# Люби свой yield

Александр Кошкин, Positive Technologies

# Почему генераторы?

1. Мы повсеместно используем итераторы.
2. Воспринимаем генератор как ленивый итератор.
3. Гуглим Generators: The Final Frontier (David Beazley).
4. Понимаем что жизнь прожита зря.

# Таким образом мне бы хотелось

1. Показать чем же на самом деле является genobject.
2. Возможно даже важнее: чем он не является.
3. Как genobject вписывается в общую канву.
4. Показать несколько примеров использования за гранью простого ‘for loop’.

# Как python исполняет код

```
def fac(n):
    return 1 if n==1 else fac(n-1)*n
```

# code object - единица 'кода'

```
def fac(n):
    return 1 if n==1 else fac(n-1)*n
```

```
>>> dis.disassemble(fac.__code__)
2         0 LOAD_FAST                0 (n)
          3 LOAD_CONST               1 (1)
          6 COMPARE_OP               2 (==)
          9 POP_JUMP_IF_FALSE       16
         12 LOAD_CONST               1 (1)
         15 RETURN_VALUE
    >>   16 LOAD_GLOBAL              0 (fac)
         19 LOAD_FAST                0 (n)
         22 LOAD_CONST               1 (1)
         25 BINARY_SUBTRACT
         26 CALL_FUNCTION            1 (...)
         29 LOAD_FAST                0 (n)
         32 BINARY_MULTIPLY
         33 RETURN_VALUE
```

# code objectы повсюду

```
>>> __build_class__
0: <built-in function __build_class__>
```

# code objectы повсюду

```
>>> __build_class__
0: <built-in function __build_class__>

>>> __builtins__['__build_class__'] = lambda *ag, **kw: print(ag, kw)

>>> class A: pass
(<function A at 0x7faec2ea1b70>, 'A') {}
```

Это еще что такое?

https://github.com/magniff/magic

# Python frames

```python
def fac(n):
    return 1 if n==1 else fac(n-1)*n
```

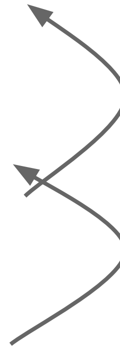
**Как тогда работает рекурсия?**

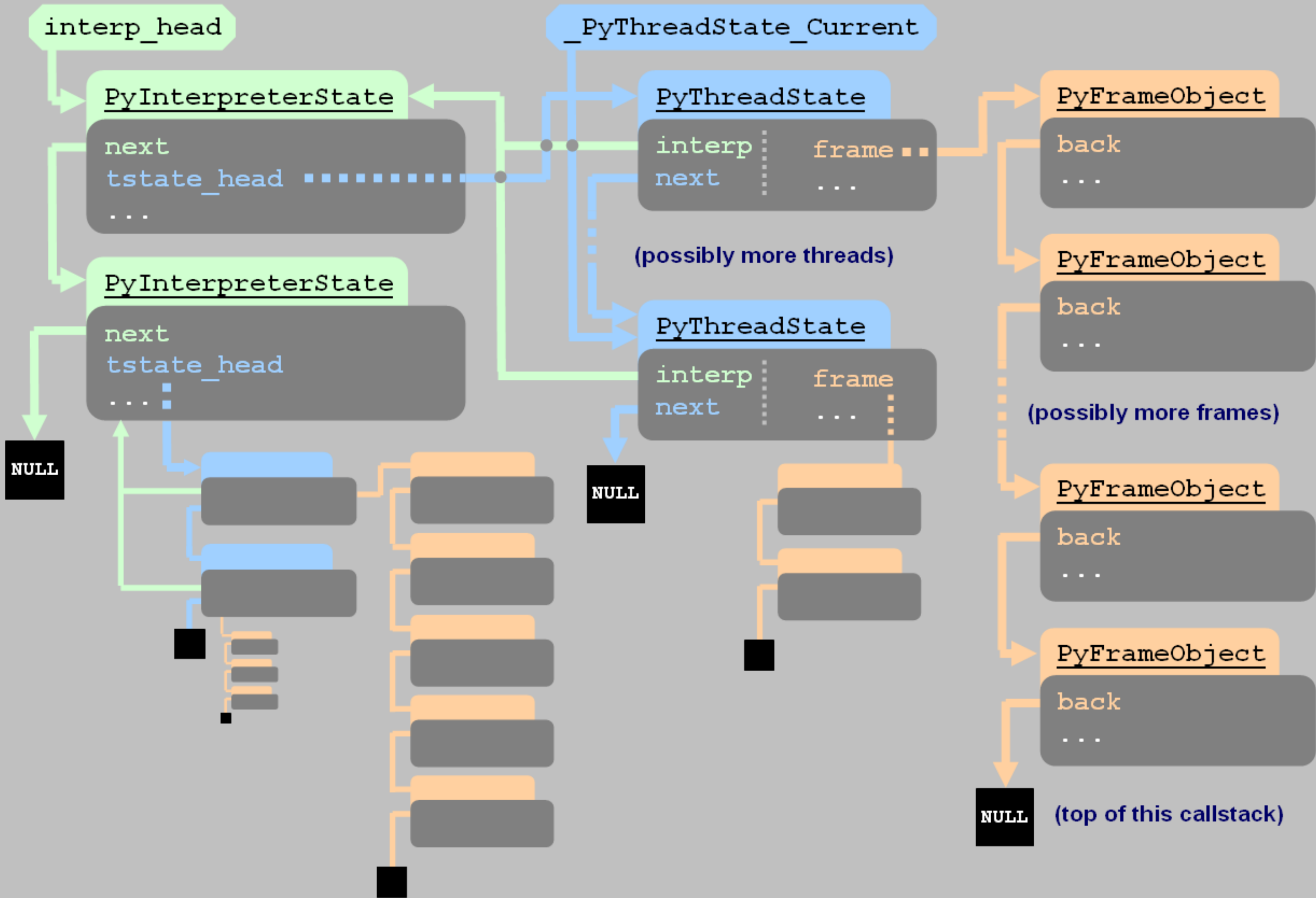
# Python frames

```
def fac(n):
    frame = inspect.currentframe()
    print('current value:', n)
    print('current frame:', frame)
    print('parent  frame:', frame.f_back)
    print('code object', frame.f_code)
    return 1 if n==1 else fac(n-1)*n
```

# Python callstack

```
>>> fac(3)
current value: 3
current frame: <frame object at 0x0000000002FDE748>
parent  frame: <frame object at 0x0000000002FDEAC8>
code object <code object fac at 0x00000000028C1DB0>
current value: 2
current frame: <frame object at 0x0000000002FDEC88>
parent  frame: <frame object at 0x0000000002FDE748>
code object <code object fac at 0x00000000028C1DB0>
current value: 1
current frame: <frame object at 0x0000000002FDEE48>
parent  frame: <frame object at 0x0000000002FDEC88>
code object <code object fac at 0x00000000028C1DB0>
```





interp_head
_PyThreadState_Current
PyInterpreterState
next
tstate_head
...
PyInterpreterState
next
tstate_head
...
NULL
PyThreadState
interp
next
frame
...
(possibly more threads)
PyThreadState
interp
next
frame
...
NULL
PyFrameObject
back
...
PyFrameObject
back
...
(possibly more frames)
PyFrameObject
back
...
PyFrameObject
back
...
NULL
(top of this callstack)
© yaniv aknin

# Откуда берутся генераторы?

Действительно, откуда?

```
def gen(): yield
gen()
```

# Откуда берутся генераторы?

**file: /home/magniff/Desktop/probes/probe2.py**

```
signal.signal(signal.SIGTRAP, lambda *args, **kwargs: None)
stop = lambda: os.kill(os.getpid(), signal.SIGTRAP)


stop()
def gen(): yield
gen()
```

# Расчехляем gdb!

```
gdb --args \
/home/magniff/Downloads/Python-3.4.2/python  \
/home/magniff/Desktop/probes/probe2.py
```

```
(gdb) run
Starting program: /home/magniff/Downloads/Python-3.4.2/python
/home/magniff/Desktop/probes/probe2.py
...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x00007ffff7127c07 in kill () from /lib64/libc.so.6
(gdb)
```

# Расчехляем gdb!

```
(gdb) b PyGen_New
Breakpoint 1 at 0x65818b: file Objects/genobject.c, line 515.
(gdb) cont    # продолжаем
Continuing.

Breakpoint 1, PyGen_New (f=0xa27ec8) at Objects/genobject.c:515
515     PyGenObject *gen = PyObject_GC_New(PyGenObject, &PyGen_Type);
(gdb) bt
```

# native callstack

```
#0  PyGen_New (f=0xa28788) at Objects/genobject.c:515
#1  0x00000000005aa1e8 in PyEval_EvalCodeEx ... at Python/ceval.c:3583
#2  0x00000000005ad219 in fast_function ... at Python/ceval.c:4342
#3  0x00000000005acced in call_function ... at Python/ceval.c:4260
#4  0x00000000005a530a in PyEval_EvalFrameEx ... at Python/ceval.c:2835
#5  0x00000000005aa1fe in PyEval_EvalCodeEx ... at Python/ceval.c:3586
#6  0x0000000000595e0d in PyEval_EvalCode ... at Python/ceval.c:773
#7  0x0000000000423cab in run_mod ... at Python/pythonrun.c:2180
. . .
#13 0x000000000041adaf in main (argc=2, argv=0x7fffffffe548) at .
/Modules/python.c:69
```

# native callstack

```
#0  PyGen_New (f=0xa28788) at Objects/genobject.c:515
#1  0x00000000005aa1e8 in PyEval_EvalCodeEx ... at Python/ceval.c:3583
#2  0x00000000005ad219 in fast_function ... at Python/ceval.c:4342
#3  0x00000000005acced in call_function ... at Python/ceval.c:4260
#4  0x00000000005a530a in PyEval_EvalFrameEx ... at Python/ceval.c:2835
#5  0x00000000005aa1fe in PyEval_EvalCodeEx ... at Python/ceval.c:3586
#6  0x0000000000595e0d in PyEval_EvalCode ... at Python/ceval.c:773
#7  0x0000000000423cab in run_mod ... at Python/pythonrun.c:2180
. . .
#13 0x000000000041adaf in main (argc=2, argv=0x7fffffffe548) at .
/Modules/python.c:69
```

# native callstack

```
#0  PyGen_New (f=0xa28788) at Objects/genobject.c:515
#1  0x00000000005aa1e8 in PyEval_EvalCodeEx ... at Python/ceval.c:3583
#2  0x00000000005ad219 in fast_function ... at Python/ceval.c:4342
#3  0x00000000005acced in call_function ... at Python/ceval.c:4260
#4  0x00000000005a530a in PyEval_EvalFrameEx ... at Python/ceval.c:2835
#5  0x00000000005aa1fe in PyEval_EvalCodeEx ... at Python/ceval.c:3586
#6  0x0000000000595e0d in PyEval_EvalCode ... at Python/ceval.c:773
#7  0x0000000000423cab in run_mod ... at Python/pythonrun.c:2180
. . .
#13 0x000000000041adaf in main (argc=2, argv=0x7fffffffe548) at .
/Modules/python.c:69
```

# native callstack

```
#0  PyGen_New (f=0xa28788) at Objects/genobject.c:515
#1  0x00000000005aa1e8 in PyEval_EvalCodeEx ... at Python/ceval.c:3583
#2  0x00000000005ad219 in fast_function ... at Python/ceval.c:4342
#3  0x00000000005acced in call_function ... at Python/ceval.c:4260
#4  0x00000000005a530a in PyEval_EvalFrameEx ... at Python/ceval.c:2835
#5  0x00000000005aa1fe in PyEval_EvalCodeEx ... at Python/ceval.c:3586
#6  0x0000000000595e0d in PyEval_EvalCode ... at Python/ceval.c:773
#7  0x0000000000423cab in run_mod ... at Python/pythonrun.c:2180
. . .
#13 0x000000000041adaf in main (argc=2, argv=0x7fffffffe548) at .
/Modules/python.c:69
```

# История начинается тут

```
(gdb) frame 4
#4  0x00000000005a530a in PyEval_EvalFrameEx (f=0xa0c398, throwflag=0) at Python/ceval.c:2835
2835                res = call_function(&sp, oparg, &intr0, &intr1);

(gdb) p opcode
$1 = 131
(gdb) list
2830            TARGET(CALL_FUNCTION) {
2831                PyObject **sp, *res;
2832                PCALL(PCALL_ALL);
2833                sp = stack_pointer;
2834    #ifdef WITH_TSC
2835                res = call_function(&sp, oparg, &intr0, &intr1);
2836    #else
2837                res = call_function(&sp, oparg);
2838    #endif
2839                stack_pointer = sp;
```

# PyEval_EvalCodeEx

Python/ceval.c

```
PyObject *
PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals, ...)
{
    PyCodeObject* co = (PyCodeObject*)_co;
    PyFrameObject *f;
    ...
    # TL;DR block
    ...
    if (co->co_flags & CO_GENERATOR) {
        PCALL(PCALL_GENERATOR);
        return PyGen_New(f);
    }

    retval = PyEval_EvalFrameEx(f,0);
    ...
    return retval;
}
```

# Как устроена PyGen_New

Objects/genobject.c

```
PyObject *
PyGen_New(PyFrameObject *f)
{
    PyGenObject *gen = PyObject_GC_New(PyGenObject, &PyGen_Type);
    if (gen == NULL) {
        Py_DECREF(f);
        return NULL;
    }
    gen->gi_frame = f;
    f->f_gen = (PyObject *) gen;
    Py_INCREF(f->f_code);
    gen->gi_code = (PyObject *)(f->f_code);
    gen->gi_running = 0;
    gen->gi_weakreflist = NULL;
    _PyObject_GC_TRACK(gen);
    return (PyObject *)gen;
}
```

# Как устроен PyGenObject

Include/genobject.h

```
typedef struct {
    PyObject_HEAD
    struct _frame *gi_frame;
    char gi_running;
    PyObject *gi_code;
    PyObject *gi_weakreflist;
} PyGenObject;
```

Все совсем просто

# PEP 255

```
>>> def my_gen():
...     print('Generator body.')
...     yield 'Hello!'
>>> my_gen
0: <function my_gen at 0x7f2293d16620>
>>> hex(my_gen.__code__.co_flags)
1: '0x63'
>>> gen = my_gen()
>>> gen
2: <generator object my_gen at 0x7f2291802d38>
>>> gen.gi_frame
3: <frame object at 0x7f22917e1ac8>
>>> gen.gi_code
4: <code object my_gen at 0x7f2291a334b0, file "<pyshell#0>", line 1>
>>> gen.gi_frame.f_builtins
5: {'ArithmeticError': <class 'ArithmeticError'>, ...}
>>> next(gen)
Generator body.
6: 'Hello!'
>>> next(gen)    # Stop iteration
```

1. В 2.2 появились простые генераторы
2. Ну правда простые (try/except)
3. Поддержка yield statement

# PEP 342

```
>>> def my_gen():
...     yield 'Hello'
...     value = yield 'world'
...     print('got value', value)
>>> g = my_gen()
>>> g
0: <generator object my_gen at 0x000000000298A318>
>>> next(g)     # g.send(None)
1: 'Hello'
>>> next(g)
2: 'world'
>>> g.send('pep342')
got value pep342
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    g.send('pep342')
StopIteration
```

1. В 2.5 появились “корутины”
2. yield стал expression`ом
3. новые методы send(), throw(), close()

# PEP 380 aka yield from

```
>>> def chain(a,b):
...     yield from a
...     yield from b
>>> chain(chain([1,2,3], [4, 5, 6]), [7, 8, 9])
0: <generator object chain at 0x7f3897326d38>
>>> list(_)
1: [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> def fac(n):
...     return 1 if n==1 else (yield from fac(n-1))*n
>>> next(fac(10))
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    next(fac(10))
StopIteration: 3628800
```

Это работает через проброс send() через весь стек

1. В 3.3 можно делегировать генерацию
2. Делает re-send, re-throw
3. StopIteration как возвращаемое значение

# yield from не так хорош

```
list(chain(chain([1, 2, 3], [4, 5, 6]), [7, 8, 9]))
```

```
#7  0x00000000005a530a in PyEval_EvalFrameEx (f=0xa2b808,. . .)
#8  0x0000000000656a58 in gen_send_ex (gen=0x7ffff70efc88, . . .)
#9  0x0000000000656f6e in _PyGen_Send (gen=0x7ffff70efc88, . . .)
#10 0x000000000059cc30 in PyEval_EvalFrameEx (f=0xa28fa8, throwflag=0) at
#11 0x0000000000656a58 in gen_send_ex (gen=0x7ffff7065238, . . .)
#12 0x0000000000657e47 in gen_iternext (gen=0x7ffff7065238) at
#13 0x0000000000498101 in listextend (self=0x7ffff706d7c8, b=0x7ffff7065238) at
#14 0x000000000049ba57 in list_init (self=0x7ffff706d7c8, args=0x7ffff70e0878,
#15 0x00000000004dc509 in type_call (type=0x910bc0 <PyList_Type>,
#16 0x00000000004603fa in PyObject_Call (func=0x910bc0 <PyList_Type>,
#17 0x00000000005adadf in do_call (func=0x910bc0 <PyList_Type>,
#18 0x00000000005acd0c in call_function (pp_stack=0x7fffffffc0f0, oparg=1,
#19 0x00000000005a530a in PyEval_EvalFrameEx (f=0xa0cbb8, throwflag=0) at
```

Если высота дерева велика, нам обеспечен старый добрый RuntimeError: maximum recursion depth exceeded.

# yield from не так хорош

## Optimisations

Using a specialised syntax opens up possibilities for optimisation when there is a long chain of generators. Such chains can arise, for instance, when recursively traversing a tree structure. The overhead of passing `__next__()` calls and yielded values down and up the chain can cause what ought to be an O(n) operation to become, in the worst case, O(n**2).

A possible strategy is to add a slot to generator objects to hold a generator being delegated to. When a `__next__()` or `send()` call is made on the generator, this slot is checked first, and if it is nonempty, the generator that it references is resumed instead. If it raises StopIteration, the slot is cleared and the main generator is resumed.

This would reduce the delegation overhead to a chain of C function calls involving no Python code execution. A possible enhancement would be to traverse the whole chain of generators in a loop and directly resume the one at the end, although the handling of StopIteration is more complicated then.

# корутины не так уж хороши

```
>>> @coroutine
... def ping():
...     while 1:
...         value = yield
...         print('ping: got value', value)
...         pong.send(value+1)
...
... @coroutine
... def pong():
...     while 1:
...         value = yield
...         print('pong: got value', value)
...         ping.send(value+1)
>>> ping=ping()
>>> pong=pong()
>>> ping.send(1)
```

# корутины не так уж хороши

```
>>> @coroutine
... def ping():
...     while 1:
...         value = yield
...         print('ping: got value', value)
...         pong.send(value+1)
...
... @coroutine
... def pong():
...     while 1:
...         value = yield
...         print('pong: got value', value)
...         ping.send(value+1)
>>> ping=ping()
>>> pong=pong()
>>> ping.send(1)
ping: got value 1
pong: got value 2
Traceback (most recent call last): ...
ValueError: generator already executing
```

# можно переписать так

```python
from collections import deque

_queue = deque()
_registry = {}


def run():
    while _queue:
        name, value = _queue.popleft()
        _registry[name].send(value)

def send(name, value):
    _queue.append((name, value))
```

```python
@coroutine
def ping():
    while 1:
        value = yield
        print('ping: got value', value)
        send('pong', value+1)

@coroutine
def pong():
    while 1:
        value = yield
        print('pong: got value', value)
        send('ping', value+1)
```

```
>>> send('ping', 1); run()
ping: got value 1
pong: got value 2
ping: got value 3
. . .
```

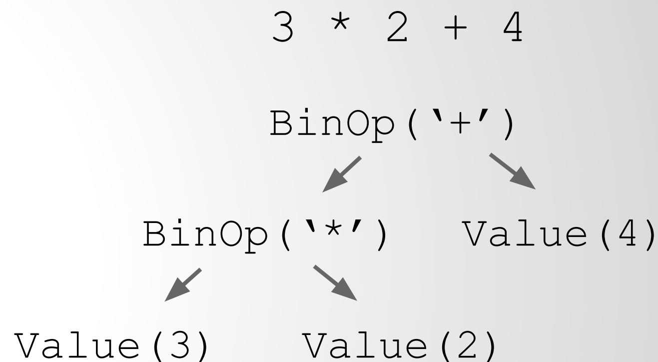
# Beazley и его 'компилятор'

```
text = '2 + 3*4 - 5'
toks = tokenize(text)
tree = parse(toks)
result = Evaluator().visit(tree)


class Node:
    _fields = []
    def __init__(self, *args):
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

class BinOp(Node):
    _fields = ['op', 'left', 'right']

class Number(Node):
    _fields = ['value']
```




code by david beazley http://dabeaz.com/finalgenerator/compiler1.py

# простой визитор

```
class NodeVisitor:
    def visit(self, node):
        return getattr(self, 'visit_' + type(node).__name__)(node)

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_BinOp(self, node):
        leftval = self.visit(node.left)
        rightval = self.visit(node.right)
        if node.op == '+':
            return leftval + rightval
        elif node.op == '-':
            return leftval - rightval
        elif node.op == '*':
            return leftval * rightval
        elif node.op == '/':
            return leftval * rightval
```

code by david beazley http://dabeaz.com/finalgenerator/compiler1.py

# ‘вам тут не рады’ визитор

```
class NodeVisitor:
    def visit(self, node):
        stack = [ self.genvisit(node) ]
        result = None
        while stack:
            try:
                node = stack[-1].send(result)
                stack.append(self.genvisit(node))
                result = None
            except StopIteration as exc:
                stack.pop()
                result = exc.value
        return result

    def genvisit(self, node):
        result = getattr(self, 'visit_' + type(node).__name__)(node)
        return (yield from result) if isinstance(result, types.GeneratorType) else result
```

code by david beazley http://dabeaz.com/finalgenerator/compiler2.py

# пробуем

```
text = '+'.join(str(x) for x in range(1000))
toks = tokenize(text)
tree = parse(toks)
print(Evaluator().visit(tree))
```

code by david beazley http://dabeaz.com/finalgenerator/compiler2.py

# пробуем

```
text = '+'.join(str(x) for x in range(1000))
toks = tokenize(text)
tree = parse(toks)
print(Evaluator().visit(tree))
```

```
>>> 499500
```

code by david beazley http://dabeaz.com/finalgenerator/compiler2.py

# Таким образом генератор это:

1. Простая структура **PyGenObject**.
2. Эта структура держит ссылку на питоновский фрейм.
3. Фрейм знает о своем состоянии: текущее положение в байткоде, состояние локальных переменных.
4. Фрейм можно запустить с точки останова, и остановить в точке, отличной от RETURN_VALUE, как это принято у всех порядочных фреймов.
5. Взаимодействие с генератором всегда блокирует вызывающий код.
6. Генератор во многом похож на обычную функцию, только его нельзя вызывать, зато он поддерживает итерации.

# Менеджер контекста

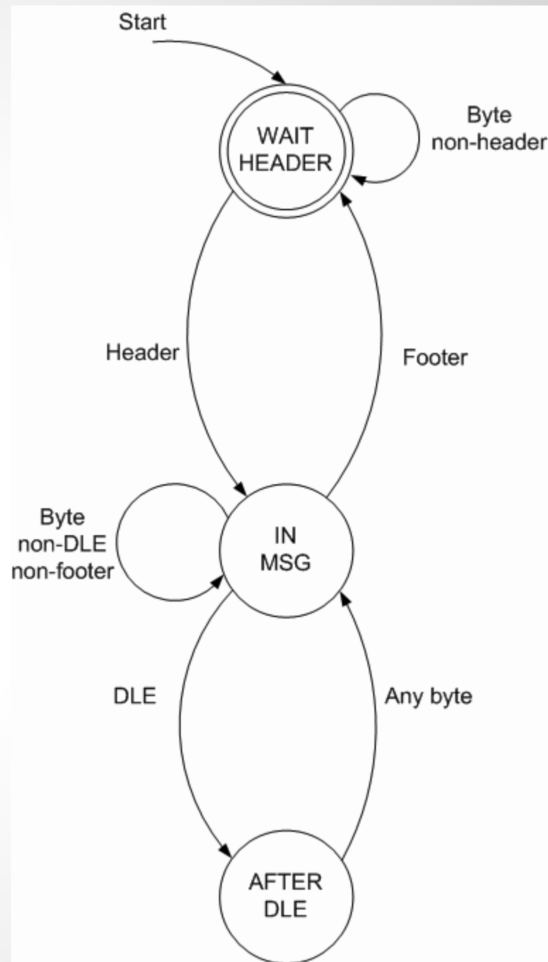
```
from contextlib import contextmanager


@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)


>>> with tag("h1"):
...     print("foo")
...
<h1>
foo
</h1>
```

# Конечный автомат

```
@coroutine
def match_frame (header='\x61',
                 footer ='\x62',
                 dle ='\xAB',
                 after_dle_func =lambda x: x,
                 target =None):
    while True:
        byte = yield
        frame = ''
        if byte == header:
            while True:
                byte = yield
                if byte == footer:
                    target .send(frame)
                    break
                elif byte == dle:
                    byte = yield
                    frame += after_dle_func (byte)
                else:
                    frame += byte
```



# Автомат: делаем ‘сток’

```
@coroutine
def frame_receiver():
    """ A simple co-routine "sink" for receiving
        full frames.
    """
    while True:
        frame = yield
        print('Got frame:', frame.encode('hex'))



unwrapper = match_frame(target=frame_receiver())
```

# Автомат: пример разбора

```
bytes = ''.join(chr(b) for b in
                [0x70, 0x24,
                 0x61, 0x99, 0xAF, 0xD1, 0x62,
                 0x56, 0x62,
                 0x61, 0xAB, 0xAB, 0x14, 0x62,
                 0x7
                ])


for byte in bytes:
    unwrapper.send(byte)


Got frame: 99afd1
Got frame: ab14
```

# Пайпы-генераторы

https://github.com/JulienPalard/Pipe

```
class Pipe:

    def __init__(self, function):
        self.function = function

    def __ror__(self, other):
        return self.function(other)

    def __call__(self, *args, **kwargs):
        return Pipe(lambda x: self.function(x, *args, **kwargs))


@Pipe
def where(iterable, predicate):
    return (x for x in iterable if predicate(x))
```

# Пайпы-генераторы: примеры

```
from pipe import *
>>> [1,2,3,4] | where(lambda x: x<=2)
#<generator object <genexpr> at 0x88231e4>


>>> [1, 2, 3, 4, 5, 6] | average
3.5


>>> [1, 2, 3] | select(lambda x: x * x) | concat
'1, 4, 9'


>>> (1, 3, 5, 6, 7) | any(lambda x: x > 7)
False
```

десятки примеров в гитхабе автора

# Пайпы-генераторы: есть нюанс

```
>>> a = [1,2,3] | where(lambda x: x == 2)
>>> a
<generator object <genexpr> at 0x7fe3362fd0d8>


>>> list(a)
[2]
>>> list(a)
[]
```

**Такое поведение в принципе не удивительно, но в данном случае нежелательно.**

# Пайпы-корутины

https://github.com/magniff/copper

# copper: простой пример

```python
from copper import IteratorBasedSource, Filter, Apply, OutLines, mainloop

source = IteratorBasedSource(range(1,4))
source >> Apply(lambda x: x**2) >> OutLines()

mainloop.run(source)
```

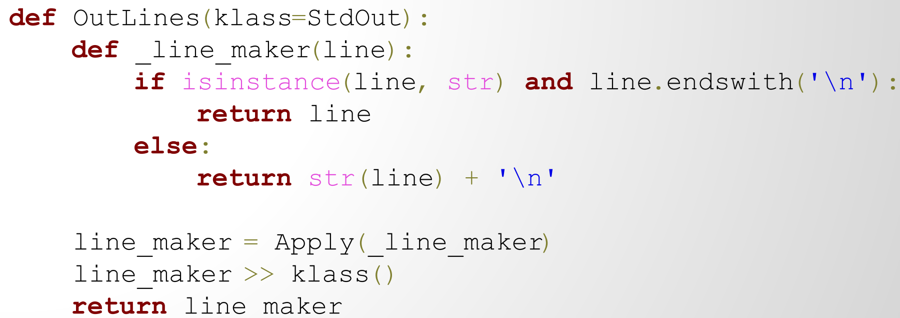
```
python examples/simple_numbers.py
1
4
9
```

# copper: простой пример

```
from copper import IteratorBasedSource, Filter, Apply, OutLines, mainloop


source = IteratorBasedSource(range(1,4))
source >> Apply(lambda x: x**2) >> OutLines()

mainloop.run(source)
```



```
def OutLines(klass=StdOut):
    def _line_maker(line):
        if isinstance(line, str) and line.endswith('\n'):
            return line
        else:
            return str(line) + '\n'

    line_maker = Apply(_line_maker)
    line_maker >> klass()
    return line_maker
```

```
python examples/simple_numbers.py
1
4
9
```

# copper: числа фибоначчи

```
def inp():
    yield (1, 1)


source = IteratorBasedSource(inp())
fib = Apply(lambda x: (x[1], x[0]+x[1]))

source >> fib

fib >> fib
fib >> Apply(lambda x: x[0]) >> OutLines()
```

# copper: числа фибоначчи

```
source = IteratorBasedSource(inp())
fib = Apply(lambda x: (x[1], x[0]+x[1]))

source >> fib

fib >> fib

fib_unpacked = fib >> Apply(lambda x: x[0])


fib_unpacked >> FSM(devide) >> OutLines()
fib_unpacked >> FSFileWriter('fib_data.txt')
```

# copper: числа фибоначчи

```
source = IteratorBasedSource(inp())
fib = Apply(lambda x: (x[1], x[0]+x[1]))

source >> fib

fib >> fib

fib_unpacked = fib >> Apply(lambda x: x[0])
```

```
def devide(callback):
    while 1:
        c0 = yield
        c1 = yield
        callback(
            Decimal(c1)/Decimal(c0)
        )
```

```
fib_unpacked >> FSM(devide) >> OutLines()
fib_unpacked >> FSFileWriter('fib_data.txt')
```

# copper: io

```python
from copper import StdIn, Apply, StdOut, mainloop

source = StdIn()
source >> Apply(lambda line: 'copper: '+line) >> StdOut()
mainloop.run(source)
```

```
python examples/basic_stdin.py
100
copper: 100
Hello
copper: Hello
```

# copper: io

```
echo 'test pipe' | python examples/basic_stdin.py
copper: test pipe
```

# copper: sed (kinda)

```
#! /usr/bin/env python

import argparse
from copper import FSFileReader, Apply, StdIn, StdOut, mainloop

parser = argparse.ArgumentParser()
parser.add_argument('-f', dest='file', type=str)
parser.add_argument('-s', dest='sub', nargs=2, required=1)
args = parser.parse_args()

source = FSFileReader(args.file) if args.file else StdIn()
source >> Apply(lambda line: line.replace(*args.sub)) >> StdOut()

mainloop.run(source)
```

# copper: sed (kinda)

```
echo 'hello gvido' | sed.py -s gvido david
hello david


cat > hello.txt
I hate python

sed.py -s hate love -f hello.txt
I love python
```

# greenlets, stackless & co

Думаю, хватит.
Спасибо за внимание.