

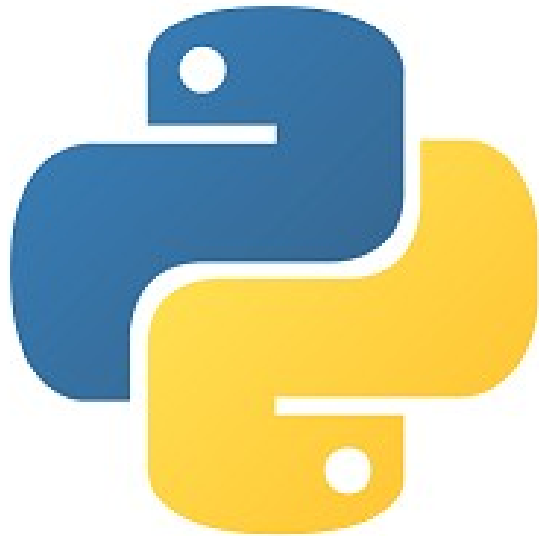
# Итераторы, генераторы и менеджеры контента



# Итератор

- Итератор (iterator) – это объект, который используется для прохода по итерируемому элементу.
- В основном используется для коллекция(списки, словари и т.д)
- Протокол Iterator в Python включает две функции. Один – `iter()`, другой – `next()`. И два дандера **`__iter__`** **`__next__`**

## ***Iterator in Python***



**`__iter__`** (It convert Iterable to Iterator Object)

**`__next__`** (It return the next element from the collection)

# Дандер `__iter__`

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]
```

```
for i in num_list:
```

```
    print(i)
```

```
dir(num_list)
```

```
['__add__', '__class__', '__contains__',  
'__delattr__', '__delitem__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattr__',  
'__getitem__', '__gt__', '__hash__', '__iadd__',  
'__imul__', '__init__', '__init_subclass__',  
'__iter__', ...]
```

# Дандер `__iter__`

Рассмотрим пример:

```
num_list = [1, 2, 3, 4, 5]
```

1. Конструкция `for` в момент выполнения берет у коллекции объект `__iter__`

```
for i in num_list:
```

```
    print(i)
```

```
print(num_list.__iter__)
```

```
<method-wrapper '__iter__' of list object at 0x7f602f9b0bc0>
```

2. Получим объект итерации через функцию `iter()`

```
it = num_list.__iter__()
```

3. Посмотрим тип объекта

```
print(it) → <list_iterator object at 0x7f602f730d90>
```

```
dir(it)
```

```
['__class__', '__delattr__', '__dir__', '__doc__',  
'__init__', '__init_subclass__', '__iter__',  
'__new__', '__next__', '..']
```



# Дом, который построил Джек

*Вот дом,  
Который построил Джек.*

*А это пшеница,  
Которая в тёмном чулане хранится  
В доме,  
Который построил Джек.*

*А это весёлая птица-синица,  
Которая часто ворует пшеницу,  
Которая в тёмном чулане хранится  
В доме,  
Который построил Джек.*

*..*

*--Самуил Маршак*

# Вывод.

- Как мы могли убедиться, цикл **for** использует так называемые итераторы.
- Коллекция содержит метод-wrapper **\_\_iter\_\_** .  
Который в свою очередь возвращает объект итератор. В котором реализован метод **\_\_next\_\_**. Который в свою очередь возвращает последующий итерируемый элемент
- Цикла **for** можно разложить на след. операции:

```
num_list = [1, 2, 3, 4, 5]
itr = iter(num_list)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

# Создание собственных итераторов

Разобравшись как это работает мы можем написать собственную реализацию итератора. Достаточно реализовать дандер **\_\_next\_\_**

```
class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1)) → Что произойдет ?
```





# Итератор для цикла


Для итерации в цикле нам необходимо реализовать дандер **`__iter__`** который возвращает `self`

```
class SimpleIterator:
    def __iter__(self):
        return self
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0
    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration
s_iter2 = SimpleIterator(5)
for i in s_iter2:
    print(i)
```



# Генераторы – это упрощенные итераторы

Чтобы облегчить написание итераторов используются генераторы и ключевое слово `yield`



# Меняем бесконечный генератор на итератор

Перепишем итератор

```
class Repeater:
    def __init__(self, value):
        self.value = value
    def __iter__(self):
        return self
    def __next__(self):
        return self.value
```

```
def repeater(value):
    while True
        yield value
```

```
for x in repeater(0):
    Print(x)
```

```
#0
```

```
#0
```

```
#0
```



## Как это работает ?

Генераторы похожи на нормальные функции , но их поведение различаются. Вызов функции-генератора вообще не выполняет функцию а просто создает и возвращает объект-генератор

```
def repeater(value):  
    while True  
        yield value
```

```
generator_obj = repeater("Hello")
```

Программный код выполняется тогда когда функция next вызывается с объектом генератора

```
next(generator_obj)
```




## Как прекратить генерацию?

Генераторы прекращают порождать значения, как только поток управления возвращается из функции-генератора каким-либо иным способом, кроме инструкции `yield`

```
bounder_repeater(value, max_repeats):  
    count = 0  
    while True:  
        If count >= max_repeats:  
            return  
        Count += 1  
        yield value
```

```
for x in bounder_repeater("Раз",  
    print()  
"Раз"  
"Раз"
```



# Что заставляет генератор прекращать работу ?

```
def repeater_tree_value():  
    yield 1  
    yield 2  
    yield 3
```

```
for i in repeater_tree_value():  
    print(i)
```


```
1  
2  
3
```

```
it = repeater_tree_value()  
print(next(it))  
print(next(it))  
print(next(it))  
print(next(it)) → StopIteration
```



# Выводы

- Функции-генераторы являются синтаксическим сахаром для написания объектов, которые поддерживают протокол итератора. Генератор абстрагирует от всей кухни шаблонного кода.
- Инструкция **yield** позволяет временно приостановить исполнение функции-генератора и передавать из него значения назад.
- Генераторы начинают вызывать исключения **StopIteration** после того, как поток управления покидает функцию-генератор каким-либо иным способом, кроме инструкции `yield`



# Контекстные менеджеры и инструкция **with**

```
with open(path, 'w') as f_obj:  
    f_obj.write(some_data)
```

В данном случае конструкция **with** гарантирует автоматическое закрытие открытых дескрипторов файла после того, как выполнение программы покидает контекст инструкции **with**. На внутреннем уровне данный пример кода выглядит следующим образом.

```
f = open('hello', 'w')  
  
try:  
    f.write('hello, my sun!')  
  
finally:  
    f.close()
```

При возникновении исключения будет его обработка.





# Что такое менеджер контекста ?

Менеджер контекста – это интерфейс который должен соблюдать объект для того, чтобы поддерживать инструкцию **with**. Чтобы объект функционировал как менеджер контекста нужно добавить в него методы **\_\_enter\_\_** и **\_\_exit\_\_**

**\_\_enter\_\_** – Python вызывает метод когда входит в контекст инструкции **with** и наступает момент получения ресурса.

**\_\_exit\_\_** – метод вызывается для высвобождения ресурса



# Пример

```
class WritableFile:
    def __init__(self, file_path):
        self.file_path = file_path

    def __enter__(self):
        self.file_obj = open(self.file_path, mode="w")
        return self.file_obj

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file_obj:
            self.file_obj.close()

with WritableFile("hello.txt") as file:
    file.write("Hello, World!")
```



```
import sqlite3
class DataConn:

    def __init__(self, db_name):
        """Конструктор"""
        self.db_name = db_name

    def __enter__(self):
        """
        Открываем подключение с базой данных.
        """
        self.conn = sqlite3.connect(self.db_name)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        Закрываем подключение.
        """
        self.conn.close()
        if exc_val:
            raise

with DataConn(db) as conn:
    cursor = conn.cursor()
```



## Выводы

- Инструкция **with** упрощает обработку исключений путем инкапсуляции стандартных случаев применения инструкции `try/finally` в так называемые менеджеры контекста
- Чаще всего менеджер контекста используется для управления безопасным подключением и высвобождением системных ресурсов. Ресурсы выделяются при помощи инструкции **with** и высвобождаются автоматически, когда поток исполнения покидает **with**
- Эффективное применение инструкции **with** помогает избежать утечки ресурсов и облегчает ее восприятие.