??? ????????? ??????? ? Python

Словари являются одной из наиболее важных и полезных структур данных в Python. Они могут помочь вам решить широкий спектр задач программирования. Из этой статьи вы узнаете, как итерировать словарь в Python.

К концу этой статьи вы узнаете:

- Что такое словари, а также некоторые их основные функции и детали реализации
- Как итерировать словарь в Python, используя основные инструменты, предлагаемые языком
- Какие реальные задачи вы можете выполнить, просматривая словарь в Python
- Как использовать некоторые более продвинутые методы и стратегии для перебора словаря в Python

Для получения дополнительной информации о словарях, вы можете обратиться к следующим ресурсам:

- Dictionaries in Python
- Itertools in Python 3, By Example
- The documentation for `map()` and `filter()`

Готовы? Поехали!

????????? ???? ? ????????

Словари являются краеугольным камнем Python. Сам язык построен вокруг словарей. Модули, классы, объекты, globals(), locals(): все это словари. Словари были центральным элементом для Python с самого начала.

Официальная документация Python определяет словарь следующим образом:

> Ассоциативный массив, где произвольные ключи отображаются на значения. Ключами могут быть любые объекты с методами __hash__() и __eq__(). (Источник)

Есть несколько моментов, о которых следует помнить:

1. Словари сопоставляют ключи со значениями и сохраняют их в массиве или коллекции.
2. Ключи должны быть хэшируемого типа (hashable), что означает, что в качестве ключа используют хэш значения ключа, который никогда не меняется в течение срока жизни ключа.

Словари часто используются для решения всевозможных задач программирования, поэтому они являются фундаментальной частью инструментария разработчика Python.

В отличие от последовательностей (**sequences**), которые так же являются итерационными поддерживающими доступ к элементам с использованием целочисленных индексов, словари индексируются по ключам.

Ключи в словаре очень похожи на множества `set`, представляющие собой коллекцию уникальных объектов, которые можно хэшировать. Поскольку объекты должны быть хешируемыми, изменяемые (mutable) объекты нельзя использовать в качестве ключей словаря.

С другой стороны, значения могут быть любого типа Python, независимо от того, являются они хэшируемыми или нет. Там нет буквально никаких ограничений.

В Python 3.6 и более поздних версиях ключи и значения словаря перебираются в том же порядке, в котором они были созданы. Однако это поведение может отличаться в разных версиях Python и зависит от истории вставок и удалений словаря.

В Python 2.7 словари являются неупорядоченными структурами. Порядок элементом словарей неизменяемый. Это означает, что порядок элементов является детерминированным и повторяемым. Давайте посмотрим на пример:

```
>>> # Python 2.7
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Если вы покинете интерпритатор и позже откроете новый интерактивный сеанс, вы получите тот же порядок элементов:

```
>>> # Python 2.7. New interactive session
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
```

```
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Более внимательное рассмотрение этих двух выходных данных показывает, что результирующий порядок в обоих случаях одинаков. Вот почему вы можете сказать, что порядок является детерминированным.

В Python 3.5 словари все еще неупорядочены, но на этот раз **рандомизированные** структуры данных. Это означает, что каждый раз, когда вы снова запускаете словарь, вы получаете другой порядок элементов. Давайте посмотрим:

```
>>> # Python 3.5
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
>>> a_dict
{'color': 'blue', 'pet': 'dog', 'fruit': 'apple'}
```

Если вы войдете в новый интерактивный сеанс, вы получите следующее:

```
>>> # Python 3.5. New interactive session
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'fruit': 'apple', 'pet': 'dog', 'color': 'blue'}
>>> a_dict
{'fruit': 'apple', 'pet': 'dog', 'color': 'blue'}
```

На этот раз вы можете видеть, что порядок элементов отличается на обоих выходах. Вот почему вы можете сказать, что это рандомизированные структуры данных.

В Python 3.6 и более **поздних версиях словари являются упорядоченными структурами данных**, что означает, что они хранят свои элементы в том же порядке, в котором они были созданы, как вы можете видеть здесь:

```
>>> # Python 3.6 and beyond
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
```

Это относительно новая функция словарей Python, и она очень полезна. Но если вы пишете код, который может быть запущен в разных версиях Python, вы не должны полагаться на этот функционал, поскольку он может генерировать некорректное поведение.

Другой важной особенностью словарей является то, что они являются изменчивыми структурами данных, что означает, что вы можете добавлять, удалять и обновлять их элементы. Стоит отметить, что это также означает, что их нельзя использовать в качестве ключей для других словарей, так как они не являются объектами, которые можно хэшировать.

Примечание. Все, что описано в этом разделе, относится к базовой реализации Python, CPython.

Другие реализации Python, такие как PyPy, IronPython или Jython, могут демонстрировать другое поведение словаря, которые выходят за рамки данной статьи.

??? ??????????? ???????? ? Python: ??????

Словари являются полезной и широко используемой структурой данных в Python. Как программист Python, вы часто будете в ситуациях, когда вам придется перебирать словарь, в то время как вы выполняете некоторые действия над его парами ключ-значение.

Когда дело доходит до перебора словаря в Python, язык предоставляет вам несколько отличных инструментов, которые мы рассмотрим в этой статье.

?????? ???????? ?? ??????

Словари Python являются отображающими объектами. Это означает, что они наследуют некоторые **специальные методы**, которые Python внутренне использует для выполнения некоторых операций. Эти методы называются с использованием соглашения об именах, заключающегося в добавлении двойного подчеркивания в начале и в конце имени метода.

Чтобы визуализировать методы и атрибуты любого объекта Python, вы можете использовать **dir()**, которая является встроенной функцией. Если вы запустите **dir()** с пустым словарем в качестве аргумента, вы сможете увидеть все методы и атрибуты, которые реализуют словари:

```
>>> dir({})
['__class__', '__contains__', '__delattr__', ... , '__iter__', ...]
```

Если вы внимательно посмотрите на предыдущий вывод, вы увидите **__iter__**. Это метод, который вызывается, когда для контейнера требуется итератор, и он должен возвращать новый объект итератора, который может выполнять итерацию по всем объектам в контейнере.

Примечание: вывод предыдущего кода был сокращен (...) для экономии места.

Для мапинга (например, словарей) **.__iter__()** должен перебирать ключи. Это означает, что если вы поместите словарь непосредственно в цикл `for`, Python автоматически вызовет **.__iter__()** для этого словаря, и вы получите итератор по его ключам:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> for key in a_dict:
... print(key)
...
color
fruit
pet
```

Python достаточно умен, чтобы знать, что **a_dict** – это словарь и что он реализует **.__iter__()**. В этом примере Python автоматически вызывает **.__iter__(),** и это позволяет вам перебирать ключи **a_dict**.

Это самый простой способ перебора словаря в Python. Просто поместите его прямо в цикл **for**, и все готово!

Если вы используете этот подход вместе с небольшой уловкой, то вы можете обрабатывать ключи и значения любого словаря. Хитрость заключается в использовании оператора индексации [] со словарем и его ключами для получения доступа к значениям:

```
>>> for key in a_dict:
... print(key, '->', a_dict[key])
...
color -> blue
fruit -> apple
pet -> dog
```

Предыдущий код позволил вам получить доступ к ключам (**key**) и значениям (**a_dict[key]**) **a_dict** одновременно. Таким образом, вы можете выполнить любую операцию как с ключами, так и со значениями.

???????? ?? .items()

Когда вы работаете со словарями, вероятно, вы захотите работать как с ключами, так и со значениями. Одним из наиболее полезных способов перебора словаря в Python является использование метода **.items()**, который возвращает новый вид элементов словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> d_items = a_dict.items()
>>> d_items # Here d_items is a view of items
dict_items([('color', 'blue'), ('fruit', 'apple'), ('pet', 'dog')])
```

Представления словаря, такое как **d_items**, обеспечивают динамическое представление записей словаря, что означает, что при изменении словаря представления отражают эти изменения.

Представления могут быть перебраны для получения соответствующих данных, поэтому вы можете итерировать словарь в Python, используя объект представления, возвращаемый **.items()**:

```
>>> for item in a_dict.items():
... print(item)
...
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
```

Объект представления, возвращаемый функцией **.items()**, выдает пары ключ-значение по одной и позволяет перебирать словарь, и таким образом, вы получаете доступ к ключам и значениям одновременно.

Если вы присмотритесь к отдельным элементам, полученным с помощью **.items()**, вы заметите, что они действительно являются кортежами объектов. Давайте посмотрим:

```
>>> for item in a_dict.items():
... print(item)
... print(type(item))
...
('color', 'blue')
<class 'tuple'>
('fruit', 'apple')
<class 'tuple'>
('pet', 'dog')
<class 'tuple'>
```

Вы можете использовать распаковку кортежей для перебора ключей и значений словаря. Для этого вам просто нужно распаковать элементы каждого элемента в две разные переменные, представляющие ключ и значение:

```
>>> for key, value in a_dict.items():
... print(key, '->', value)
```

```
...
color -> blue
fruit -> apple
pet -> dog
```

Здесь, переменные **key** и **value** в заголовке вашего цикла **for** распаковываются. Каждый раз, когда цикл запускается, **key** будет хранить ключ, а **value** будет хранить значение элемента, который был обработан. Таким образом, у вас будет больше контроля над элементами словаря, и вы сможете обрабатывать ключи и значения отдельно.

Примечание: обратите внимание, что **.values()** и **.keys()** возвращают объекты представления так же, как **.items()**, как вы увидите в следующих двух разделах.

???????? ????? .keys()

Если вам просто нужно работать с ключами словаря, то вы можете использовать метод **.keys()**, который возвращает новый объект представления, содержащий ключи словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> keys = a_dict.keys()
>>> keys
dict_keys(['color', 'fruit', 'pet'])
```

Чтобы перебрать словарь в Python с помощью **.keys()**, вам просто нужно вызвать **.keys()** в заголовке цикла for:

```
>>> for key in a_dict.keys():
... print(key)
...
color
fruit
pet
```

Используя тот же трюк, который вы видели ранее (оператор индексации []), вы можете получить доступ к значениям словаря:

```
>>> for key in a_dict.keys():
... print(key, '->', a_dict[key])
...
color -> blue
fruit -> apple
pet -> dog
```

Таким образом, вы получите доступ к ключам (**key**) и значениям (**a_dict[key]**) **a_dict** одновременно, и вы сможете выполнять с ними любые действия.

???????? ?? .values()

Также можно использовать значения для перебора словаря. Один из способов сделать это – использовать **.values()**, который возвращает представление со значениями словаря:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> values = a_dict.values()
>>> values
dict_values(['blue', 'apple', 'dog'])
```

В предыдущем коде **values** содержит ссылку на объект представления, содержащий значения **a_dict**.

Как и любой объект представления, объект, возвращаемый функцией **.values()**, также может быть итерирован. В этом случае **.values()** возвращает значения **a_dict**:

```
>>> for value in a_dict.values():
... print(value)
...
blue
apple
dog
```

Стоит отметить, что методы **keys()** и **values()** также поддерживают тесты членства (in) (membership tests (`in`)), что является важной функцией, если вы пытаетесь узнать, есть ли определенный элемент в словаре или нет:

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> 'pet' in a_dict.keys()
True
>>> 'apple' in a_dict.values()
True
>>> 'onion' in a_dict.values()
False
```

Проверка членства с помощью **in** возвращает **True**, если ключ (или значение или элемент) присутствует в тестируемом словаре, и возвращает **False** в противном случае. Тест на членство позволяет вам не выполнять итерацию по словарю в Python, если вы просто хотите узнать, присутствует ли определенный словарь (или значение, или элемент) в словаре или нет.

????????? ???????? ? ??????

Часто бывает необходимо изменить значения и ключа, когда вы перебираете словарь в Python. Есть некоторые моменты, которые вы должны принять во внимание, чтобы выполнить эту задачу.

Например, значения можно изменять всякий раз, когда вам нужно, но вам нужно будет использовать исходный словарь и ключ, который отображает значение, которое вы хотите изменить:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for k, v in prices.items():
... prices[k] = round(v * 0.9, 2) # Apply a 10% discount
...
>>> prices
{'apple': 0.36, 'orange': 0.32, 'banana': 0.23}
```

В этом примере кода, чтобы изменить значения **prices** и применить скидку 10%, мы использовали выражение **prices[k] = round(v * 0.9, 2)**.

Так зачем вам использовать оригинальный словарь, если у вас есть доступ к его ключу (**k)** и его значениям (**v**)? Если мы можем изменить их напрямую?

Реальная проблема заключается в том, что изменения **k** и **v** не отражаются в исходном словаре. То есть, если вы измените какой-либо из них (**k** или **v**) непосредственно внутри цикла, то, что действительно происходит, так это то, что вы потеряете ссылку на соответствующий компонент словаря, не изменяя ничего в словаре.

С другой стороны, ключи могут быть добавлены или удалены из словаря путем преобразования представления, возвращаемого функцией **.keys()**, в объект **list**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for key in list(prices.keys()): # Use a list instead of a view
... if key == 'orange':
... del prices[key] # Delete a key from prices
...
>>> prices
{'apple': 0.4, 'banana': 0.25}
```

Этот подход может иметь некоторые последствия для производительности, в основном связанные с потреблением памяти. Например, вместо объекта просмотра, который выдает элементы по требованию, у вас будет полный новый **list** в памяти вашей системы. Тем не менее, это может быть безопасным способом изменения ключей при переборе словаря в Python.

Наконец, если вы попытаетесь удалить ключ из **prices**, используя напрямую **.keys()**, тогда Python вызовет **RuntimeError**, сообщающую, что размер словаря изменился во время итерации:

```
>>> # Python 3. dict.keys() returns a view object, not a list
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> for key in prices.keys():
... if key == 'orange':
... del prices[key]
...
Traceback (most recent call last):
File "<input>", line 1, in <module>
for key in prices.keys():
RuntimeError: dictionary changed size during iteration
```

Это потому, что **.keys()** возвращает объект словаря-представления, который выдает ключи по запросу по одному, и если вы удаляете элемент (**del values [key]**), то Python вызывает **RuntimeError**, потому что вы изменили словарь во время итерации.

Примечание. В Python 2 объекты **.items()**, **.keys()** и **.values()** возвращают список объектов. Но **.iteritems()**, **iterkeys()** и **.itervalues()** возвращают итераторы. Итак, если вы используете Python 2, то вы можете изменить ключи словаря, используя **.keys()** напрямую.

С другой стороны, если вы используете **iterkeys()** в своем коде Python 2 и пытаетесь изменить ключи словаря, вы получите **RuntimeError**.

??????? ?? ?????????? ????

До сих пор вы видели более простые способы перебора словаря в Python. Теперь пришло время посмотреть, как вы можете выполнять некоторые действия с элементами словаря во время итерации. Давайте посмотрим на некоторые примеры из реальной жизни.

Примечание. Позже в этой статье вы увидите другой способ решения тех же самых проблем с помощью других инструментов Python.

??????????? ?????? ? ???????? ? ????????

Предположим, у вас есть словарь и по какой-то причине необходимо превратить ключи в значения и наоборот. В этой ситуации вы можете использовать цикл **for** для перебора словаря и создания нового словаря, используя ключи в качестве значений и наоборот:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {}
>>> for key, value in a_dict.items():
... new_dict[value] = key
...
>>> new_dict
{1: 'one', 2: 'two', 3: 'thee', 4: 'four'}
```

Выражение **new_dict[value] = key** сделает всю работу за вас, превратив ключи в значения и используя значения в качестве ключей. Чтобы этот код работал, данные, хранящиеся в исходных значениях, должны иметь тип данных, который можно хэшировать.

??????????

Иногда вы будете в ситуациях, когда у вас есть словарь, и вы захотите создать новый, чтобы хранить только данные, которые удовлетворяют заданному условию. Вы можете сделать это с помощью **if** внутри цикла **for** следующим образом:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {} # Create a new empty dictionary
>>> for key, value in a_dict.items():
... # If value satisfies the condition, then store it in new_dict
... if value <= 2:
... new_dict[key] = value
...
>>> new_dict
{'one': 1, 'two': 2}
```

В этом примере вы отфильтровали элементы со значением больше 2. Теперь **new_dict** содержит только элементы, которые удовлетворяют условному значению <= 2. Это одно из возможных решений для такого рода проблем. Позже вы увидите более понятный и понятный способ получить тот же результат.

????????? ????????? ???????

Также часто требуется выполнять некоторые вычисления, пока вы перебираете словарь в Python. Предположим, вы сохранили данные о продажах вашей компании в словаре, и теперь вы хотите узнать общий доход за год.

Чтобы решить эту проблему, вы можете определить переменную с начальным значением ноль. Затем вы можете накапливать каждое значение вашего словаря в этой переменной:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> total_income = 0.00
>>> for value in incomes.values():
... total_income += value # Accumulate the values in total_income
...
>>> total_income
14100.0
```

Здесь вы просматривали **incomes** и последовательно накапливали их значения в **total_income**, как и хотели. Выражение **total_income += value** делает все работу, и в конце цикла вы получите общий доход за год. Обратите внимание, что **total_income += value** эквивалентно **total_income = total_income + value**.

?????????????? ??????????? (comprehensions)

Генераторы словарей (**Dictionary comprehension**) – это компактный способ обработки всех или части элементов в коллекции и возврата словаря в качестве результата. В отличие от списочных представлений, им нужны два выражения, разделенные двоеточием, за которым следуют предложения **for** и **if** (необязательно). Когда выполняется генератор словаря, полученные пары ключ-значение вставляются в новый словарь в том же порядке, в котором они были созданы.

Предположим, например, что у вас есть два списка данных, и вам нужно создать новый словарь из них. В этом случае вы можете использовать zip (* iterables) в Python для циклического обхода элементов обоих списков:

```
>>> objects = ['blue', 'apple', 'dog']
>>> categories = ['color', 'fruit', 'pet']
>>> a_dict = {key: value for key, value in zip(categories, objects)}
>>> a_dict
{'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
```

Здесь **zip()** получает два итерируемых объекта categories и objects в качестве аргументов и создает итератор, который объединяет элементы из каждого объекта. Объекты кортежа, сгенерированные zip(), затем распаковываются в ключ и значение, которые в итоге используются для создания нового словаря.

Генератор словарей открывает широкий спектр новых возможностей и предоставляет вам отличный инструмент для перебора словаря в Python.

??? ??? ? ??????????? ?????? ? ???????? ? ????????

Если вы по-другому взгляните на проблему превращения ключей в значения и наоборот, вы увидите, что вы могли бы написать более эффективное решение с использованием генератора словарей:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {value: key for key, value in a_dict.items()}
>>> new_dict
{1: 'one', 2: 'two', 3: 'thee', 4: 'four'}
```

С генератором словаря вы создали совершенно новый словарь, в котором ключи заняли место значений, и наоборот. Этот новый подход дал вам возможность писать более читабельный, лаконичный, эффективный и Pythonic код.

Условие для работы этого кода такое же, как вы видели ранее: значения должны быть объектами, которые можно хэшировать. В противном случае вы не сможете использовать их в качестве ключей для **a_dict**.

????????? ??????????

Чтобы отфильтровать элементы в словаре с генератором, вам просто нужно добавить условие **if**, которое определяет условие, которое вы хотите выполнить. В предыдущем примере, когда вы фильтровали словарь, это условие было, если v <= 2. С этим условием **if**, добавленным в конец генератора словаря, вы отфильтруете элементы, значения которых больше 2. Давайте посмотрим:

```
>>> a_dict = {'one': 1, 'two': 2, 'thee': 3, 'four': 4}
>>> new_dict = {k: v for k, v in a_dict.items() if v <= 2}
>>> new_dict
{'one': 1, 'two': 2}
```

Теперь **new_dict** содержит только элементы, которые удовлетворяют вашему условию. По сравнению с предыдущими решениями, это код более Pythonic и эффективный.

????????? ?????????? ????????

Помните пример с продажами компании? Если вы используете генератор списков (**list comprehension**) для перебора значений словаря, вы получите более компактный, быстрый и Pythonic-код:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> total_income = sum([value for value in incomes.values()])
>>> total_income
14100.0
```

Генератор списков создал объект **list**, содержащий значения **incomes**, а затем мы суммировали их все с помощью функции **sum()** и сохранили результат в **total_income**.

Если вы работаете с действительно большим словарем, и использование памяти является проблемой для вас, тогда вы можете использовать **выражение-генератор** (**generator expression**) вместо генератора списков. **выражение-генератор** – это выражение, которое возвращает итератор. Это похоже на генератор списков, но вместо квадратных скобок для его определения необходимо использовать круглые скобки:

```
>>> total_income = sum(value for value in incomes.values())
>>> total_income
14100.0
```

То есть если вы поменяете квадратные скобки для пары круглых скобок (здесь круглые скобки sum()), вы превратите генератор списков в выражение-генератор, и ваш код будет более эффективным в памяти, потому что выражение-генератор использует элементы по запросу. Вместо того, чтобы создавать и хранить весь список в памяти, он будет хранить только один элемент за раз.

Примечание. Если вы новичок в выражение-генератор (generator expressions), вы можете взглянуть на Introduction to Python Generators, чтобы лучше изучить тему.

Наконец, есть более простой способ решить эту проблему, просто используя **incomes.values()** непосредственно в качестве аргумента для **sum()**:

```
>>> total_income = sum(incomes.values())
>>> total_income
14100.0
```

sum() получает в качестве аргумента итерацию и возвращает общую сумму его элементов. Здесь **incomes.values()** играет роль итерируемого значения, переданного в sum(). Результат – общий доход, который вы искали.

???????? ????????? ?????????

Теперь предположим, что у вас есть словарь, и вам нужно создать новый с удаленными выбранными ключами. Помните, как объекты словаря похожи на sets? Эти сходства выходят за рамки просто коллекции хэшируемых и уникальных объектов. Эти объекты также поддерживают общие операции над множествами. Давайте посмотрим, как вы можете

воспользоваться этим, чтобы удалить определенные элементы из словаря:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> non_citric = {k: incomes[k] for k in incomes.keys() - {'orange'}}
>>> non_citric
{'apple': 5600.0, 'banana': 5000.0}
```

Этот код работает, потому что элементы словаря поддерживают операции над множествами, такие как объединения, пересечения и различия. Когда вы написали **incomes.keys () – {‘orange’}**, вы выполняли операцию с заданным различием. Если вам нужно выполнить какие-либо операции над множествами с ключами словаря, то вы можете просто использовать элементы напрямую, без предварительного преобразования его в set.

?????????? ???????

Часто необходимо сортировать элементы коллекции. Начиная с Python 3.6, словари являются упорядоченными структурами данных, поэтому, если вы используете Python 3.6 (и более поздние версии), вы сможете сортировать элементы любого словаря с помощью **sorted()** и с помощью генератора словаря:

```
>>> # Python 3.6, and beyond
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> sorted_income = {k: incomes[k] for k in sorted(incomes)}
>>> sorted_income
{'apple': 5600.0, 'banana': 5000.0, 'orange': 3500.0}
```

Этот код позволяет создать новый словарь с ключами в отсортированном порядке. Это возможно, потому что **sorted(incomes)** возвращает список отсортированных ключей, которые можно использовать для создания нового словаря **sorted_dict**.

???????? ? ??????????????? ???????

Иногда вам может понадобиться перебрать словарь в Python, но вы хотите сделать это в отсортированном порядке. Это может быть достигнуто с помощью **sorted()**. Когда вы вызываете **sorted(iterable)**, вы получаете list с элементами в отсортированном порядке.

Давайте посмотрим, как вы можете использовать **sorted()** для перебора словаря, когда вам нужно сделать это в отсортированном порядке.

?????????? ?? ??????

Если вам нужно перебрать словарь в Python и отсортировать его по ключам, вы можете использовать свой словарь в качестве аргумента для **sorted()**. Это вернет list, содержащий ключи в отсортированном порядке, и вы сможете их перебирать:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> for key in sorted(incomes):
... print(key, '->', incomes[key])
...
apple -> 5600.0
banana -> 5000.0
orange -> 3500.0
```

В этом примере вы отсортировали словарь (в алфавитном порядке) по ключам, используя **sorted(incomes)** в заголовке цикла **for**. Обратите внимание, что вы также можете использовать **sorted(incomes.keys())**, чтобы получить тот же результат. В обоих случаях вы получите список, содержащий ключи вашего словаря в отсортированном порядке.

Примечание. Порядок сортировки будет зависеть от типа данных, который вы используете для ключей или значений, и от внутренних правил, которые Python использует для сортировки этих типов данных.

?????????? ?? ?????????

Вам также может понадобиться перебрать словарь в Python с его элементами, отсортированными по значениям. Вы также можете использовать **sorted()**, но со вторым аргументом **key**.

Аргумент **key** определяет функцию одного аргумента, которая используется для извлечения ключа сравнения для каждого элемента, который вы обрабатываете.

Чтобы отсортировать элементы словаря по значениям, вы можете написать функцию, которая возвращает значение каждого элемента, и использовать эту функцию в **key** аргумента для **sorted()**:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> def by_value(item):
... return item[1]
...
>>> for k, v in sorted(incomes.items(), key=by_value):
... print(k, '->', v)
...
```

```
('orange', '->', 3500.0)
('banana', '->', 5000.0)
('apple', '->', 5600.0)
```

В этом примере мы определили **by_value()** и использовали его для сортировки **incomes** по значению. Затем мы перебираем словарь в порядке сортировки, используя **sorted()**. Ключевая функция (**by_value ()**) сообщает **sorted()** отсортировать **incomes.items()** по второму элементу каждого элемента, то есть по значению (**item [1]**).

Вы также можете просто перебирать значения словаря в отсортированном порядке, не беспокоясь о ключах. В этом случае вы можете использовать **.values()** следующим образом:

```
>>> for value in sorted(incomes.values()):
... print(value)
...
3500.0
5000.0
5600.0
```

sorted(comes.values ()) возвращает значения словаря в отсортированном порядке по вашему желанию. Ключи не будут доступны, если вы используете **incomes.values()**, но иногда вам не нужны ключи, только значения, и это быстрый способ получить к ним доступ.

????????

Если вам нужно отсортировать словари в обратном порядке, вы можете добавить **reverse = True** в качестве аргумента для **sorted()**. Ключевое слово аргумент **reverse** должно принимать логическое значение. Если установлено значение **True**, то элементы сортируются в обратном порядке:

```
>>> incomes = {'apple': 5600.00, 'orange': 3500.00, 'banana': 5000.00}
>>> for key in sorted(incomes, reverse=True):
... print(key, '->', incomes[key])
...
orange -> 3500.0
banana -> 5000.0
apple -> 5600.0
```

Здесь вы перебирали ключи доходов в обратном порядке, используя **sorted(incomes, reverse=True)** в заголовке цикла **for**.

Наконец, важно отметить, что **sorted()** не меняет порядок основного словаря. Что на самом деле происходит, так это то, что **sorted()** создает независимый список со своими элементами в отсортированном порядке, поэтому **incomes** остаются прежними:

>>> incomes
{'apple': 5600.0, 'orange': 3500.0, 'banana': 5000.0}

?????????????? ???????? ? ??????? .popitem()

Иногда нужно перебрать словарь в Python и последовательно удалить его элементы. Для выполнения этой задачи можно использовать **.popitem()**, которая удалит и возвратит произвольную пару ключ-значение из словаря. С другой стороны, когда вы вызываете **.popitem()** в пустом словаре, он вызывает ошибку **KeyError**.

Пример:

```
# File: dict_popitem.py

a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}

while True:
try:
print(f'Dictionary length: {len(a_dict)}')
item = a_dict.popitem()
# Do something with item here...
print(f'{item} removed')
except KeyError:
print('The dictionary has no item now...')
break
```

Внутри цикла **while** мы определили блок **try…except** для того, чтобы поймать **KeyError**, сгенерированный функцией **.popitems()**, когда **a_dict** станет пустым. В блоке **try…except** мы обрабатываем словарь, удаляя элемент в каждой итерации. Переменная **item** хранит ссылку на последующие элементы и позволяет нам выполнять с ними некоторые действия.

Если мы запустим этот скрипт из командной строки, то мы получим следующие результаты:

```
$ python3 dict_popitem.py
Dictionary length: 3
('pet', 'dog') removed
```

```
Dictionary length: 2
('fruit', 'apple') removed
Dictionary length: 1
('color', 'blue') removed
Dictionary length: 0
The dictionary has no item now...
```

Здесь **.popitem()** последовательно удаляет элементы из **a_dict**. Цикл прервался, когда словарь стал пустым, и **.popitem()** вызвал исключение **KeyError**.

?????????????? ?????????? ?? ?????????? ??????? Python

Python предоставляет некоторые встроенные функции, которые могут быть полезны при работе со словарями. Эти функции являются своего рода инструментом итерации, который предоставляет вам еще один способ перебора словаря. Давайте посмотрим на некоторые из них.

map()

map() определяется как **map(function, iterable, ...)** и возвращает итератор, который применяет функцию к каждому элементу итерируемого. Таким образом, **map()** можно рассматривать как инструмент итерации, который можно использовать для перебора словаря.

Предположим, у вас есть словарь, содержащий цены на несколько продуктов, и вам нужно применить скидку к ним. В этом случае вы можете определить функцию, которая управляет скидкой, а затем использует ее в качестве первого аргумента для **map()**. Вторым аргументом может быть **price.items()**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> def discount(current_price):
... return (current_price[0], round(current_price[1] * 0.95, 2))
...
>>> new_prices = dict(map(discount, prices.items()))
>>> new_prices
{'apple': 0.38, 'orange': 0.33, 'banana': 0.24}
```

Здесь **map()** перебирает элементы словаря (**prices.items()**), чтобы применить скидку 5% к каждому фрукту с помощью **discount()**. В этом случае вам нужно использовать **dict()** для создания словаря **new_prices** из итератора, возвращаемого **map()**.

Обратите внимание, что **discount()** возвращает кортеж в форме (**key, value**), где **current_price[0]** представляет ключ, а **round(current_price [1] * 0.95, 2)** представляет новое значение.

filter()

filter () – это еще одна встроенная функция, которую вы можете использовать для перебора словаря и фильтрации некоторых его элементов. Эта функция определяется как **filter(function, iterable)** и возвращает итератор из тех элементов итерируемого, для которых функция возвращает значение **True**.

Предположим, вы хотите знать продукты с ценой ниже 0,40. Вам нужно определить функцию, чтобы определить, удовлетворяет ли цена этому условию, и передать ее в качестве первого аргумента для **filter()**. Вторым аргументом может быть **prices.keys()**:

```
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> def has_low_price(price):
... return prices[price] < 0.4
...
>>> low_price = list(filter(has_low_price, prices.keys()))
>>> low_price
['orange', 'banana']
```

Здесь вы перебирали ключи **prices** с помощью **filter()**. Тогда **filter()** применяет **has_low_price()** к каждому ключу **prices**. Наконец нужно использовать **list()** для генерации списка продуктов с низкой ценой, потому что **filter()** возвращает итератор, и нам нужен объект list.

????????????? collections.ChainMap

`collections` – полезный модуль из стандартной библиотеки Python, предоставляющий специализированные типы данных контейнеров. Одним из таких типов данных является **ChainMap**, который является словарным классом для создания единого представления нескольких сопоставлений (например, словарей). С **ChainMap** вы можете сгруппировать несколько словарей вместе, чтобы создать одно обновляемое представление.

Теперь предположим, что у вас есть два (или более) словаря, и вам нужно перебирать их вместе как один. Для этого вы можете создать объект **ChainMap** и инициализировать его своими словарями:

```
>>> from collections import ChainMap
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35}
```

```
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> chained_dict = ChainMap(fruit_prices, vegetable_prices)
>>> chained_dict # A ChainMap object
ChainMap({'apple': 0.4, 'orange': 0.35}, {'pepper': 0.2, 'onion': 0.55})
>>> for key in chained_dict:
... print(key, '->', chained_dict[key])
...
pepper -> 0.2
orange -> 0.35
onion -> 0.55
apple -> 0.4
```

После импорта **ChainMap** из **collections** вам необходимо создать объект **ChainMap** со словарями, которые вы хотите объединить в цепочку, а затем вы можете свободно перебирать полученный объект, как если бы вы делали это с обычным словарем.

Объекты **ChainMap** также реализуют **.keys()**, **values()** и **.items()**, как это делает стандартный словарь, поэтому вы можете использовать эти методы для итерации по словарному объекту, сгенерированному **ChainMap**, точно так же, как вы это делаете с обычным словарь:

```
>>> for key, value in chained_dict.items():
... print(key, '->', value)
...
apple -> 0.4
pepper -> 0.2
orange -> 0.35
onion -> 0.55
```

В этом случае мы вызвали **.items()** для объекта **ChainMap**. Объект **ChainMap** вел себя так, как будто это был обычный словарь, а **.items()** возвращает объект представления словаря, который можно перебирать как обычно.

????????????? itertools

Iterotools – модуль, который предоставляет некоторые полезные инструменты для выполнения итерационных задач. Давайте посмотрим, как можно использовать некоторые из них для перебора словаря в Python.

??????????? ???????? ? ??????? cycle()

Предположим, вы хотите перебрать словарь в Python, но вам нужно перебирать его несколько раз в одном цикле. Чтобы выполнить эту задачу, вы можете использовать **itertools.cycle(iterable)**, который заставляет итератор возвращать элементы из **iterable** и сохранять копию каждого из них. Когда итерация исчерпана, **cycle()** возвращает элементы из сохраненной копии. Это выполняется циклически, поэтому вы можете остановить цикл.

В следующем примере вы будете перебирать элементы словаря три раза подряд:

```
>>> from itertools import cycle
>>> prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> times = 3 # Define how many times you need to iterate through prices
>>> total_items = times * len(prices)
>>> for item in cycle(prices.items()):
... if not total_items:
... break
... total_items -= 1
... print(item)
...
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
```

Этот код перебирает **prices** определенное количество раз (в данном случае 3). Этот цикл может длиться столько, сколько нужно, но вы должны позаботиться о его остановке. Условие **if** прерывает цикл, когда **total_items** ведет обратный отсчет до нуля.

???????? ? chain()

itertools также предоставляет функцию **chain(*iterables)**, которая получает некоторые итерируемые аргументы в качестве аргументов и создает итератор, который возвращает элементы из итерируемого объекта до тех пор, пока он не будет исчерпан, а затем итерирует по следующему итерируемуго объекту и т. д., пока все они не будут исчерпаны.

Это позволяет вам перебирать несколько словарей в цепочке, как в случае с **collections.ChainMap**:

```
>>> from itertools import chain
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35, 'banana': 0.25}
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55, 'tomato': 0.42}
>>> for item in chain(fruit_prices.items(), vegetable_prices.items()):
... print(item)
...
('apple', 0.4)
('orange', 0.35)
('banana', 0.25)
('pepper', 0.2)
('onion', 0.55)
('tomato', 0.42)
```

В приведенном выше коде **chain()** вернула итерацию, которая объединила элементы из **fruit_prices** и **vegetable_prices**.

Также возможно использовать **.keys()** или **.values()**, в зависимости от ваших потребностей, с условием быть однородным: если вы используете **.keys()** в качестве аргумента для **chain()**, то вам нужно использовать **.keys( )** для остальных из них.

?????????????? ????????? ?????????? ??????? (**)

Python 3.5 приносит новую и интересную функцию. PEP 448 – Additional Unpacking Generalizations могут упростить вашу жизнь, когда дело доходит до перебора нескольких словарей в Python. Давайте посмотрим, как это работает, на коротком примере.

Предположим, у вас есть два (или более) словаря, и вам нужно выполнять их итерацию вместе, без использования **collection.ChainMap** или **itertools.chain()**. В этом случае можно использовать оператор распаковки словаря **(**)**, чтобы объединить два словаря в новый и затем выполнить итерацию по нему:

```
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35}
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> # How to use the unpacking operator **
>>> {**vegetable_prices, **fruit_prices}
{'pepper': 0.2, 'onion': 0.55, 'apple': 0.4, 'orange': 0.35}
```

```
>>> # You can use this feature to iterate through multiple dictionaries
>>> for k, v in {**vegetable_prices, **fruit_prices}.items():
... print(k, '->', v)
...
pepper -> 0.2
onion -> 0.55
apple -> 0.4
orange -> 0.35
```

Оператор распаковки словаря **(**)** действительно замечательная функция в Python. Она позволяет объединить несколько словарей в один новый, как мы это делали в примере с **vegetable_prices** и **fruit_prices**. После объединения словарей с оператором распаковки вы можете перебирать новый словарь как обычно.

Важно отметить, что если словари, которые вы пытаетесь объединить, имеют повторяющиеся или общие ключи, то значения самого послденего словаря будут преобладать:

```
>>> vegetable_prices = {'pepper': 0.20, 'onion': 0.55}
>>> fruit_prices = {'apple': 0.40, 'orange': 0.35, 'pepper': .25}
>>> {**vegetable_prices, **fruit_prices}
{'pepper': 0.25, 'onion': 0.55, 'apple': 0.4, 'orange': 0.35}
```

Ключ **pepper** присутствует в обоих словарях. После объединения их значение **fruit_prices** для **pepper** (0.25) превалирует, потому что **fruit_prices** – самый последний словарь.

??????????

В этой статье мы основы того, как перебирать словарь в Python, а также некоторые более продвинутые методы и стратегии!

Вы узнали:

- Что такое словари, а также некоторые их основные функции и детали реализации
- Каковы основные способы перебора словаря в Python:
- Какие задачи вы можете выполнить, итерируя словарь
- Как использовать некоторые более сложные методы и стратегии для перебора словаря

Revision #1
Created 2024-08-26 13:58:00 UTC by Антон Сергеевич Абраменко
Updated 2024-08-26 14:13:43 UTC by Антон Сергеевич Абраменко