterator))

try:
    print(next(chocolaterator))
except StopIteration:
    print("No more left!")
```

demo

处理 StopIteration

- 配合while来处理迭代

```
ranked_chocolates = ("Dark", "Milk", "White")
chocolaterator = iter(ranked_chocolates)

try:
    while True:
        choco = next(chocolaterator)
        print(choco)
except StopIteration:
    print("No more left!")
```

demo

Iterators vs. For loops

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator = iter(ranked_chocolates)

try:
    while True:
        choco = next(chocorator)
        print(choco)
except StopIteration:
    print("No more left!")
```

```
ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)
```

Actually, a for loop is just syntactic sugar! 🍬

再次回顾for语句

```
for <name> in <expression>:
    <suite>
```

- 语义：
  - 1. Python 首先求值头部的<expression>，确保其产生一个Iterable
  - 2. Python 得到iterable的迭代器
  - 3. Python 利用iterable得到其next value，并绑定到当前帧的name
  - 4. Python 执行<suite>中的语句
  - 5. Python 重复上述操作直到 StopIteration error

```
iterator = iter(<expression>)
try:
    while True:
        <name> = next(iterator)
        <suite>
except StopIteration:
    pass
```

内部的__next__()和__iter__()

- iter()函数本质上调用该对象"自己"的__iter__()

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator1 = iter(ranked_chocolates)
chocorator2 = ranked_chocolates.__iter__()
```

什么叫自己的?

- next()函数本质上调用该迭代器"自己"的__next__()

```
ranked_chocolates = ("Dark", "Milk", "White")
chocolate1 = next(chocorator1)
chocolate2 = chocorator2.__next__()
```

demo

比较两种迭代

for

```
ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)
```

Iterator

```
ranked_chocolates = ("Dark", "Milk", "White")
chocorator = iter(ranked_chocolates)
try:
    while True:
        print(next(chocorator))
except StopIteration:
    pass
```

行为相同不等于实现相同

- For循环和迭代器的行为是一样的，但是Python实现是不同的

	10,000 runs	**1,000,000 runs**
For loop	3.2 milliseconds	336 milliseconds
Iterator	8.3 milliseconds	798 milliseconds

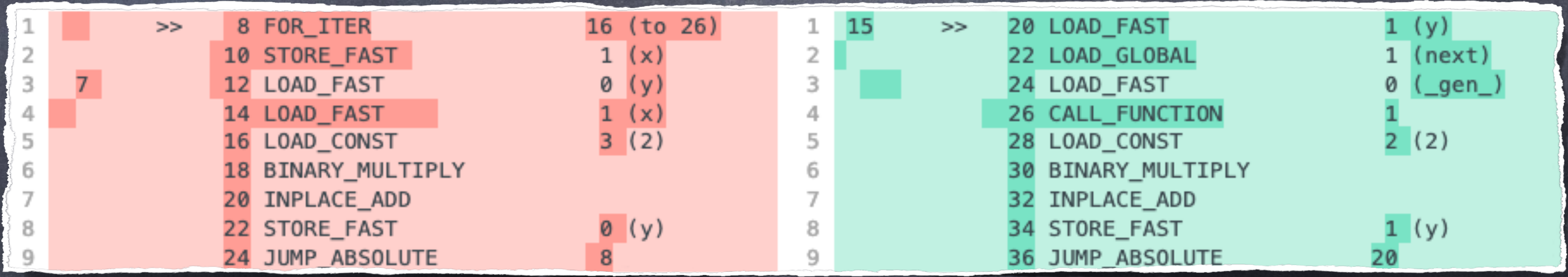
具体实现的不同

用dis模块来查看具体的不同

```
import dis
def for_version():
    y = 0
    for x in [1, 2, 3]:
        y += x * 2

def iter_version():
    _gen_ = iter([1, 2, 3])
    y = 0
    try:
        while True:
            y += next(_gen_) * 2
    except StopIteration:
        pass
```

```
dis.dis(for_version)
dis.dis(iter_version)
```



```
        >>    8 FOR_ITER                16 (to 26)
             10 STORE_FAST               1 (x)
  7          12 LOAD_FAST                0 (y)
             14 LOAD_FAST                1 (x)
             16 LOAD_CONST               3 (2)
             18 BINARY_MULTIPLY
             20 INPLACE_ADD
             22 STORE_FAST               0 (y)
             24 JUMP_ABSOLUTE            8
```

```
 15     >>   20 LOAD_FAST                1 (y)
             22 LOAD_GLOBAL              1 (next)
             24 LOAD_FAST                0 (_gen_)
             26 CALL_FUNCTION            1
             28 LOAD_CONST               2 (2)
             30 BINARY_MULTIPLY
             32 INPLACE_ADD
             34 STORE_FAST               1 (y)
             36 JUMP_ABSOLUTE           20
```

Any questions ?