

---

---

# 15. Итератори и генератори

— 20 април 2026 —

---

---

# Но преди итераторите!

- Всяка жена харесва мъже, които се справят с “битовите операции”



# По-битови операции

- Числата в програмирането се съхраняват като комбинация от 0 и 1-ци, това го знаете
- Обикновено работим с human-readable числа, а не с нули и единици
- Какво правим ако искаме да пипнем по-дълбоко, обаче?
- Enter bitwise operations

<b>Number 1</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>1</b>
<b>Number 2</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>0</b>
-----					
<b>AND</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>
<b>OR</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>
<b>XOR</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>

&, |, ^, ~, >>, <<, (⌋ ◻ ◌)⌋ ⌒ ┌┌┌

- Всяка от тези операции работи директно с двоичната репрезентация на числата
- & - побитово (логическо) **И**
- | - побитово **ИЛИ**
- ^ - побитово изключващо или (**XOR**)
- ~ - побитово отрицание (**НЪЕ**)
- >> - **shift надясно**
- << - **shift наляво**
- (⌋ ◻ ◌)⌋ ⌒ ┌┌┌ - побитово 'бал съм му майката

# Особености

```
>>> bin(-2)
'-0b10'
>>> bin(-3)
'-0b11'
>>> -2 & -3
-4
```

- Отрицателните числа в Python са имплементирани използвайки механизма two's complement
- Реално `-2` не е `-0b10`, това е human-readable двоична репрезентация
- Всъщност е `1.....1111111111111110`
- Няма да навлизаме в детайли как работят компютрите, защото става дълбоко, но предлагаме [добро четиво по темата](#)

# Особености 2

```
>>> 0.1 & 0.2
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#30>", line 1, in <module>
```

```
    0.1 & 0.2
```

```
TypeError: unsupported operand type(s) for &: 'float' and 'float'
```

- Числата с плаваща запетая са имплементирани с експонента и мантиса
- Ерго побитовите операции нямат смисъл в този контекст

# Защо бихме ги използвали?

- Защото пишем на друг език преди 20 години
- Защото са секси и мацките / пичовете (много сме прогресивни) им се кефят
- Защото искаме да слагаме ей такива коментари в кода си:

```
i = * ( long * ) &y;          // evil floating point bit level hacking
i = 0x5f3759df - ( i >> 1 );    // what the fuck?
y = * ( float * ) &i;
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
```

- Флагове
- `chmod 644` някой?

# Още целочислени литерали

- [Познатите ни] шестнадесетични литерали:
  - `0x1234567890abcdef`
- Осмични...
  - `0o12345670`
  - `$ chmod 644 some_path`
  - `>>> os.chmod(some_path, 0o644)`
- Двоични...
  - `0x101001010110101`

# Стига толкова

- Добре, това беше само вметка, защото темата е много кратка
- Да се върнем обратно към основната тема...

# Що е итерация?

- Когато имаме някакви данни, обикновено искаме да ги обхождаме
- Как итериране? - обикновено с `for` loop
- Примери за итеруеми
  - List
  - Set
  - Dict
  - Tuple
  - Map & filter създават итеруеми обекти
  - Всичко в collections
  - *Всеки обект от клас имплементиращ `__getitem__`*

# Итеруемо vs итератор?

- итеруемото (*iterable*) притежава `__iter__` метода
- итератора (*iterator*) притежава `__next__` метода
- Разликата е в “мързеливостта”
- И в това колко пъти можем да итерираме през обекта

# Итеруемо

```
answers_for_the_upcoming_exam = ['a', 'a', 'в', 'г', 'з', '15', 'една кофа']  
for answer in answers_for_the_upcoming_exam:  
    print(answer)
```

a

a

в

г

з

15

една кофа

П.П. Не сме казали, че са **верните** отговори...

# iter

```
>>> dir(answers_for_the_upcoming_exam)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',  
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',  
 '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',  
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
 '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',  
 '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy',  
 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

- `__iter__` се грижи за това да конструира обект, който може да бъде итериран с `for`
- С други думи - той дефинира свойството “итеруемост” на даден обект

# Итератор

```
answers_for_the_upcoming_exam = ['a', 'a', 'в', 'г', 'з', '15', 'една кофа']  
answers_iterator = iter(answers_for_the_upcoming_exam)
```

```
print(answers_iterator) # <list_iterator object at 0x0000020086AEE770>
```

```
print(next(answers_iterator)) # a
```

```
print(next(answers_iterator)) # a
```

```
print(next(answers_iterator)) # в
```

```
print(next(answers_iterator)) # г
```

- Този обект вече е итератор, създаден на базата на итеруемо
- `iter(__iter__)` създава итератора
- `next(__next__)` връща следващата стойността

# next

```
>>> dir(answers_iterator)
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__iter__', '__le__', '__length_hint__', '__lt__',  
 '__ne__', '__new__', '__next__', '__reduce__', '__reduce_ex__', '__repr__',  
 '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__']
```

- `__next__` се грижи за връщане на следващата стойност в стъпката на итерация
- С други думи - дефинира поведението на итератора

# Какво прави for?

```
for answer in answers_for_the_upcoming_exam:  
    print(answer)
```

На практика горният `for` изглежда по подобен начин под капака:

```
iterator = iter(answers_for_the_upcoming_exam) # "iterator" за нас е невидимо  
try:  
    while True:  
        answer = next(iterator)  
        print(answer)  
except StopIteration:  
    pass
```

- `StopIteration` е просто изключение, което индикира “край” на итерираният обект

# Мързеливост

```
answers_for_the_upcoming_exam = ['a', 'a', 'в', 'г', 'з', '15', 'една кофа']
answers_iterator = iter(answers_for_the_upcoming_exam)

print(answers_for_the_upcoming_exam) # ['a', 'в', 'a', 'г', 'з', ...]
print(answers_iterator) # <list_iterator object at 0x00000020086AEE770>
```

- Говорили сме го в контекста на `map`, `filter` и `comprehensions`
- Можем да итерираме и през двата обекта
- Но `answers_iterator` не се оценява до пълното си съдържание, а връща само следващата необходима стойност

# Ама наистина мързеливи

- Не говорим за прокрастиниране от ранга на “предавам си домашното в 17:59”
- Говорим за “от 3 години не съм ходил на зъболекар, ама то не ме боли чак толкова често”
- Нагледно:

```
answers_for_the_upcoming_exam = ['a', 'a', 'в', 'г', 'з', '15', 'една кофа']  
answers_iterator = iter(answers_for_the_upcoming_exam)
```

```
answers_for_the_upcoming_exam[0] = 'леля Гинче домакинката от 51 блок'  
print(next(answers_iterator)) # леля Гинче домакинката от 51 блок
```

# Колко пъти можем да итерираме?

```
for answer in answers_for_the_upcoming_exam:  
    print(answer) # а а в г з 15 една кофа  
for answer in answers_for_the_upcoming_exam:  
    print(answer) # а а в г з 15 една кофа
```

```
answers_iterator = iter(answers_for_the_upcoming_exam)  
for answer in answers_iterator:  
    print(answer) # а а в г з 15 една кофа  
for answer in answers_iterator:  
    print(answer) # НИЩО! Итераторът е "празен"
```

- Многократно можем да итерираме списъци, множества, речници и т.н.
- Колко пъти можем да итерираме е свързано с това дали обектите са мързеливи, така че такива като `map`, `filter` и т.н. - веднъж

# Обобщено за `iter` и `next`

- `iter` се опитва да извика `__iter__` метода на аргумента си
- Ако се окаже, че такъв няма, конструира итератор като просто извиква `__getitem__` с последователни естествени числа започвайки от нула
- Ако има - започва да изпълнява `__next__` на върнатия обект (итератор)
- ... докато не се хвърли грешка

# Можем и сами

```
class Squarer:
    def __init__(self, end=0):
        self.end = end
    def __iter__(self):
        return SquarerIterator(0, self.end)
```

```
class SquarerIterator:
    def __init__(self, number, end):
        self.num = number
        self.end = end

    def __next__(self):
        if self.num <= self.end:
            result = self.num ** 2
            self.num += 1
            return result
        else:
            raise StopIteration
```

# Можем и по-кратко

```
class Squarer:
    def __init__(self, end=0):
        self.end = end

    def __iter__(self):
        self.num = 0
        return self

    def __next__(self):
        if self.num <= self.end:
            result = self.num ** 2
            self.num += 1
            return result
        else:
            raise StopIteration
```

## Можем и сами (2)

```
numbers = Squarer(3)
squares_iter = iter(numbers)
print(next(squares_iter)) # prints 0
print(next(squares_iter)) # prints 1
print(next(squares_iter)) # prints 4
print(next(squares_iter)) # prints 9
print(next(squares_iter))
```

???

```
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    print(next(squares_iter))
  File "<pyshell#60>", line 7, in __next__
    raise StopIteration
StopIteration
```

# Iter 2.0

- `iter` метода има и втора версия, в която въвеждаме втора стойност - така наречения стражник (sentinel). Итерацията ще спре, когато достигнем **ТОЧНО** тази стойност.

```
class Squarerer:  
  
    def __init__(self, start):  
        self.num = start  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        result = self.num ** 2  
        self.num += 1  
        return result  
  
    __call__ = __next__  
  
numbers = iter(Squarerer(1), 100)  
for n in numbers:  
    print(n) # 1 4 9 16 25 36 49 64 81
```



# Кога?

- Ефективност откъм памет
- Възможност да са безкрайни
- Много дълга верига от операции (pipeline)



# Кога не?

- Не позволяват индексирание
- Обхождат се само по веднъж

# Генератори

- Генераторите са също итератори
- Може би най-целесъобразния начин за създаване на итератор
- Може би не
- Така е в живота
- Не пазят стойностите в паметта, а ги генерират когато е необходимо
- Дефинират се като обикновена функция с една основна разлика, вместо return използваме ключовата дума `yield`

# yield

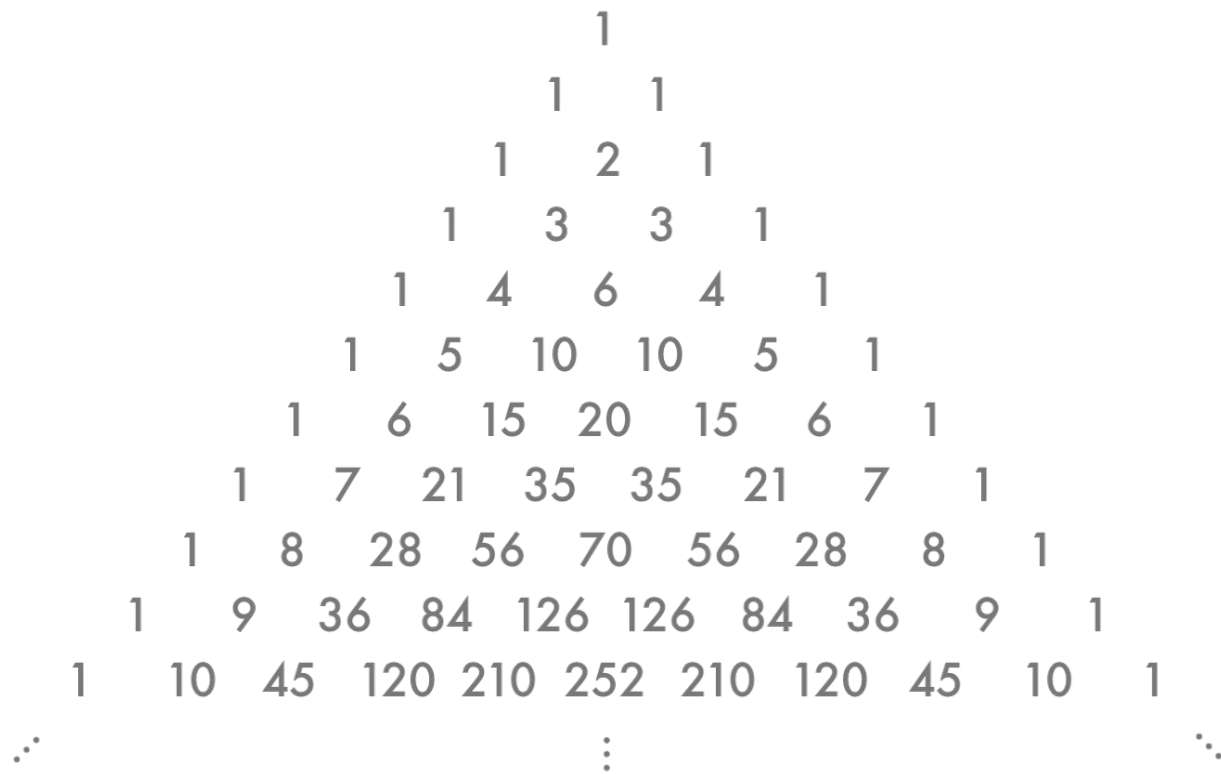
- Ключова дума
- Връща обект
- Албум на Pearl Jam
- `__iter__` и `__next__` се имплементират директно зад завесите
- Когато функцията `yield`-не, не бива терминирана, а просто паузирана
- Локалните променливи и техните текущи състояния са запомнени между последователни извиквания
- Когато функцията приключи, `StopIteration` се възбужда автоматично
- Използват се и за корутини (ще го пропуснем в курса)

# Генератор нагледно

```
def squarererer(start, limit):  
    current = start  
    while current ** 2 < limit:  
        yield current ** 2  
        current += 1  
  
numbers = squarererer(1, 100)  
for n in numbers:  
    print(n) # 1 4 9 16 25 36 49 64 81
```

- Същото нещо като в примера по-рано
- На 2 пъти по-малко редове
- Не можем да дефинираме методи (напр. `__len__`/`__length_hint__`)

# Безкраен генератор на триъгълника на Паскал



```
class PascalTriangle:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        self.row = [self.start]
        return self._generator

    @property
    def _inner(self):
        for left, right in zip(self.row, self.row[1:]):
            yield left + right

    @property
    def _generator(self):
        yield self.row
        self.row = [self.start, *self._inner, self.start]
        yield from self._generator
```

```
# Примерно използване
for row in PascalTriangle(1):
    print(row)
    if len(row) > 20:
        break
```

# list/set/dict comprehension

```
>>> [x ** 2 for x in range(5) if x % 2 == 0]  
[0, 4, 16]
```

```
>>> {x: x ** 2 for x in range(5) if x % 2 == 0}  
{0: 0, 2: 4, 4: 16}
```

```
>>> {x ** 2 for x in range(5) if x % 2 == 0}  
{0, 16, 4}
```

# Generator expression

```
lazy_list_comprehension = (x ** 2 for x in range(5) if x % 2 == 0)
print(lazy_list_comprehension) # <generator object <genexpr> at
0x0000001F9E8276260>
print(list(lazy_list_comprehension)) # [0, 4, 16]
```

- Като list comprehension, но с обли скоби и мързелив
- Показвали сме ви го
- Понякога можем да пропуснем скобите

# Функции по темата

- `any`, `all`
- `map`, `filter`
- `list`, `tuple`, `set`
- `enumerate`
- `zip`

# map и filter

Приемат итерувани и връщат итератори.

```
def numbers():  
    num = 0  
    while True:  
        yield num  
        num += 1  
  
doubles = map(lambda num: num*2, numbers())
```

# functools.reduce

```
from functools import reduce
sum_of_numbers = reduce(lambda a, b: a + b, range(1, 101))

print(sum_of_numbers)
# 5050
```

# itertools.accumulate

```
from itertools import accumulate
sums = accumulate(range(1, 101), lambda a, b: a + b)

print(sums)
# <itertools.accumulate object at 0x1076d27c0>

print(next(sums))
# 1

print(next(sums))
# 3

print(list(sums))
# [6, 10, 15, ..., 4950, 5050]
```

# enumerate

```
necessities = ['Бира', 'Риба', 'Николай или Никола, или някое производно']  
for index, necessity in enumerate(necessities, 1):  
    print(f'{index}. {necessity}')
```

```
# 1. Бира
```

```
# 2. Риба
```

```
# 3. Николай или Никола, или някое производно
```

# zip

```
from itertools import zip_longest
```

```
numbers = [1, 2, 3]  
letters = ['a', 'b', 'c']  
longest = range(5)
```

```
zipped_normal = zip(numbers, letters, longest)  
zipped_longest = zip_longest(numbers, letters, longest, fillvalue='?')
```

```
print(zipped_normal)  
# [(1, 'a', 0), (2, 'b', 1), (3, 'c', 2)]  
print(zipped_longest)  
# [(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), ('?', '?', 3), ('?', '?', 4)]
```

# itertools.combinations

```
from itertools import combinations
```

```
options = ['бира', 'цаца', 'цици', 'плаж']  
combinations_of_two = combinations(options, 2)
```

```
print(list(combinations_of_two))  
# [('бира', 'цаца'), ('бира', 'цици'), ('бира', 'плаж'), ('цаца', 'цици'),  
  ('цаца', 'плаж'), ('цици', 'плаж')]
```

```
print(list(combinations_of_two))  
# []
```

- Не забравяйте - изчерпват се!

# itertools.cycle

```
from itertools import cycle

final_grading_system = cycle([3, 4, 5, 6])
for i in final_grading_system:
    print(i)
# 3
# 4
# 5
# 6
# 3
# 4
# 5
# ...
```

# itertools.count

```
from itertools import count

i = iter(count(firstval=5, step=3))
print(next(i)) # 5
print(next(i)) # 8
print(next(i)) # 11
# ...
```



# itertools.repeat

```
from itertools import repeat

print(list(repeat('Берое!', 3)))
# ['Берое!', 'Берое!', 'Берое!']
```

# Още itertools

- `itertools.repeat(objects[, times])` - връща итеруемо с определен брой (или безкрайно много) повторения на един обект
- `itertools.cycle(iterable)` - безкрайна конкатенация на един итеруем обект
- `itertools.filterfalse(function, iterable)` - filter, тълкуващ предиката на обратно (ако function е None връща falsy елементите)
- `itertools.permutations(iterable)` - пермутациите на елементите в итеруемото
- `itertools.product(*iterables [, repeat=1])` - декартово произведение
- `itertools.takewhile(function, iterable)` - генерира елементите на итеруемото, до първото което не отговаря на предиката
- `itertools.dropwhile(function, iterable)` - генерира елементите на итеруемото, от първото което не отговаря на предиката нататък
- `itertools.tee(iterable, n)` - връща кортеж от n независими итеруеми
- `itertools.chain(*iterables)` - "залепя" няколко итеруеми в един

# Разгадайте itertools

<https://docs.python.org/3.10/library/itertools.html>

**Въпроси?**