[笔记][Python高级编程-董伟明]

Python

Python 竟然没有 end

- Ruby 说：Python 竟然没有 end
- Python 说：其实***可以有***

```
# Python 可以有 end
__builtins__.end = None


def pos_or_neg(x):
    if x > 0:
        print(str(x) + '是正数')
    else:
        print(str(x) + '是负数')
    end
end


def main():
    pos_or_neg(1)
    print('Python 可以使用 end! ')
end
```

```
main()
"""
1是正数
Python 可以使用 end!
"""
```

设置全局变量

知识点

- **`globals()` 函数**
  - 以**字典类型**返回当前位置的全部全局变量
  - 即**返回全局变量的字典**
- **`update()` 方法**
  - 语法：`dict.update(dict2)`
  - 把字典 `dict2` 中的键值对更新到 `dict` 中

场景

- 变量不是直接赋值，而是存在于一个数据结构（比如字典中）。
- 或者是要从文件中提取出来，比如第一列是字母，第二列是数字。要让第一列作为变量名，第二列作为变量的值。

解决方法

- 此时可以使用 `exec()` 函数。

```
>>> d = {'a': 1, 'b': 2}
>>> # 粗暴的写法
... for k, v in d.items():
...     exec('{}={}'.format(k, v))
...
>>> a
1
>>> b
2
```

- 如果意识到 `globals()` 返回的是全局变量的字典，这样就可以配合字典的 `update()` 方法，写的更优雅一点。让字典的键作为变量名，字典的值作为变量名。

```
>>> d = {'a': 1, 'b': 2}
>>> # 文艺的写法
```

```
... globals().update(d)
>>> a, b
(1, 2)
>>> globals()['a'] = 'b'
>>> a, b
('b', 2)
```

示例代码

```
# 设置全局变量

# 粗暴的写法：通过 exec
data1 = {'a': 1, 'b': 2}
for k, v in data1.items():
    exec('{}={}'.format(k, v))
print('a=%s' % a)
print('b=%s' % b)
"""
a=1
b=2
"""


# 文艺的写法：对 globals() 返回的字典使用 update()
data2 = {'c': 3, 'd': 4}
globals().update(data2)
print('c=%s' % c)
print('d=%s' % d)
"""
c=3
d=4
"""

# 直接对 a 重新赋值
globals()['a'] = '你好'
print('a=%s' % a)
"""
a=你好
"""
```

字符串格式化

> 注：我对本条进行了补充
> 可以直接参考我的博客：https://blog.csdn.net/jpch89/article/details/84099277

列表去重

列表去重有字典取键法和集合法，后者比前者快。

```
from timeit import timeit
# timeit('语句'[, number=100000])
# timeit('函数名', 'from __main__ import 函数名'[, number=100000])
# timeit('函数名', globals=globals())

from random import randint
# randint(a, b) 生成 [a, b] 的随机数

old_li = [1, 2, 2, 3, 3, 3]
# 列表去重1：list({}.fromkeys(原列表).keys)
# 我叫做：字典取键法
# 如果是 Python 2 的话，最外层不需要用 list 转换
new_li = list({}.fromkeys(old_li).keys())
print(new_li)
"""
[1, 2, 3]
"""

# 列表去重2：list(set(原列表))
# 我叫做：集合法
new_li = list(set(old_li))
print(new_li)
"""
[1, 2, 3]
"""

# 性能分析1：原列表较短的情况
# 字典取键法
# t = timeit('list({}.fromkeys(old_li).keys())', 'from __main__ import ol
d_li')
t = timeit('list({}.fromkeys(old_li).keys())', globals=globals())
print(t)
"""
0.7955012980800552
"""
# 集合法
t = timeit('list(set(old_li))', globals=globals())
print(t)
"""
0.4911843005941092
"""

# 性能分析2：原列表较大的情况
big_li = [randint(1, 50) for i in range(10000)]
```

```
# 字典取键法
t = timeit('list({}.fromkeys(big_li).keys())',
    number=10000, globals=globals())
print(t)
"""
2.6739583454187015
"""
# 集合法
t = timeit('list(set(big_li))',
    number=10000, globals=globals())
print(t)
"""
1.1371806004635423
"""
```

字典取键法可以保持原列表中元素出现的先后顺序。
而集合法则不可以。

> 补充：关于字典的键顺序
> https://stackoverflow.com/questions/5629023/key-order-in-python-dictionaries
>
> Python 3.7+
> In Python 3.7.0 the insertion-order preservation nature of dict objects has been declared to be an official part of the Python language spec. Therefore, you can depend on it.
>
> Python 3.6 (CPython)
> As of Python 3.6, for the CPython implementation of Python, dictionaries maintain insertion order by default. This is considered an implementation detail though; you should still use collections.OrderedDict if you want insertion ordering that's guaranteed across other implementations of Python.
>
> Python <3.6
> Use the collections.OrderedDict class when you need a dict that remembers the order of items inserted.

还有一种方法是**有序字典取键**法：

```
l = [1, 2, 4, 7, 2, 1, 8, 6, 1]
# 字典取键法可以保证原列表元素顺序
print(list(dict.fromkeys(l).keys()))
# 集合法则不可以
print(list(set(l)))

# 还有一种方法：有序字典取键法
# 其实在 Python 3.6 及后续版本这个方法等价于字典取键法
from collections import OrderedDict
print(list(OrderedDict.fromkeys(l).keys()))
```

操作字典

> 其实本节应该叫做**生成字典**比较好，因为讲了各种生成字典的姿势。

- 使用 `dict(可迭代对象)` 构造字典
- 使用 `dict.fromkeys(可迭代对象[, 默认值])` 构造字典
- **字典解析**构造字典
- **花括号语法**构造字典

```
# 使用可迭代对象构造字典
d = dict((['a', 1], ['b', 2]))
print(d)
"""
{'a': 1, 'b': 2}
"""

# zip 对象也是可迭代的，所以可以用来构造字典
d = dict(zip('ab', range(2)))
print(d)
"""
{'a': 0, 'b': 1}
"""

# map 对象同样也是可迭代的
m = map(lambda x: (x, x ** 2), [2, 3])
from collections import Iterable
print(isinstance(m, Iterable))
"""
True
"""
d = dict(m)
print(d)
"""
{2: 4, 3: 9}
"""

# 注意，map 对象使用过一次之后就没有值了
# 因为已经迭代完毕
m = map(lambda x: x, range(5))
print(list(m))
print(list(m))
"""
[0, 1, 2, 3, 4]
[]
```

```
"""

# dict.fromkeys 方法生成字典
# 需要传递一个可迭代对象
d = dict.fromkeys('abc')
print(d)
"""
{'a': None, 'b': None, 'c': None}
"""

# 还可以提供默认值
d = {}.fromkeys(range(3), '呵呵')
print(d)
"""
{0: '呵呵', 1: '呵呵', 2: '呵呵'}
"""

# 如果提供列表作为默认值
# 则字典所有的值都指向一个列表
d = {}.fromkeys('ab', [1])
print(d)
d['a'].append('变')
print(d)
"""
{'a': [1], 'b': [1]}
{'a': [1, '变'], 'b': [1, '变']}
"""

# 字典解析生成字典
d = {k: v for k, v in zip('abc', range(3))}
print(d)
"""
{'a': 0, 'b': 1, 'c': 2}
"""

# 花括号语法生成字典
d = {'name': '狗子', 'age': 2}
print(d)
"""
{'name': '狗子', 'age': 2}
"""
```

然后讲了 `dict.setdefault(键, 值)` 的用法。

- 如果 `键` 存在，返回它对应的值
- 如果 `键` 不存在，先设置 `值`，然后返回 `值`

```
# setdefault(键, 默认值) 的用法
d = {'苹果': 6, '香蕉': 8}
ret = d.setdefault('苹果', 0)
```

```
print(f'苹果有{ret}个! ')
ret = d.setdefault('火龙果', 0)
print(f'火龙果有{ret}个! ')
"""
苹果有6个!
火龙果有0个!
"""
```

字典视图

在 `Python 3` 中，`viewitems` 和 `iteritems` 都变成了 `items`
也就是说不存在字典视图这个东西。

使用 `dict.items()` 方法得到的结果就可以进行集合运算。
运算结果是一个集合，需要类型转换才能变成字典。

```
d1 = dict(a=1, b=2)
d2 = dict(b=2, c=3)
print(d1)
print(d2)
"""
{'a': 1, 'b': 2}
{'b': 2, 'c': 3}
"""

# 尝试取交集，失败!
# print(d1 & d2)
"""
TypeError: unsupported operand type(s) for &: 'dict' and 'dict'
"""

# 但是使用 items 可以
# 注意：Python 2 中是 viewitems
i1 = d1.items()
i2 = d2.items()

# 交集
print(i1 & i2)
# 得到一个集合
print(type(i1 & i2))
"""
{('b', 2)}
<class 'set'>
"""
# 需要手动转换成字典
res = dict(i1 & i2)
print(res)
```

```
# 并集
print(dict(i1 | i2))
"""
{'c': 3, 'a': 1, 'b': 2}
"""

# 差集（仅 i1 有，i2 没有的）
print(dict(i1 - i2))
"""
{'a': 1}
"""

# 对称差集（不同时出现于 i1 和 i2 中）
print(dict(i1 ^ i2))
"""
{'c': 3, 'a': 1}
"""
```

---

vars

```
# vars() 不传递参数就跟 locals() 是一样的
# 它返回当前所有局部变量的字典
print(vars() is locals())
"""
True
"""

import sys
# vars(obj) 相当于 obj.__dict__
print(vars(sys) is sys.__dict__)
"""
True
"""
```

---

unicode_literals

`from __future__ import unicode_literals`
这是 `Python 2` 的用法，此处略过

不是支持了绝对引入，而是拒绝隐式引入

`from __future__ import absolute_import`
在 `Python 2` 中，`import` 默认是相对导入，不写 `.` 和 `..` 的则称为隐式的相对导入。
可以通过上面那句话，禁用掉隐式的相对导入，从而都使用绝对导入。

一个关于编码的问题

省略

---

super 子类调用父类

```
class LogginDict(dict):
    def __setitem__(self, key, value):
        print('设置{0}为{1}。'.format(key, value))
        dict.__setitem__(key, value)

# 假如后续要修改需要继承的类
# 那么要在类定义处修改一遍，还要在重写的方法里面修改一遍
# 容易忘记

# 更好的做法：
class LogginDict(dict):
    def __setitem__(self, key, value):
        print('设置{0}为{1}。'.format(key, value))
        super(LogginDict, self).__setitem__(key, value)
        # Python 3 写法
        # super().__setitem__(key, value)
```

迭代器

```
# 借助 list 的迭代器
class Data(object):
    def __init__(self):
        self._data = []
```

```
    def add(self, x):
        self._data.append(x)
    def data(self):
        return iter(self._data)

d = Data()
d.add(1)
d.add(2)
d.add(3)
for x in d.data():
    print(x)
"""
1
2
3
"""

# 标准迭代器
``` python
class Data(object):
    def __init__(self, *args):
        self._data = list(args)
        self._index = 0

    def __iter__(self):
        return self

    # Python 3 写法
    def __next__(self):
        if self._index >= len(self._data):
            raise StopIteration()
        d = self._data[self._index]
        self._index += 1
        return d

    # 兼容 Python 2
    def next(self):
        return self.__next__()

d = Data(1, 2, 3)
for x in d:
    print(x)
"""
1
2
3
"""
```
```