13-字典与集合

大纲:

- 字典的现代句法
- 使用 match\case 语句匹配映射
- 映射类型的标准API
- 自动处理缺失的键
- dict 的变体
- 集合论

字典的现代句法

创建字典的一些方法:

- 字典推导
- dict 函数
- 映射拆包
- 合并两个字典

字典推导

In [13]:
```
dial_codes = [                                          # <1>
    (880, 'Bangladesh'),
    (55,  'Brazil'),
    (86,  'China'),
    (91,  'India'),
    (62,  'Indonesia'),
    (81,  'Japan'),
    (234, 'Nigeria'),
    (92,  'Pakistan'),
    (7,   'Russia'),
    (1,   'United States'),
]

country_dial = {country: code for code, country in dial_codes}
country_dial
```

Out[13]:
```
{'Bangladesh': 880,
 'Brazil': 55,
 'China': 86,
 'India': 91,
 'Indonesia': 62,
 'Japan': 81,
 'Nigeria': 234,
 'Pakistan': 92,
 'Russia': 7,
 'United States': 1}
```

In [12]:
```
{code: country.upper()
    for country, code in sorted(country_dial.items())
    if code < 70}
```

Out[12]:
```
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'}
```

dict 函数, 成对的序列数据可以直接转换成 dict

In [9]:
```
dict(dial_codes)
```

Out[9]:
```
{880: 'Bangladesh',
 55: 'Brazil',
 86: 'China',
 91: 'India',
 62: 'Indonesia',
 81: 'Japan',
 234: 'Nigeria',
 92: 'Pakistan',
 7: 'Russia',
 1: 'United States'}
```

映射拆包

In [14]:
```
# 函数参数传入任意个键值对，返回包含所有键值对的dict
def dump(**kwargs):
    return kwargs

dump(x=1, y=2, z=3)
```

Out[14]:
```
{'x': 1, 'y': 2, 'z': 3}
```

In [15]:
```
dump(a=1, b=2, **{'y' : 3, 'z' : 4}, )
```

Out[15]:
```
{'a': 1, 'b': 2, 'y': 3, 'z': 4}
```

在Python字面量中使用**拆包

In [16]:
```
{'a': 0, **{'x': 1}, 'y': 2, **{'z': 3, 'x': 4}}
```

Out[16]:
```
{'a': 0, 'x': 4, 'y': 2, 'z': 3}
```

使用 | 合并映射

In [17]:
```
d1 = {'a': 1, 'b': 3}
d2 = {'a': 2, 'b': 4, 'c': 6}
d1 | d2
```

Out[17]:
```
{'a': 2, 'b': 4, 'c': 6}
```

使用match/case语句匹配映射

match/case 语句能够匹配 collections.abc.Mapping 的任何具体子类或虚拟子类, 我们经常需要从 JSON API和具有半结构化模式的数据库（例如 MongoDB、EdgeDB 或 PostgreSQL）中读取这类记录.

In [1]:
```
def get_creators(record: dict) -> list:
    match record:
        # 匹配'type'为'book', 'api'为2, 'authors'对应一个作者列表，返回该作者列表
        case {'type': 'book', 'api': 2, 'authors': [*names]}:  # <1>
            return names

        # 匹配'type'为'book', 'api'为2, 'author'对应一个作者，返回一个作者的列表
```

```
        case {'type': 'book', 'api': 1, 'author': name}:  # <2>
            return [name]

        # 匹配'type'为'book', 其他键值没有匹配的情况, 抛出ValueError
        case {'type': 'book'}:  # <3>
            raise ValueError(f"Invalid 'book' record: {record!r}")

        # 匹配'type'为'movie', 'director'对应导演的名字, 返回该导演的列表
        case {'type': 'movie', 'director': name}:  # <4>
            return [name]
        case _:  # <5>
            raise ValueError(f'Invalid record: {record!r}')
```

In [2]:
```
# 匹配到一个book作者的情况
b1 = dict(api=1, author='Douglas Hofstadter',
          type='book', title='Gödel, Escher, Bach')
get_creators(b1)
```

Out[2]:
```
['Douglas Hofstadter']
```

In [3]:
```
# 匹配到book作者列表的情况
from collections import OrderedDict
b2 = OrderedDict(api=2, type='book',
          title='Python in a Nutshell',
          authors='Martelli Ravenscroft Holden'.split())
get_creators(b2)
```

Out[3]:
```
['Martelli', 'Ravenscroft', 'Holden']
```

In [ ]:
```
# book没有作者时, 抛出异常
get_creators({'type': 'book', 'pages': 770})
```

In [ ]:
```
# 没有匹配到任意一种情况（case分支）
get_creators('Spam, spam, spam')
```

映射类型的标准API

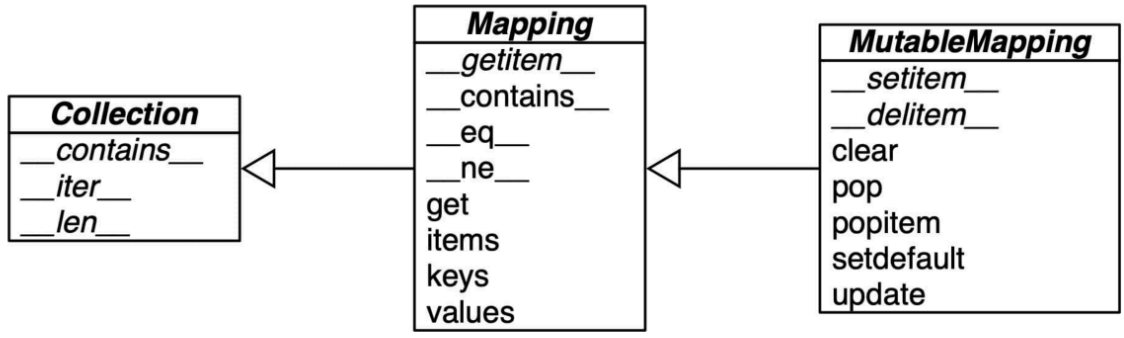

图 3-1：`MutableMapping` 及其在 `collections.abc` 中的超类的简化 UML 类图

“可哈希”指什么

如果一个对象是可哈希的：

- 如果一个对象的哈希码在整个生命周期内永不改变（依托 `__hash__()` 方法）
- 而且可与其他对象比较（依托 `__eq__()` 方法）

- 而且可与其他对象比较（依托 __eq__() 方法）
- 两个可哈希对象的哈希码相同，这两个对象相等
- 如果两个可哈希对象相等，它们的哈希码一定相同
- 两个不相等的可哈希对象，哈希码有可能相同，这种情况称为哈希冲突（Hash collision）

In [3]:
```
# 元组是可哈希的
tt = (1, 2, (30, 40))
hash(tt)
```

Out[3]:
```
-3907003130834322577
```

In [4]:
```
# 列表对象不可哈希
tl = (1, 2, [30, 40])
hash(tl)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 tl = (1, 2, [30, 40])
----> 2 hash(tl)

TypeError: unhashable type: 'list'
```

In [5]:
```
tf = (1, 2, frozenset([30, 40]))
hash(tf)
```

Out[5]:
```
5149391500123939311
```

- 一个对象的哈希码根据所用的 Python 版本和设备架构有所不同。如果出于安全考量而在哈希计算过程中加盐，那么哈希码也会发生变化。正确实现的对象，其哈希码在一个 Python 进程内保持不变。
- 默认情况下，用户定义的类型是可哈希的，因为自定义类型的哈希码取自 id()，而且继承自 object 类的 __eq__() 方法只不过是比较对象 ID。

常用的映射类型：

- dict
- collections.defaultdict
- collections.OrderedDict
- collections.ChainMap
- collections.Counter
- shelve.Shelf
- collections.UserDict

自动处理缺失的键

有时搜索的键不一定存在，为了以防万一，可以人为设置一个值，以方便某些情况的处理。

- get时使用default参数，setdefault方法
- 把普通的 dict 换成 defaultdict；
- 是定义 dict 或其他映射类型的子类，实现 __missing__ 方法。

In [6]:

In [6]:
```
d = {1: 'Hello', 2: 'For', 3: 'Geeks'}
d[4]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[6], line 2
      1 d = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
----> 2 d[4]

KeyError: 4
```

In [7]:
```
from collections import defaultdict

# 返回默认值的函数
def def_value():
    return "Not Present"

# Defining the dict
d = defaultdict(def_value)
d["a"] = 1
d["b"] = 2

print(d["a"])
print(d["b"])
print(d["c"])
```

```
1
2
Not Present
```

In [8]:
```
from collections import defaultdict

# Using List as default_factory
# list函数返回的是[]
dd = defaultdict(list)

for i in range(5):
    dd[i].append(i)

print("Dictionary with values as list:")
print(dd)
```

```
Dictionary with values as list:
defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]})
```

In [9]:
```
from collections import defaultdict

# Using int as default_factory
# int函数默认返回0
dd = defaultdict(int)

L = [1, 2, 3, 4, 2, 4, 1, 2]

for i in L:
    # The default value is 0
    dd[i] += 1

print(dd)
```

```
defaultdict(<class 'int'>, {1: 2, 2: 3, 3: 1, 4: 2})
```

In [10]:
```
from collections import defaultdict

# Defining the dict
d = defaultdict(lambda: "Not Present")
```

```
d["a"] = 1
d["b"] = 2

# 当key缺失时会自动调用默认方法__missing__
print(d.__missing__('a'))
print(d.__missing__('d'))
```

```
Not Present
Not Present
```

In [15]:

```
# 自定义类继承dict，并实现__missing__方法
class My_Dict_Subclass(dict):
    def __missing__(self, key):
        return 'My default value'

x = {'Alice': 23, 'Bob': 24, 'Carl': 25}
my_dict = My_Dict_Subclass(x)

print(my_dict['Bob'])
# 24

# Try accessing missing key:
print(my_dict['David'])
# 'My default value'
```

```
24
My default value
```

dict 的变体

- collections.OrderedDict: 保持元素的键的顺序
- collections.ChainMap: 存放一组映射，可作为一个整体来搜索
- collections.Counter: 计数器，对键进行计数的映射
- collections.UserDict: 用于创建用户自定义的字典

collections.OrderedDict

自 Python 3.6 起，内置的 dict 也保留键的顺序。使用 OrderedDict 最主要的原因是编写与早期Python 版本兼容的代码。不过，dict 和 OrderedDict 之间还有一些差异:

- OrderedDict 的等值检查考虑顺序。
- OrderedDict 的 popitem()方法签名不同，可通过一个可选参数指定移除哪一项。
- OrderedDict 多了一个 move_to_end() 方法，便于把元素的位置移到某一端。
- 常规的 dict 主要用于执行映射操作，插入顺序是次要的。
- OrderedDict 的目的是方便执行重新排序操作，空间利用率、迭代速度和更新操作的性能是次要的。
- 从算法上看，OrderedDict 处理频繁重新排序操作的效果比 dict 好，因此适合用于跟踪近期存取情况（例如在 LRU 缓存中）。

collections.ChainMap

In [8]:

```
d1 = dict(a=1, b=3)
d2 = dict(a=2, b=4, c=6)

from collections import ChainMap
chain = ChainMap(d1, d2)
```

```
chain = ChainMap(d1, d2)
print(chain)
```

```
ChainMap({'a': 1, 'b': 3}, {'a': 2, 'b': 4, 'c': 6})
```

In [ ]:
```
print(chain['a'])
print(chain['c'])
```

In [11]:
```
# ChainMap 的更新或插入操作只影响第一个输入映射
chain['c'] = -1
chain
```

Out[11]:
```
ChainMap({'a': 1, 'b': 3, 'c': -1}, {'a': 2, 'b': 4, 'c': 6})
```

ChainMap 可用于实现支持嵌套作用域的语言解释器，按嵌套层级从内到外，一个映射表示一个作用域上下文。collections 文档中的"ChainMap objects"一节举了几个ChainMap 用法示例，其中一个就是模仿 Python 查找变量的基本规则，如下所示:

In [12]:
```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

collections.Counter

In [1]:
```
from collections import Counter
numbers = [1, 3, 1, 1, 1, 2, 3, 4]
Counter(numbers)
```

Out[1]:
```
Counter({1: 4, 3: 2, 2: 1, 4: 1})
```

子类应继承 UserDict 而不是 dict

子类最好继承 UserDict 的主要原因是，内置的 dict 在实现上走了一些捷径，如果继承 dict，那就不得不覆盖一些方法，而继承UserDict 则没有这些问题。

In [ ]:
```
# 自定义字典最好继承自collections.UserDict
import collections
class My_UserDict_Subclass(collections.UserDict):
    def __missing__(self, key):
        return 'Default value'

x = {'Alice': 23, 'Bob': 24, 'Carl': 25}
my_dict2 = My_UserDict_Subclass()

print(my_dict2['Z'])
```

集合论

- 集合主要使用的是set 和 frozenset。
- set是可变的，frozenset不可变。
- 集合的基本作用是去除重复项。
- 集合元素必须是可哈希的对象。set 类型不可哈希，因此不能构建嵌套 set 实例的 set对象。但是 frozenset 可以哈希，所以 set 对象可以包含 frozenset 元素。

- 集合类型通过中缀运算符实现了许多集合运算。给定两个集合 a 和b，a | b 计算并集，a & b 计算交集，a - b 计算差集，a ^ b 计算对称差集

In [13]:

```
fruits = frozenset(["apple", "banana", "orange"])
print(fruits)
fruits.append("pink")
print(fruits)
```

```
frozenset({'apple', 'orange', 'banana'})
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[13], line 3
      1 fruits = frozenset(["apple", "banana", "orange"])
      2 print(fruits)
----> 3 fruits.append("pink")
      4 print(fruits)

AttributeError: 'frozenset' object has no attribute 'append'
```

In [17]:

```
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])

print(A | B)
print(A & B)
print(A - B)
print(A ^ B)
```

```
frozenset({1, 2, 3, 4, 5, 6})
frozenset({3, 4})
frozenset({1, 2})
frozenset({1, 2, 5, 6})
```