IT CAREER HUB

Урок 21.

Модуль collections

IT CAREER HUB

Модуль time

Модуль `time` предоставляет функции для работы с временем, включая получение текущего времени, измерение интервалов и управление задержками выполнения кода.

Основные функции модуля `time`

1. `time()`

Возвращает текущее время в секундах, прошедшее с 1 января 1970 года.

```python
import time

current_time = time.time()
print(current_time)
```

2. `sleep(seconds)`

Приостанавливает выполнение программы на указанное количество секунд.

```python
import time

time.sleep(2)  # Задержка на 2 секунды
print("2 секунды спустя...")
```

Пример: измерение времени выполнения программы

```python
import time

start_time = time.time()
```

```
# Создание объекта range
range_million = range(1000000)
end_time = time.time()
print(f"Время создания range: {end_time - start_time:.10f} секунд")

start_time = time.time()
# Создание объекта списка
lst = [x for x in range(1000000)]
end_time = time.time()

print(f"Время создания list: {end_time - start_time:.10f} секунд")
```

IT CAREER HUB

Модуль collections

Модуль `collections` предоставляет дополнительные структуры данных, которые дополняют стандартные типы Python. Эти структуры данных оптимизированы для выполнения различных задач и могут быть более эффективными по сравнению с обычными списками и словарями.

Класс OrderedDict

`OrderedDict` — это класс из модуля `collections`, который представляет собой словарь, сохраняющий порядок добавления элементов.

В отличие от стандартных словарей в старых версиях Python (до 3.7), где порядок ключей не гарантировался, `OrderedDict` всегда сохраняет порядок добавления.

Синтаксис

```python
from collections import OrderedDict

ordered_dict = OrderedDict(iterable)
```

`iterable` — любой итерируемый объект, который предоставляет пары ключ: значение (например, список кортежей, словарь, генераторы и т.д.). Если объект не указан, создаётся пустой `OrderedDict`.

Пример создания `OrderedDict`:

```python
# Импорт класса из модуля collections
from collections import OrderedDict

# Создание пустого OrderedDict
od = OrderedDict()
```

```
# Добавление элементов аналогично работе со словарем
od["a"] = 1
od["b"] = 2
od["c"] = 3

print(od)
```

Реализация `OrderedDict` на основе словаря

Класс `OrderedDict` реализован на основе встроенного словаря Python. Это означает, что он использует все преимущества стандартного словаря, такие как быстрое хешированное обращение к элементам, но добавляет дополнительную функциональность.

IT CAREER HUB

Метод popitem()

Метод `popitem()` в `OrderedDict`, как и в `dict`, позволяет удалять и возвращать пары "ключ-значение" из словаря.

В отличие от стандартного словаря, `OrderedDict` позволяет указать, с какой стороны забрать значения.

Синтаксис

```python
key, value = ordered_dict.popitem(last=True)
```

- **`last=True`** (по умолчанию) — удаляет последний добавленный элемент.
- **`last=False`** — удаляет первый добавленный элемент.

Примеры использования

1. **Удаление элементов**

```python
from collections import OrderedDict

od = OrderedDict()
od["a"] = 1
od["b"] = 2
od["c"] = 3
od["d"] = 4

# Удаление последнего элемента
print(od.popitem())
print(od)

# Удаление первого элемента
print(od.popitem(last=False))
print(od)
```

2. Реализация очереди

```python
from collections import OrderedDict

queue = OrderedDict()
queue["first"] = 1
queue["second"] = 2
queue["third"] = 3

# Удаляем элементы с начала очереди
while queue:
    print(queue.popitem(last=False))
```

Метод `move_to_end`

Метод `move_to_end()` уникален для `OrderedDict` и не доступен в стандартных словарях Python. Его функциональность позволяет легко перемещать элементы в начало или конец словаря, сохраняя при этом порядок остальных элементов.

Это делает `OrderedDict` полезным инструментом для задач, где важно гибкое управление порядком элементов, таких как реализация очередей с приоритетами или перераспределение задач.

Синтаксис

```python
ordered_dict.move_to_end(key, last=True)
```

- `key` — ключ элемента, который нужно переместить.
- `last=True` (по умолчанию) указывает, что элемент будет перемещён в конец.
- `last=False` указывает, что элемент будет перемещён в начало.

Пример применения для управления очередью

Python

```python
from collections import OrderedDict

queue = OrderedDict()
queue["task1"] = "low priority"
queue["task2"] = "medium priority"
queue["task3"] = "low priority"
queue["task4"] = "high priority"
queue["task5"] = "medium priority"

# Собираем ключи для перемещения
keys_to_end = [key for key, value in queue.items() if "low" in value]
keys_to_start = [key for key, value in queue.items() if "high" in value]

# Перемещаем задачи с низким приоритетом в конец
for key in keys_to_end:
    queue.move_to_end(key)
# Перемещаем задачи с высоким приоритетом в начало
for key in keys_to_start:
    queue.move_to_end(key, last=False)

print(queue)
```

IT CAREER HUB

Кэш

Кэш — это механизм, позволяющий хранить результаты вычислений или часто запрашиваемые данные, чтобы ускорить доступ к ним в будущем. В программировании кэш используется для оптимизации производительности, уменьшая количество повторных вычислений или обращений к медленным ресурсам, таким как база данных или файловая система.

Зачем нужен кэш?

- Ускорение вычислений: Если результат операции уже известен, его можно взять из кэша вместо повторного вычисления.
- Оптимизация работы с ресурсами: Кэш снижает нагрузку на внешние системы, например, базы данных, API или файловую систему.
- Снижение времени отклика: Часто используемые данные, такие как настройки или результаты сложных вычислений, можно хранить в памяти для быстрого доступа.

Примеры кэша

- **Кэширование функций:** Сохранение результатов выполнения функции для повторного использования при тех же входных данных.
- **Кэширование данных:** Хранение часто запрашиваемых данных в памяти или на диске.
- **Кэш браузера:** Сохранение страниц, изображений и других ресурсов для ускорения загрузки веб-сайтов.

Применение кэша

- Веб-приложения:
    - Сохранение результатов запросов к базе данных.
    - Хранение сессий пользователей.
- Обработка данных:
    - Кэширование промежуточных результатов в сложных расчётах.
- Машинное обучение:
    - Кэширование предварительно обработанных данных или моделей.
- Компьютерные игры:
    - Хранение игровых текстур и настроек.

IT CAREER HUB

Понятие LRU-кэша

LRU-кэш (Least Recently Used) — это структура данных или алгоритм, используемый для кэширования данных таким образом, чтобы сохранять только наиболее недавно использовавшиеся элементы. Когда кэш достигает максимального размера и необходимо добавить новый элемент, LRU-кэш удаляет элемент, который использовался дольше всех.

Основные принципы работы LRU-кэша:

- Кэширование данных: LRU-кэш хранит фиксированное количество элементов (например, результаты вычислений или данные, которые часто запрашиваются), что ускоряет доступ к данным, не требуя повторных вычислений или долгих операций извлечения.
- Удаление старых данных: Когда кэш достигает максимального размера, он удаляет элемент, который использовался дольше всех, чтобы освободить место для нового элемента.
- Упорядочение данных: LRU-кэш может быть реализован с помощью `OrderedDict`, чтобы поддерживать порядок использования элементов.

Пример применения

Представьте, что у вас есть веб-приложение, которое кэширует результаты запросов к базе данных. LRU-кэш будет хранить наиболее часто используемые запросы и удалять старые результаты, которые больше не востребованы. Это уменьшает время доступа к данным и повышает производительность.

Реализация LRU-кэша в Python

Python предоставляет встроенный механизм для работы с LRU-кэшем в виде декоратора `@lru_cache` из модуля `functools`. `@lru_cache` автоматически сохраняет результаты вызовов функции с одинаковыми аргументами.

- maxsize — количество сохранённых результатов. Если кэш переполняется, старые записи удаляются.
- Декоратор — это функция в Python, которая изменяет или расширяет функциональность другой функции или метода без изменения их кода.

Синтаксис декоратора `@lru_cache`

```python
from functools import lru_cache


@lru_cache(maxsize=128, typed=False)

def function_name(arguments):

    # тело функции
```

Параметры:

- maxsize: Максимальное количество сохраняемых записей. Если кэш переполняется, старые записи удаляются согласно принципу LRU. Значение None позволяет кэшу быть неограниченным. По умолчанию: 128.
- typed: Если `True`, кэширование различает аргументы по их типу. Например, 1 и `1.0` будут считаться разными ключами. По умолчанию: `False`.

Пример использования

```python
from time import time, sleep
from functools import lru_cache

# Функция с временной задержкой для имитации сложных вычислений
@lru_cache(maxsize=2)
def compute_square(n):
    print(f"Вычисляю квадрат числа {n}...")
    sleep(2)  # Имитация долгой операции
    return n * n

# Измерение времени выполнения
start_time = time()
print(f"Результат: {compute_square(2)}")  # Вычисляет
print(f"Время: {time() - start_time:.2f} секунд\n")
```

```
start_time = time()
print(f"Результат: {compute_square(3)}")  # Вычисляет
print(f"Время: {time() - start_time:.2f} секунд\n")

start_time = time()
print(f"Результат: {compute_square(2)}")  # Использует кэш
print(f"Время: {time() - start_time:.2f} секунд\n")
```

Объяснение:

- `@lru_cache(maxsize=2)` создаёт LRU-кэш, который хранит до **2 элементов**.
- Если добавляется новый элемент, а кэш уже заполнен, **самый старый (по использованию) элемент удаляется**.

Преимущества LRU-кэша:

1. Ускорение доступа: Повторные вызовы функции с теми же параметрами возвращают результат быстрее, так как используются закэшированные данные.
2. Контроль размера: Позволяет ограничить количество элементов в кэше, предотвращая избыточное потребление памяти.
3. Простота в использовании: Встроенный декоратор `lru_cache` упрощает реализацию кэша в Python.

LRU-кэш — полезный инструмент для повышения производительности приложений, в которых важно кэшировать данные с учётом их частоты использования.

IT CAREER HUB

Задания для закрепления 1

1. Какой результат будет выведен при выполнении следующего кода?

```python
from collections import OrderedDict

fruits = OrderedDict()
fruits["apple"] = 3
fruits["banana"] = 5
fruits["orange"] = 2

fruits.move_to_end("apple")
fruits.move_to_end("orange", last=False)
print(list(fruits.keys()))
```

a. ['orange', 'banana', 'apple']
b. ['banana', 'apple', 'orange']
c. ['orange', 'apple', 'banana']
d. ['apple', 'banana', 'orange']

Посмотреть ответ

IT CAREER HUB

Класс defaultdict

`defaultdict` — это класс из модуля `collections`, который расширяет возможности стандартного словаря Python. Его ключевая особенность — возможность задавать значение **по умолчанию** для отсутствующих ключей, что упрощает работу с данными и уменьшает вероятность возникновения ошибок `KeyError`.

Синтаксис:

```python
from collections import defaultdict

defaultdict(default_type)
```

- **default_type** — это функция или класс, который автоматически создаёт значение **по умолчанию** для отсутствующего ключа.

Особенности и ограничения

Сохранение значения по умолчанию:

После обращения к отсутствующему ключу он **автоматически добавляется** в словарь со значением по умолчанию.

```python
from collections import defaultdict

dd = defaultdict(int)
print(dd['missing'])  # Добавлено значение 0
print(dd)
```

Совместимость:

- `defaultdict` полностью совместим с методами стандартного словаря Python.

Отсутствие default_type:

- Если при создании **не указан** `default_type`, `defaultdict` ведёт себя как обычный словарь и вызывает `KeyError` при доступе к отсутствующему ключу.

Пример использования

1. Создание словаря со значением типа `int` по умолчанию:

```python
from collections import defaultdict

# Словарь с числовым значением по умолчанию
dd = defaultdict(int)
# Присваивает ключу базовое значение int
print(dd['a'])
# Обновляет имеющийся ключ
dd['a'] += 1
print(dd['a'])
# Присваивает ключу базовое значение int и обновляет его
dd['b'] += 10
print(dd['b'])
print(dd)
```

Здесь `int` используется как `default_type`, который возвращает `0` для отсутствующих ключей.

2. Создание словаря со списком по умолчанию:

```python
from collections import defaultdict

# Словарь с пустым списком по умолчанию
dd = defaultdict(list)
# Присваивает ключу базовое значение list и обновляет его
dd['a'].append(1)
# Обновляет список по имеющемуся ключу
dd['a'].append(2)
# Присваивает ключу базовое значение list и обновляет его
dd['b'].append(10)
print(dd)
```

Это полезно, когда нужно обработать данные и сгруппировать их по определенному критерию.

3. Использование кастомной функции:

```python
from collections import defaultdict

# Пользовательская функция для значения по умолчанию
def default_value():
    return "default"

dd = defaultdict(default_value)
print(dd['missing_key'])
print(dd)
```

Преимущества defaultdict

- Упрощение кода

Нет необходимости проверять существование ключа перед изменением значения.

```python
# Обычный словарь
my_dict = {}
if 'a' not in my_dict:
    my_dict['a'] = []
my_dict['a'].append(1)

# defaultdict
from collections import defaultdict
dd = defaultdict(list)
dd['a'].append(1)
```

- Уменьшение количества ошибок

Исключает вероятность возникновения `KeyError` при доступе к отсутствующему ключу.

- Гибкость

Можно задать любое значение по умолчанию, используя функцию или класс.

IT CAREER HUB

Задания для закрепления 2

1. Какой результат будет выведен при выполнении следующего кода?

```python
from collections import defaultdict

dd = defaultdict(list)
dd['x'].append(1)
dd['y'].extend([2, 3])
print(dd['z'])
```

a. `[]`
b. `[1]`
c. `[2, 3]`
d. Ошибка

Посмотреть ответ

2. Какой результат будет выведен при выполнении следующего кода?

```python
from collections import defaultdict

words = ["apple", "banana", "apple", "orange", "banana"]
word_count = defaultdict(int)

for word in words:
    word_count[word] += 1

print(word_count["apple"])
```

a. `0`
b. 1
c. 2
d. 3

Посмотреть ответ

3. Какой результат будет выведен при выполнении следующего кода?

```python
from collections import defaultdict

data = [("class1", "Alice"), ("class2", "Bob"), ("class1", "Charlie")]
grouped = defaultdict(list)

for group, name in data:
    grouped[group].append(name)

print(grouped["class1"])
```

a. `["Alice"]`
b. `["Charlie"]`
c. `["Alice", "Charlie"]`
d. `["class1", "Alice", "Charlie"]`

Посмотреть ответ

4. Чем отличается `defaultdict` от обычного словаря?

a. `defaultdict` сохраняет порядок ключей
b. `defaultdict` автоматически добавляет значение по умолчанию для отсутствующего ключа
c. `defaultdict` быстрее обычного словаря
d. Никаких отличий нет

Посмотреть ответ

IT CAREER HUB

Класс Counter

`Counter` — это класс из модуля collections, предназначенный для подсчёта количества элементов в итерируемом объекте. Он автоматически создаёт словарь, где элементы — это ключи, а их количество — значения.

`Counter` — полезный инструмент для подсчёта элементов в данных, который упрощает обработку задач, связанных с частотным анализом.

Синтаксис:

```python
from collections import Counter

Counter(iterable)
Counter(mapping)
Counter(**kwargs)
```

- `iterable` — итерируемый объект (строка, список и т.д.), элементы которого нужно подсчитать.
- `mapping` — словарь, где ключи — элементы, а значения — их количество.
- `kwargs` — произвольные пары ключ-значение.

Основные особенности:

- **Автоматический подсчёт элементов:** Класс `Counter` позволяет легко подсчитать количество одинаковых элементов в коллекции без явного написания циклов.
- **Поддержка стандартных операций со словарями:** Работает как словарь, предоставляя доступ к методам, значениям и ключам.
- **Поддержка арифметических операций:** Позволяет выполнять сложение, вычитание и другие операции между объектами `Counter`.

Примеры использования

1. Подсчёт символов в строке:

```python
from collections import Counter
```

```
text = "hello world"
counter = Counter(text)
print(counter)
```

2. Подсчёт слов в списке:

```python
from collections import Counter

words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
counter = Counter(words)
print(counter)
```

3. Создание `Counter` из словаря:

```python
from collections import Counter

data = {"apple": 3, "banana": 2, "cherry": 1}
counter = Counter(data)
print(counter)
```

IT CAREER HUB

Методы класса Counter

`most_common([n])`

- Возвращает список из n наиболее часто встречающихся элементов, отсортированных по убыванию.
- Если `n` не указано, возвращаются все элементы.

```python
counter = Counter("banana")
print(counter.most_common(2))
```

`elements()`

- Возвращает итератор, который повторяет элементы столько раз, сколько они встречаются.
- Если элемент имеет отрицательное или нулевое количество, он игнорируется.

```python
counter = Counter({"a": 3, "b": 1, "c": 0})
iter_count = counter.elements()
print(list(iter_count))
```

`subtract([iterable-or-mapping])`

- Вычитает элементы, уменьшая их количество. Может принимать как итерируемый объект, так и словарь.
- Значения могут стать отрицательными.

```python
counter = Counter("banana")
counter.subtract("an")
print(counter)
```

`update([iterable-or-mapping])`

- Увеличивает количество элементов из переданного объекта.

IT CAREER HUB

```python
counter = Counter("banana")
counter.update("nan")
print(counter)
```

IT CAREER HUB

Операции между объектами Counter

Сложение:

Объединяет два `Counter`, складывая количества одинаковых элементов.

```python
c1 = Counter("banana")
c2 = Counter("apple")
print(c1 + c2)
```

Вычитание:

Вычитает количества, игнорируя отрицательные результаты.

```python
c1 = Counter("banana")
c2 = Counter("an")
print(c1 - c2)
```

Пересечение:

Оставляет минимальные количества одинаковых элементов.

```python
c1 = Counter("banana")
c2 = Counter("an")
print(c1 & c2)
```

Объединение:

Оставляет максимальные количества одинаковых элементов.

```python
c1 = Counter("banana")
c2 = Counter("an")
print(c1 | c2)
```

IT CAREER HUB

Задания для закрепления 3

1. **Какой метод возвращает наиболее часто встречающиеся элементы?**

a. `elements()`
b. `most_popular()`
c. `popular()`
d. `most_common()`

Посмотреть ответ

IT CAREER HUB

Ответы на задания

Задания на закрепление 1	Вернуться к заданиям
1. Результат выполнения кода	Ответ: a
Задания на закрепление 2	Вернуться к заданиям
1. Результат выполнения кода	Ответ: a
2. Результат выполнения кода	Ответ: c
3. Результат выполнения кода	Ответ: c
4. Отличия `defaultdict` от обычного словаря	Ответ: b
Задания на закрепление 3	Вернуться к заданиям
1. Метод, возвращающий наиболее часто встречающиеся элементы	Ответ: d

IT CAREER HUB

🔎 Практическая работа

1. Частотный анализ слов

Напишите программу, которая подсчитывает количество вхождений каждого слова в тексте.
Программа должна игнорировать регистр слов и символы `.` и `,`.

Данные:

```python
text = "This is a test. This test is only a test."
```

Пример вывода:

```
{'this': 2, 'is': 2, 'a': 2, 'test': 3, 'only': 1}
```

Решение:

```python
from collections import Counter

text = "This is a test. This test is only a test."
# Приводим текст к нижнему регистру и разделяем на слова
words = text.lower().replace(".", "").replace(",", "").split()
# Подсчитываем частоту слов
word_count = Counter(words)

print(dict(word_count))
```

2. Список студентов по факультетам

Напишите программу, которая принимает список студентов и их факультетов (кортежи) и группирует студентов по факультетам в словарь.

Данные:

```python
students = [
    ("Иван", "Физика"),
    ("Мария", "Математика"),
    ("Пётр", "Физика"),
    ("Анна", "Математика"),
    ("Олег", "Информатика"),
    ("Наталья", "Физика"),
]
```

Пример вывода:

```
Факультеты и студенты:
Физика: ['Иван', 'Пётр', 'Наталья']
Математика: ['Мария', 'Анна']
Информатика: ['Олег']
```

Решение:

```python
from collections import defaultdict

def group_students_by_faculty(students):
    faculty_dict = defaultdict(list)
    for name, faculty in students:
        faculty_dict[faculty].append(name)
    return dict(faculty_dict)

students = [
    ("Иван", "Физика"),
    ("Мария", "Математика"),
    ("Пётр", "Физика"),
    ("Анна", "Математика"),
    ("Олег", "Информатика"),
    ("Наталья", "Физика"),
]

result = group_students_by_faculty(students)
print("Факультеты и студенты:")
for faculty, names in result.items():
    print(f"{faculty}: {names}")
```