Python 高级编程技术

- 生成器函数
- 文件 (file)
  - 概念与用途、基本文件操作、实例
- 闭包
  - 简单装饰器
- 命令行和命令行参数
- 运行中的错误检查和处理，异常
- with 语句

迭代器 vs. 可迭代对象

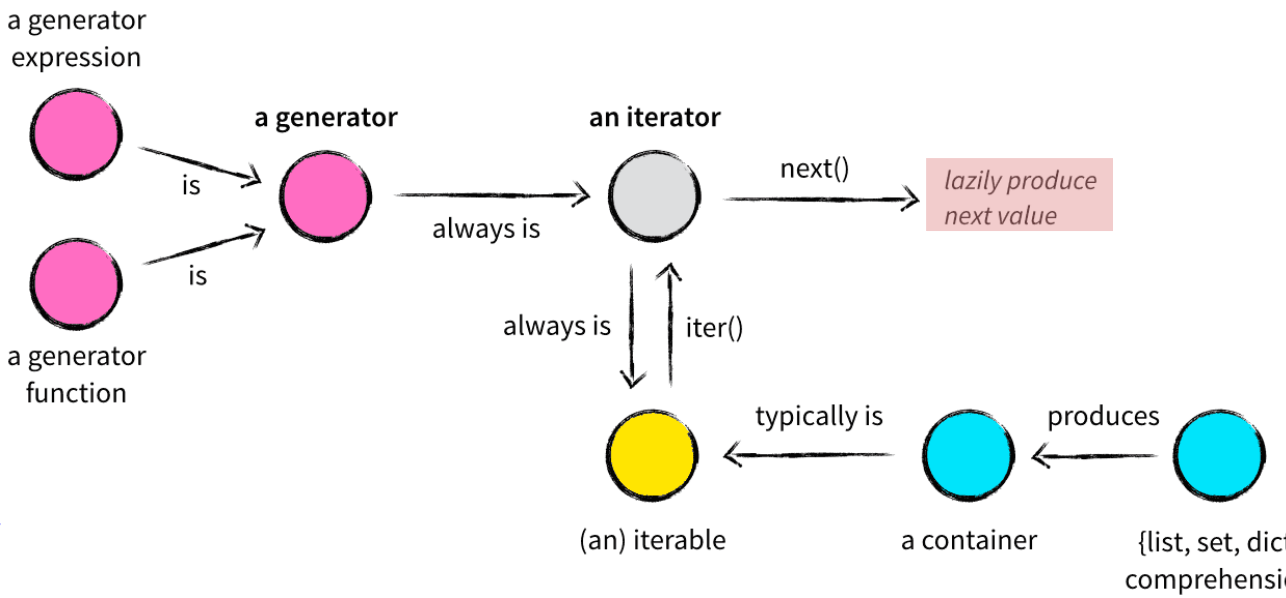
- **Python 从语义上深入地支持 '迭代'**

iterable -- 可迭代对象

能够逐一返回其成员项的对象。 可迭代对象的例子包括所有序列类型 (例如 `list`, `str` 和 `tuple`) 以及某些非序列类型例如 `dict`, 文件对象 以及定义了 `__iter__()` 方 的任意自定义类对象。

iterator -- 迭代器

用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration` 异常。



迭代器对象 vs. 可迭代对象

- **Python 从语义上深入地支持 '迭代'**

iterable -- 可迭代对象

能够逐一返回其成员项的对象。 可迭代对象的例子包括所有序列类型 (例如 `list`, `str` 和 `tuple`) 以及某些非序列类型例如 `dict`, 文件对象 以及定义了 `__iter__()` 方 的任意自定义类对象。

iterator -- 迭代器

用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration` 异常。

- **迭代器并不是一种专门的类型，也没有专用的语言结构**
  - **内置函数 iter 可从一个 iterable 构建一个 iterator**
  - **生成器表达式可简洁地创建生成器对象 (一种迭代器对象)**

```
>>> iter([1, 2, 3, 4, 5])
<list_iterator object at 0x00001813E986DC0>
>>> (i**2 for i in range(5))
<generator object <genexpr> at 0x00001813C961F20>
```

生成器函数 (generator function)

- 生成器函数：定义迭代器的常用方法
  - 语法形式：函数定义 **def** 语句
  - 当一个函数定义的体中有 (一条或多条) **yield 语句**，所定义的是特殊的生成器函数 (而非普通的函数)
  - 调用生成器函数时，会返回一个**生成器对象 (generator object)**
- **yield** 语句：**yield 表达式 [,表达式, ...]**
  - 语义：使生成器对象生成并送出表达式的值 (如有多个表达式，则生成并送出表达式值构成的元组)
- 一个生成器对象被使用时，会按需送出 (yield) 值：(可实现惰性求值)
  - 当送出 (yield) 一个值之后，该生成器对象的执行暂停在 '送出值的 **yield** 语句的位置'
  - 一旦被再次要求一个值 (或被特殊形式调用)，该生成器对象将从暂停处继续执行，直至再次遇到 **yield** 语句并送出下一个值

示例

```
#### Fibonacci 数列生成器，生成直至 limit 项的数列
def fib_gen(limit):
    f0, f1 = 0, 1
    for n in range(limit):
        yield f0
        f0, f1 = f1, f0 + f1
```

```
>>> from inspect import isgeneratorfunction
>>> isgeneratorfunction(fib_gen)
True
>>> fib_gen(15)
<generator object fib_gen at 0x00000000030AC1A8>
```

- ***Note***：对于一个生成器对象，最重要的是它 **yield** 出的值，而不是它的返回 **(return)** 值
  - 程序无法直接得到生成器对象执行结束的返回值

生成器对象的使用 (1)

- 对生成器函数的每一次调用都得到一个新建的生成器对象
  - 用同一个生成器函数可以创建多个 、独立的生成器对象 (即彼此无关，各自迭代)
- 生成器对象可以作为 for 语句头部、各种推导式里的迭代器
  - 该生成器对象会被反复执行取其 yield 值，直到结束 (即遇到 return 语句，或执行完函数体代码)
  - 生成器对象的执行结束时，会抛出 StopIteration 异常，for 语句接收到该异常时退出循环

```
>>> list(fib_gen(15))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

```
>>> for x in fib_gen(5):
        for y in fib_gen(3):
            print((x, y), end=" ")
        print("")
```

```
(0, 0) (0, 1) (0, 1)
(1, 0) (1, 1) (1, 1)
(1, 0) (1, 1) (1, 1)
(2, 0) (2, 1) (2, 1)
(3, 0) (3, 1) (3, 1)
```

生成器对象的使用 (2)

- 独立使用生成器对象
  - 如果 g 的值是一个生成器对象 (或迭代器对象)，每次调用内置函数 next(g) 将得到该生成器对象 yield 出的下一个值

```
g = fib_gen(50) # 建立生成器对象并赋给 g

for i in range(10): # 输出 g yield 出的 1~10 项
    print(next(g), end=' ' if i<9 else '\n')

for i in range(39): # g yield 出 11~ 49 项
    next(g)

print(next(g))    # 输出 g yield 出的第 50 项

print(next(g))    # 此时报 StopIteration 异常
```

  - 内置函数 map, filter, zip, enumerate, reversed 也得到和生成器对象性质类似的对象，同样可以对其调用 next 函数

生成器对象的执行、数据抽象

- 一个生成器对象
  - 在被创建时，将建立其初始内部状态，包括其局部变量和 **yield** 语句的执行情况
  - 在执行过程中，随着 **yield** 出一个个值，其内部状态不断变化
  - 对其调用 **next** 函数也将改变其内部状态
- 一个生成器函数
  - 定义一类生成器对象，将按照同样规则生成 (独立的) 值序列

➔ 生成器函数是一种数据抽象机制

  - 可用来描述符合特定规律的值序列
  - 且无需暴露值序列的内部表示 (封装)
  - 支持对值序列的多种遍历

无穷生成器

- 无穷生成器：能产生任意多个值的生成器
  - 允许任意多次地对其调用 next 函数
  - 如果在 for 等上下文中使用，无穷生成器将导致无穷循环

```
#### 无穷生成器，生成任意多个唯一性文件名（字符串）
def id_gen(s):
    count = 0
    while True:
        yield str(s) + str(count).rjust(3,'0')
        count += 1

filename_gen = id_gen("tmp")
print(next(filename_gen))
print(next(filename_gen))
```

print(next(id_gen("a"))
print(next(id_gen("a"))
分别输出什么？

```
def primes():
    def is_prime(cand):
        for p in plist:
            if cand % p == 0: return False
            if p * p > cand: return True

    plist = [2]                 # 2 是第一个素数，其下标为 0
    yield 2                     # 送出第一个素数 2
    cand = 3
    while True:
        if is_prime(cand):
            plist.append(cand)
            yield cand          # 送出新找到的素数
        cand += 2

ps = primes()
for i in range(10000): next(ps)

print(next(ps))
print(next(ps))
```

yield 语句的其它形式

- **yield from *subiterator***
  - ***subiterator***：一个值为可迭代对象的表达式
  - 可简单地理解为：**for item in subiterable: yield item**
    - 即：执行 **yield from …** 语句时，该语句会把生成对象的工作委托给 ***subiterator***，它自己的工作只是逐个转发 ***subiterator*** 生成的值，直至 ***subiterator*** 结束
  - 可用于生成器的功能分解，或组合多个已有 ***iterator***

```
>>> def g(x):
        yield from range(x, 0, -1)
        yield from range(x)
```

```
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

itertools 模块简介

- itertools 模块提供了一组有趣、有用且高效的生成器函数

无穷迭代器：

迭代器	实参	结果	示例
count()	start, [step]	start, start+step, start+2*step, ...	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, ... plast, p0, p1, ...	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, ... 重复无限次或n次	repeat(10, 3) --> 10 10 10

排列组合迭代器：

迭代器	实参	结果
product()	p, q, ... [repeat=1]	笛卡尔积，相当于嵌套的for循环
permutations()	p[, r]	长度r元组，所有可能的排列，无重复元素
combinations()	p, r	长度r元组，有序，无重复元素
combinations_with_replacement()	p, r	长度r元组，有序，元素可重复

迭代器	实参	结果	示例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	accumulate([1,2,3,4,5]) --> 1 3 6 10 15
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	chain('ABC', 'DEF') --> A B C D E F
chain.from_iterable()	iterable -- 可迭代对象	p0, p1, ... plast, q0, q1, ...	chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], ... 从pred首次真值测试失败开始	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
filterfalse()	pred, seq	seq中pred(x)为假值的元素，x是seq中的元素。	filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
groupby()	iterable[, key]	根据key(v)值分组的迭代器	
islice()	seq, [start,] stop [, step]	seq[start:stop:step]中的元素	islice('ABCDEFG', 2, None) --> C D E F G
pairwise()	iterable -- 可迭代对象	(p[0], p[1]), (p[1], p[2])	pairwise('ABCDEFG') --> AB BC CD DE EF FG
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], ..., 直到pred真值测试失败	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, ... itn 将一个迭代器拆分为n个迭代器	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-