

Лекция 14

Итераторы

Итераторы

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter (iterable):` Возвращает итератор над элементами итерируемого значения.

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter (iterable):` Возвращает итератор над элементами итерируемого значения.

`next (iterator):` Возвращает следующий элемент итератора

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

```
>>> s = [3, 4, 5]
```

`next` (iterator): Возвращает следующий элемент итератора

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

>>> s = [3, 4, 5]


Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
```



Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
```

Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```


Итераторы

Коллекции данных могут предоставлять *итератор* для последовательного доступа к данным.

`iter` (iterable): Возвращает итератор над элементами итерируемого значения.

`next` (iterator): Возвращает следующий элемент итератора

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```

(Пример)

Итерация над словарями

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Словари

Значение называют *итерируемым* если оно может быть отправлено в **iter** для получения итератора.

Итератор возвращается из **iter** и может быть направлен в **next**; все итераторы изменчивы.

Словари

Значение называют *итерируемым* если оно может быть отправлено в **iter** для получения итератора.

Итератор возвращается из **iter** и может быть направлен в **next**; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

- Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)
- Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
```

```
>>> d['zero'] = 0
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)      >>> v = iter(d.values())
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```


Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'

>>> v = iter(d.values())
>>> next(v)
1
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'

>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'

>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'

>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0

>>> i = iter(d.items())
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
```

```
>>> d['zero'] = 0
```

```
>>> k = iter(d.keys()) # или iter(d)
```

```
>>> next(k)
```

```
'one'
```

```
>>> next(k)
```

```
'two'
```

```
>>> next(k)
```

```
'three'
```

```
>>> next(k)
```

```
'zero'
```

```
>>> v = iter(d.values())
```

```
>>> next(v)
```

```
1
```

```
>>> next(v)
```

```
2
```

```
>>> next(v)
```

```
3
```

```
>>> next(v)
```

```
0
```

```
>>> i = iter(d.items())
```

```
>>> next(i)
```

```
('one', 1)
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
```


Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
```


Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```

Словари

Значение называют *итерируемым* если оно может быть отправлено в `iter` для получения итератора.

Итератор возвращается из `iter` и может быть направлен в `next`; все итераторы изменчивы.

Словарь, его ключи, значения и элементы – все они итерируемы.

– Порядок элементов в словаре соответствует порядку их добавления (Python 3.6+)

– Ранее элементы были неупорядоченны (Python 3.5 и более ранние)

(Пример)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys()) # или iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```

Инструкция for

Инструкция for

(Пример)

Встроенные функции для работы с итераторами

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable)`: Проходит по `func(x)` для `x` из `iterable`

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Чтобы увидеть результат, помести результаты в последовательность

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Чтобы увидеть результат, помести результаты в последовательность

`list(iterable):` Создаёт список, содержащий все `x` из `iterable`

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Чтобы увидеть результат, помести результаты в последовательность

`list(iterable):` Создаёт список, содержащий все `x` из `iterable`

`tuple(iterable):` Создаёт тапл, содержащий все `x` из `iterable`

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Чтобы увидеть результат, помести результаты в последовательность

`list(iterable):` Создаёт список, содержащий все `x` из `iterable`

`tuple(iterable):` Создаёт тапл, содержащий все `x` из `iterable`

`sorted(iterable):` Создаёт отсортированный список всех `x` из `iterable`

Встроенные функции для итераторов

Многие встроенные операции над последовательностями возвращают итераторы и вычисляют результаты лениво.

`map(func, iterable):` Проходит по `func(x)` для `x` из `iterable`

`filter(func, iterable):` Проходит по `x` из `iterable` если `func(x) == True`

`zip(first_iter, second_iter):` Проходит по парам `(x, y)` с одинаковым индексом

`reversed(sequence):` Проходит по `x` из `sequence` в обратном порядке

Чтобы увидеть результат, помести результаты в последовательность

`list(iterable):` Создаёт список, содержащий все `x` из `iterable`

`tuple(iterable):` Создаёт тапл, содержащий все `x` из `iterable`

(Пример) `sorted(iterable):` Создаёт сортированный список всех `x` из `iterable`

Генераторы

Генераторы и функции-генераторы

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x
```

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат одновременно, функция-генератор выдаёт результат частями.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат одновременно, функция-генератор выдаёт результат частями.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат единовременно, функция-генератор выдаёт результат частями.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат единовременно, функция-генератор выдаёт результат частями.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3  
>>> t  
<generator object plus_minus ...>
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат единовременно, функция-генератор выдаёт результат частями.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3  
>>> t  
<generator object plus_minus ...>
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат одновременно, функция-генератор выдаёт результат частями.

Генератор – это автоматически создаваемый итератор при вызове функции-генератора.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):  
...     yield x  
...     yield -x  
  
>>> t = plus_minus(3)  
>>> next(t)  
3  
>>> next(t)  
-3  
>>> t  
<generator object plus_minus ...>
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат одновременно, функция-генератор выдаёт результат частями.

Генератор – это автоматически создаваемый итератор при вызове функции-генератора.

Из функции-генератора возвращается генератор, который итерирует по выдаваемым результатам.

Генераторы и функции-генераторы

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

Функция-генератор выдаёт (`yield`), а не возвращает (`return`) результат.

Обычная функция возвращает весь результат одновременно, функция-генератор выдаёт результат частями.

Генератор – это автоматически создаваемый итератор при вызове функции-генератора.

Из функции-генератора возвращается генератор, который итерирует по выдаваемым результатам.

(Пример)

Генераторы и итераторы

Генераторы могут выдавать результаты из итераторов

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x
```


Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:  
        yield k
```

Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:  
        yield k  
        yield from countdown(k-1)
```


Генераторы могут выдавать результаты из итераторов

Инструкция `yield from` выдаёт все значения из итератора или итерируемой величины (Python 3.3)

```
>>> list(a_then_b( [3, 4], [5, 6] ))  
[3, 4, 5, 6]
```

```
def a_then_b(a, b):  
    for x in a:  
        yield x  
    for x in b:  
        yield x
```

```
def a_then_b(a, b):  
    yield from a  
    yield from b
```

```
>>> list(countdown(5))  
[5, 4, 3, 2, 1]
```

```
def countdown(k):  
    if k > 0:  
        yield k  
        yield from countdown(k-1)
```

(Пример)