python 基础学习

November 23, 2020

1 Python 基础学习

1.1 数据类型:

1. False True 、None（0 是有意义的，None 不是 0，空值

2. dict

3. set 以看成数学意义上的无序和无重复元素的集合，因此，两个 set 可以做数学意义上的交集、并集等操作.

4. list* `list.append(obj)` 在列表末尾添加新的对象 `list.count(obj)` 统计某个元素在列表中出现的次数 `list.extend(seq)` 在列表末尾一次性追加另一个序列中的多个值 (用新列表扩展原来的列表) `list.index(obj)` 从列表中找出某个值第一个匹配项的索引位置 `list.insert(index, obj)` 将对象插入列表 `list.pop(obj=list[-1])` 移除列表中的一个元素 (默认最后一个元素), 并且返回该元素的值 `list.remove(obj)` 移除列表中某个值的第一个匹配项 `list.reverse()` 反向列表中元素 `list.sort([func])` 对原列表进行排序 二维 list, `list.sort(key = lambda x:[x[0],-[x1]] ,reverse=False)` # 对 0 升序、1 降序排序 **5.tuple 值不可改变**

1.2 collections

1. namedtuple (‘名称’, [属性 list])

`_make`（构造）、 `_replace`（更改值需要返回new） 、`_asdict()`（转dict） 、`_fields`（返回名字）

2. deque（双向队列） collections.deque([iterable[, maxlen]]) 超过 maxlen，再加入则旧数据从另一端弹出;

使用list存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为list是线性存储，数据量

deque除了实现list的append()和pop()外，还支持appendleft()和popleft()，这样就可以非常高效地往头部

`append(x)`	添加 x 到右端。
`appendleft(x)`	添加 x 到左端。
`clear()`	移除所有元素，使其长度为0.
`copy()`	创建一份浅拷贝。
`count(x)`	计算 deque 中元素等于 x 的个数。
`extend(iterable)`	扩展deque的右侧，通过添加iterable参数中的元素。
`extendleft(iterable)`	扩展deque的左侧，通过添加iterable参数中的元素。注意，左添加时，在结果

```
index(x[, start[, stop]])  返回 x 在 deque 中的位置（在索引 start 之后，索引 stop 之前）。
insert(i, x)     在位置 i 插入 x 。如果插入会导致一个限长 deque 超出长度 maxlen 的话，就引发
pop()            移去并且返回一个元素，deque 最右侧的那一个。 如果没有元素的话，就引发一个
popleft()        移去并且返回一个元素，deque 最左侧的那一个。 如果没有元素的话，就引发 Index
remove(value)    移除找到的第一个 value。 如果没有的话就引发 ValueError。
reverse()        将deque逆序排列。返回 None 。
rotate(n=1)      向右循环移动 n 步。 如果 n 是负数，就向左循环(环形)。如果deque不是空的，向
```

3. defaultdict 使用 dict 时，如果引用的 Key 不存在，就会抛出 KeyError。如果希望 key 不存在时，返回一个默认值，就可以用 defaultdict；也就是添加一个默认值。

4. OrderedDict 有序字典，同 py37 以上 dict

5. ChainMap 将多个映射快速的链接到一起，作为一个单元处理 (字典处理相当于 dict.update(dict_new)，但比其更快).

6. Counter 一个 Counter 是一个 dict 的子类，用于计数可哈希对象。它是一个集合，元素像字典键 (key) 一样存储，它们的计数存储为值。计数可以是任何整数值，包括 0 和负数。

```
elements()    返回一个迭代器，其中每个元素将重复出现计数值所指定次。 元素会按**首次出现**的
most_common([n])    返回一个列表，其中包含 n 个最常见的元素及出现次数，**按常见程度由高到低
subtract([iterable-or-mapping])    从 迭代对象 或 映射对象 减去元素。像 dict.update() 但是是
update([iterable-or-mapping])     从 迭代对象 计数元素或者 从另一个 映射对象 (或计数器) 添
```

[26]:
```
from collections import namedtuple
Circle = namedtuple('Circle', ['x', 'y','r'])
c = Circle(5,8,3)
print(isinstance(c, Circle),isinstance(c, tuple),c.x,c.y,c.r)# 自定义的也属于
tuple, 输出这个自定义 tuple 的各个值
new_t = Circle._make([1,2,10])
new_t = new_t._replace(x=11)
print(new_t._fields , new_t._asdict())
```

```
True True 5 8 3
('x', 'y', 'r') {'x': 11, 'y': 2, 'r': 10}
```

[ ]:
```
from collections import deque
q = deque(['a', 'b', 'c'])
q.append('x')
q.popleft()
q.appendleft('y')
print(q)
```

[ ]:
```
from collections import defaultdict
dd = defaultdict(lambda: 'N/A')
dd['key1'] = 'abc'
print(dd['key2']) # key2 不存在，返回默认值
```

[40]:
```
from collections import ChainMap
d1 = {'a':1,'b':2,'c':3}
```

```
d2 = {'a':2,'b':10,'d':5}
new_d = d1.copy()
new_d.update(d2)
print(list(new_d),new_d)
print(list(ChainMap(d1,d2)) ,ChainMap(d1,d2))
```

```
['a', 'b', 'c', 'd']
['a', 'b', 'd', 'c']
```

[23]:
```
from collections import Counter
c = Counter("abccc")# 初始化方式,("a"=1,'b'=2,'c'=3) or ("a":1,'b':2,'c':3)
print(c["k"])#0  返回该元素的个数
c.update(['a','b','q'])# 增加
print(c)
print(c.most_common(3) )
print(list(c.elements()))
c.subtract(['a','a','q','q'])
c
```

```
0
Counter({'c': 3, 'a': 2, 'b': 2, 'q': 1})
[('c', 3), ('a', 2), ('b', 2)]
['a', 'a', 'b', 'b', 'c', 'c', 'c', 'q']
```

[23]:
```
Counter({'a': 0, 'b': 2, 'c': 3, 'q': -1})
```

1.3 函数

1.lambda（匿名函数） lambda:f(x) = def:return f(x);
过滤: L = list(filter(lambda x:x%2 > 0, range(1, 20))) ， 1-20 奇数

2.map map(f,list) = f(i) for i in list;

3.reduce reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4) ; **from functools import reduce**

4.filter 和 map() 类似，filter() 也接收一个函数和一个序列。和 map() 不同的是，filter() 把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。

5.sorted 用法 sorted(a,key=None,reverse=False) 默认从小到大;

6. 偏函数 int2 = functools.partial(int, base=2) 即创建二进制字符串转 int 的函数 int2.

7. 装饰器利用 @fuc，相当于在该函数执行前执行了 fuc(f)

[2]:
```
# 如序列变为整数:
from functools import reduce
reduce(lambda x,y:10*x+y, [1,2, 3, 4, 5])#12345

# 思考: 写一个 str2int 且不用 int()。
```

```
DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8':␣
↪8, '9': 9}
def char2num(s):
    return DIGITS[s]
def str2int(s):
    return reduce(lambda x, y: x * 10 + y, map(char2num, s))
```

[2]: 12345

[9]:
```
#L = list(filter(lambda x:x%2 > 0, range(1, 20)))

# 用 Python 来实现这个算法，可以先构造一个从 3 开始的奇数序列：
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n
# 注意这是一个生成器，并且是一个无限序列。

# 然后定义一个筛选函数：
def _not_divisible(n):
    return lambda x: x % n > 0

# 最后，定义一个生成器，不断返回下一个素数：
def primes():
    yield 2
    it = _odd_iter() # 初始序列  3 5 7 9 11 ...
    while True:
        n = next(it) # 取出并返回序列的第一个数 3
        yield n
        it = filter(_not_divisible(n), it) # 更新序列，剔除掉能整除 3 的数   5 7␣
↪11 ...

# 这个生成器先返回第一个素数 2，然后，利用 filter() 不断产生筛选后的新的序列。
primes() 也是一个无限序列


# 打印 100 以内的素数：
for n in primes():
    if n < 10:
        print(n,end='\t')
    else:
        break
```

```
2	3	5	7
```

[17]:
```
import functools
int2 = functools.partial(int, base=2)
int2('1001001101')
```

[17]: 589

[20]:
```
import time

# 三层嵌套
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            now_time =  time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
            print('%s 开始时间:%s;执行函数 %s():' % (text,now_time,func.
 ↪__name__))
            return func(*args, **kw)
        return wrapper
    return decorator

@log('自定义')
def run(a):
    a = a+1
run(1)
```

```
自定义 开始时间:2020-11-20 21:01:40;执行函数 run():
```

1.4 class 类

1. 访问限制让内部属性不被外部访问，可以把属性的名称前加上两个下划线 ___，在 Python 中，实例的变量名如果以 ___ 开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问.

2. 继承和多态当定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）；多态的好处就是，当我们需要传入子类时，我们只需要接收基类就可以了，然后，按照 Animal 类型进行操作即可，执行的对象也是该子类的。

3. slots 在定义 class 的时候，定义一个特殊的 ___slots___ 变量，来限制该 class 实例能添加的属性. 除非在子类中也定义 ___slots___，这样，子类实例允许定义的属性就是自身的 ___slots___ 加上父类的 ___slots___。

4.@property 内部属性，不暴露出去; 把一个 getter 方法变成属性，只需要加上 @property 就可以了，此时，@property 本身又创建了另一个装饰器 @score.setter，负责把一个 setter 方法变成属性赋值.

5. 多重继承 MixIn class h(One,TwoMixIn)

6. 定制类改变类的固有函数 ___func___() 参考 https://www.liaoxuefeng.com/wiki/1016959663602400/1017590712

[29]:
```
class Base:
    def __init__(self,name = 'base'):
        self.name = name
        self.__test = 0
    def run(self):
        print("run %s"%self.name)

class One(Base):
    def __init__(self,name = 'one'):
        self.name = name
        self.__test = 0
    def run(self):
        print("run %s"%self.name)
def test(base):
    base.run()
test(Base())
test(One())
Base().__test
```

```
run base
run one


      ␣
↪---------------------------------------------------------------------------

       AttributeError                            Traceback (most recent call␣
↪last)

       <ipython-input-29-81fbddd91b7d> in <module>
        16 test(Base())
        17 test(One())
   ---> 18 Base().__test


       AttributeError: 'Base' object has no attribute '__test'
```

[30]:
```
# 只允许对 Student 实例添加 name 和 age 属性
class Student(object):
    __slots__ = ('name', 'age') # 用 tuple 定义允许绑定的属性名称
s = Student() # 创建新的实例
s.name = 'Michael' # 绑定属性'name'
s.age = 25 # 绑定属性'age'
s.score = 99 # 绑定属性'score'
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-30-a4f38a7ddf64> in <module>
  5 s.name = 'Michael' # 绑定属性'name'
  6 s.age = 25 # 绑定属性'age'
----> 7 s.score = 99 # 绑定属性'score'


AttributeError: 'Student' object has no attribute 'score'
```

[46]:
```
class Student(object):
    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value

    @property
    def real_score(self):
        return  int(self._score * 0.8)  # 只能通过 self._score 修改，所以为只读
s = Student()
s.score = 19
s.real_score
```

[46]: 15

1.5 生成器（generator）、迭代器（iterator）

1.5.1 1. 赋值生成器

创建方法：`x = (variable for variable in iterable)` 例如：`x = (i for i in range(10)) print(x) <generator object <genexpr> at 0x00000000006B85C8>` 返回值：generator # 使用元祖推导式的时候回变成一个生成器。

调用方法：x.___next___() 返回值：object # 对应生成器一般使用该种方法调用，当然也可以通过 for 循环进行遍历。

1.5.2 2. 函数生成器 yield

[6]:
```
#yield 生成器
def foo():
    print("starting...")
    while True:
        res = yield 4
        print("res:",res)
f = foo()
print('end1:',f.__next__())          # 会返回 4，此时中断于 res = yield 4 后；
print('end2:',next(f))          # 再次执行 next(f)，会接着执行 print("res:",res)
然后又到该句之前；
f.send(10)      # 执行 f.send(10)，  执行完，返回 4，且打印 res:10
```

```
starting…
end1: 4
res: None
end2: 4
res: 10
```

[6]:
```
4
```

1.5.3 3. 迭代器

迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。迭代器有两个基本的方法：iter() 和 next()。字符串，列表或元组对象都可用于创建迭代器

[15]:
```
a = [i for i in range(5)]
it_a = iter(a)# 创建迭代器对象，之后就可以用 next 方法
while True:
    try:
        print(next(it_a))
    except:
        print('取完了')
        break
# 上面已经取完了，要用需要重新新建对象 ;当然迭代器都能用 for  in; 同 next

'''
    创建迭代器
    StopIteration 异常用于标识迭代的完成，防止出现无限循环的情况，
    在 __next__() 方法中我们可以设置在完成指定循环次数后触发 StopIteration 异常来
结束迭代。
'''
class MyNumbers:
  def __iter__(self):
```

```
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x,end=',')
```

```
0
1
2
3
4
取完了
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
```

[ ]: