Lazy evaluation:

Объект будет создан не когда он объявлен, а когда это нам будет нужно.

In [ ]:
```
# python2
type(xrange(10)) # list

# python3
type(range(10)) # range == generator
```

if statement with or/and is lazy evaluated

In [18]:
```
from timeit import timeit as tm

print(tm('all(range(1,1000))'))
print(tm('any(range(1,1000))'))
```

```
11.895632498999476
0.23312186600014684
```

In [19]:
```
print(tm('all([False] * 1000)'))
print(tm('any([False] * 1000)'))
```

```
2.0864930290008488
5.446446754001954
```

In [17]:
```
def foo(a=None):
    a = a or []
```

In [23]:
```
some_condition_met = True


def do_one():
    print(...)
    return True if some_condition_met else False

def do_two():
    print(type(...))

do_one() and do_two()
```

```
Ellipsis
<class 'ellipsis'>
```

In [18]:
```
Ellipsis == ...
```

Out[18]:
```
True
```

ternary conditional operator

<expression 1> if else <expression 2>

In [24]:
```
a = 1
b = 2

1 if a > b else -1
# Output is -1

1 if a > b else -1 if a < b else 0
# Output is -1
```

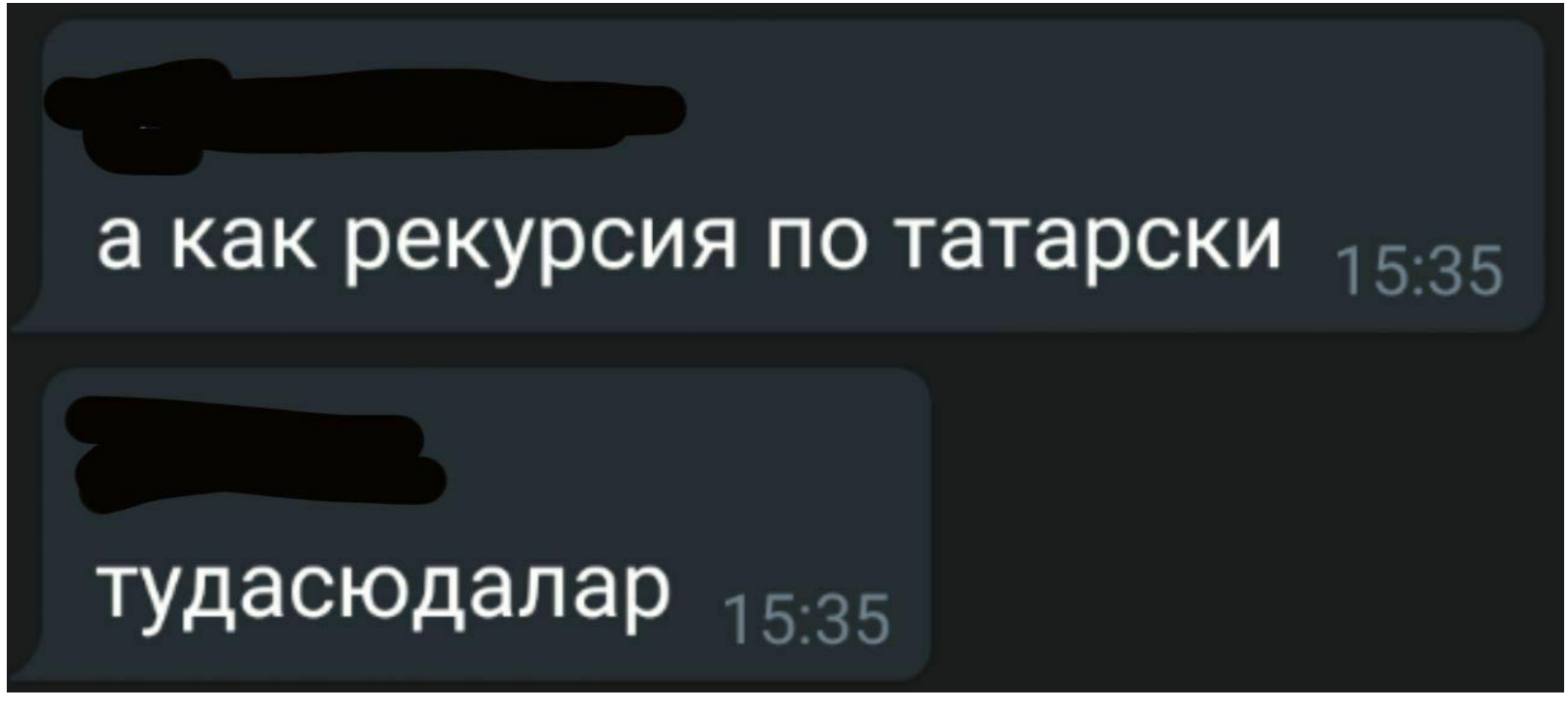
Out[24]:
```
-1
```

Отличается от своих собратьев в других языках `condition ? a : b`

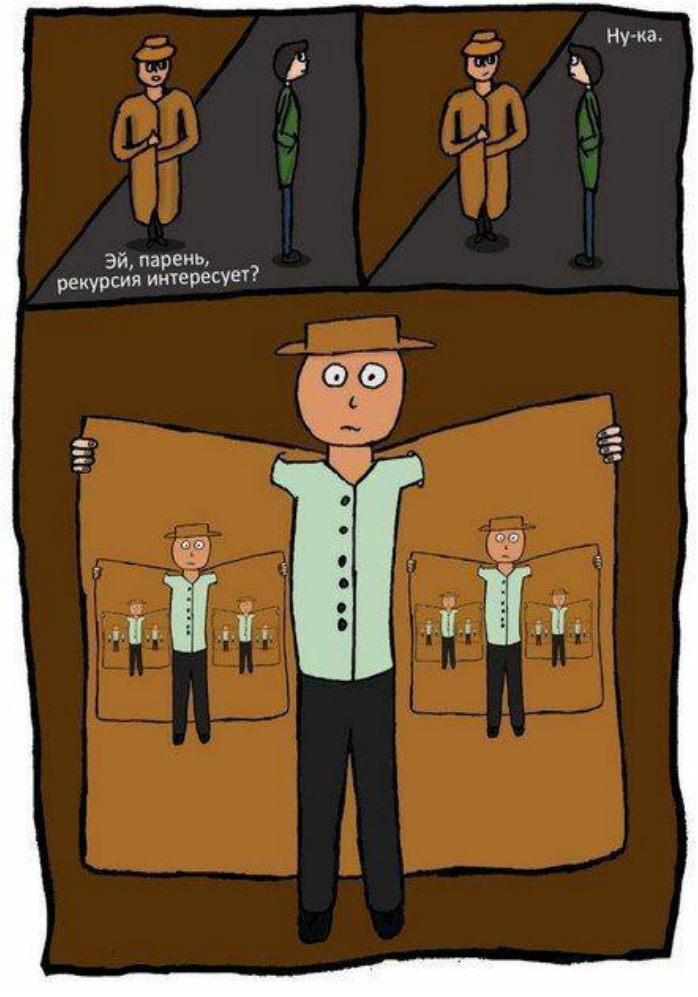
Другие варианты, но все тоже самое:

In [ ]:
```
{True:"yes", False:"no"}[boolean]

{True:"yes", False:"no", None:"maybe"}[boolean_or_none]

("no", "yes")[boolean]
```

Рекурсия


[redacted]
а как рекурсия по татарски
15:35
[redacted]
тудасюдалар
15:35

В программировании рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция `A` вызывает функцию `B`, а функция `B` — функцию `A`.



Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

In [1]:
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

In [2]:
```
print(factorial(4))  # 4 * 3 * 2 * 1 = 24
```

```
24
```

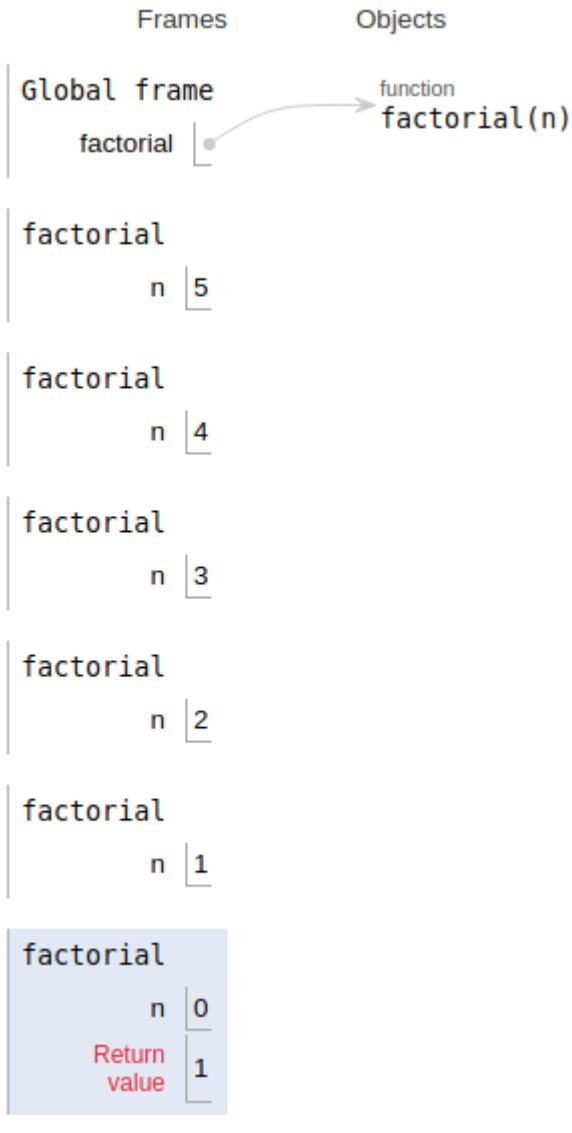
In [5]:
```
print(factorial(4000))  # OOOPS (((
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-5-f16d10eb413b> in <module>
----> 1 print(factorial(4000))  # OOOPS (((
      2 import sys
      3 sys.getrecursionlimit()

<ipython-input-3-324b4a43fba5> in factorial(n)
      2     if n == 0:
      3         return 1
----> 4     return n * factorial(n - 1)

... last 1 frames repeated, from the frame below ...

<ipython-input-3-324b4a43fba5> in factorial(n)
      2     if n == 0:
      3         return 1
----> 4     return n * factorial(n - 1)

RecursionError: maximum recursion depth exceeded in comparison
```

In [6]:
```
import sys
sys.getrecursionlimit()
```

Out[6]:
```
3000
```

Pythontutor (http://www.pythontutor.com/visualize.html)


Frames
Objects
Global frame
factorial
function
factorial(n)
factorial
n
5
factorial
n
4
factorial
n
3
factorial
n
2
factorial
n
1
factorial
n
0
Return value
1

Чуть более реалистичный вариант

In [3]:

```python
def double_all_elements(lst):
    """ Double all elements in list
    :param lst: incoming list
    :return: result list
    """

    if len(lst) == 0:
        return []
    else:
        updated_element = lst[0] * 2
        print(updated_element, len(lst))
        result = [updated_element, ] + double_all_elements(lst[1:])
    print('return list: ', result)
    return result

double_all_elements(list(range(10)))
```

```
0 10
2 9
4 8
6 7
8 6
10 5
12 4
14 3
16 2
18 1
return list:  [18]
return list:  [16, 18]
return list:  [14, 16, 18]
return list:  [12, 14, 16, 18]
return list:  [10, 12, 14, 16, 18]
return list:  [8, 10, 12, 14, 16, 18]
return list:  [6, 8, 10, 12, 14, 16, 18]
return list:  [4, 6, 8, 10, 12, 14, 16, 18]
return list:  [2, 4, 6, 8, 10, 12, 14, 16, 18]
return list:  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Tail recursion

Частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Подобный вид рекурсии примечателен тем, что может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции.

In [60]:

```
def double_all_elements(lst, result_lst=None):
    """ Double all elements in list (tail recursion example)
    :param lst: incoming list
    :return: result list
    """
    if result_lst == None:
        result_lst = []

    if len(lst) == 0:
        return result_lst
    else:
        updated_element = lst[0] * 2
        result_lst.append(updated_element)
        print(updated_element, len(lst), result_lst)
        result = double_all_elements(lst[1:], result_lst)
        print(result)
    return result

double_all_elements(list(range(10)))
```

```
0 10 [0]
2 9 [0, 2]
4 8 [0, 2, 4]
6 7 [0, 2, 4, 6]
8 6 [0, 2, 4, 6, 8]
10 5 [0, 2, 4, 6, 8, 10]
12 4 [0, 2, 4, 6, 8, 10, 12]
14 3 [0, 2, 4, 6, 8, 10, 12, 14]
16 2 [0, 2, 4, 6, 8, 10, 12, 14, 16]
18 1 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [52]:

```
def double_all_elements(lst, result_lst=None):
    """  Double all elements in list (without recursion)
    :param lst: incoming list
    :return: result list
    """

    if result_lst == None:
        result_lst = []

    while len(lst) > 0:
        updated_element = lst[0] * 2
        print(updated_element, len(lst), result_lst)
        (lst, result_lst) = (lst[1:], result_lst + [updated_element, ])
    return result_lst

double_all_elements(list(range(10)))
```

```
0 10 []
2 9 [0]
4 8 [0, 2]
6 7 [0, 2, 4]
8 6 [0, 2, 4, 6]
10 5 [0, 2, 4, 6, 8]
12 4 [0, 2, 4, 6, 8, 10]
14 3 [0, 2, 4, 6, 8, 10, 12]
16 2 [0, 2, 4, 6, 8, 10, 12, 14]
18 1 [0, 2, 4, 6, 8, 10, 12, 14, 16]
```

Out[52]:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Декоратор

Декоратор — функция, которая принимает другую функцию и что-то возвращает.

Синаксис:

In [ ]:
```
@check
def foo():
    return 'moo'
```

In [ ]:
```
def foo(x):
    return 'moo'

foo = check(foo)
```

In [10]:
```
def check(func):
    def inner(*args, **kwargs):
        res =  func(*args, **kwargs)
        print('We have done some work inside decorator')
        return res
    return inner

@check
def foo():
    return 'moo'

foo = check(foo)
foo()
```

```
We have done some work inside decorator
```

Out[10]:
```
'moo'
```

После всех манипуляций `foo` будет доступно то, что вернул декоратор

Он может вернуть объект любого типа

Грокаем декораторы

In [6]:
```
def check(func):
    def wrapper(*args, **kwargs):
        print('name:',func.__name__, '\ndoc: ', func.__doc__)
        return func(*args, **kwargs)
    return wrapper

@check
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
```

```
name: foo
doc:  I am absulutely useless foo
```

Out[6]:
```
'moo'
```

In [9]:

```
def check(func):
    def wrapper(*args, **kwargs):
        print('name:',func.__name__, '\ndoc: ', func.__doc__)
        return func(*args, **kwargs)
    return wrapper


def foo(*args, **kwargs):
    """I am absulutely useless foo"""
    return 'moo'

foo = check(foo)

print(foo, foo.__name__)
print()

foo(1, 2, 3)
```

```
<function check.<locals>.wrapper at 0x7f9fdc1fbc80> wrapper

name: foo
doc:  I am absulutely useless foo
```

Out[9]:

```
'moo'
```

Проблема:

In [7]:
```
help(foo)
```

```
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

Наивное решение:

In [12]:
```
def check(func):
    def wrapper(*args, **kwargs):
        print('name:',func.__name__, '\ndoc: ', func.__doc__)
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    return wrapper

@check
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()


print(help(foo))
```

```
name: foo
doc:  I am absulutely useless foo
Help on function foo in module __main__:

foo(*args, **kwargs)
    I am absulutely useless foo

None
```

functools

In [13]:
```
import functools

def check(func):
    def wrapper(*args, **kwargs):
        print('name:',func.__name__, '\ndoc: ', func.__doc__)
        return func(*args, **kwargs)
    functools.update_wrapper(wrapper, func)
    return wrapper

@check
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
help(foo)
```

```
name: foo
doc:  I am absulutely useless foo
Help on function foo in module __main__:

foo()
    I am absulutely useless foo
```

Или добавим декоратор в декоратор:

In [4]:
```
import functools

def check(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('name:',func.__name__, '\ndoc: ', func.__doc__)
        return func(*args, **kwargs)
    return wrapper

@check
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
help(foo)
```

```
name: foo
doc:  I am absulutely useless foo
Help on function foo in module __main__:

foo()
    I am absulutely useless foo
```

Декоратор с аргументами

In [8]:
```
@check_with_param('name')
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
```

```
name: foo
```

Out[8]:
```
'moo'
```

In [ ]:
```
def foo():
    """I am absulutely useless foo"""
    return 'moo'

deco = check(param=name)
foo = deco(foo)
```

Реализация

In [2]:
```
def check_with_param(param='both'):
    def decorator(func):
        @functools.wraps(func)
        def inner(*args, **kwargs):
            if param == 'both':
                print('name:',func.__name__, '\ndoc: ', func.__doc__)
            else:
                print('name:', func.__name__) if param == 'name' else print('do
c:', func.__doc__)
            return func(*args, **kwargs)
        return inner
    return decorator
```

В общем виде декоратор с аргументами выглядит так:

In [9]:
```
@check_with_param(param='both')
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
```

```
name: foo
doc:  I am absulutely useless foo
```

Out[9]:
```
'moo'
```

In [12]:
```
@check_with_param
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-2ea0c1a35c17> in <module>
      4     return 'moo'
      5
----> 6 foo()

TypeError: decorator() missing 1 required positional argument: 'func'
```

In [14]:
```
@check_with_param()
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo()
```

```
name: foo
doc:  I am absulutely useless foo
```

Out[14]:
```
'moo'
```

In [15]:

```
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo = check_with_param(foo)
foo()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-b6435e081987> in <module>
      4
      5 foo = check_with_param(foo)
----> 6 foo()
      7

TypeError: decorator() missing 1 required positional argument: 'func'
```

In [16]:
```
def foo():
    """I am absulutely useless foo"""
    return 'moo'

foo = check_with_param()(foo)
foo()
```

```
name: foo
doc:  I am absulutely useless foo
```

Out[16]:
```
'moo'
```

Синтаксис Python разрешает одновременное применение нескольких декораторов.

Порядок имеет значение

In [23]:
```
def square(func):
    return lambda x: func(x * x)

def addsome(func):
    return lambda x: func(x + 10)

@square
@addsome
def foo(x):
    return x

foo(2)
```

Out[23]: 14

In [24]:
```
@addsome
@square
def foo(x):
    return x

foo(2)
```

Out[24]: 144

Итого

- Декоратор - способ модифицировать поведение функции, сохраняя читаемость кода
- Декораторы бывают:
    - без аргументов `@check`
    - с аргументами:
        - с позиционными `@check_with_param('both')`
        - с опциональными `@check_with_param(param='both')`

functools

functools: lru_cach

Сохраняет фиксированное колличество поледних вызовов.

In [22]:
```
@functools.lru_cache(maxsize=64)
def fib(n):
    return fib(n-1) + fib(n-2) if n > 0 else 1

fib(100)
```

Out[22]:
```
927372692193078999176
```

In [23]:
```
fib.cache_info()
```

Out[23]:
```
CacheInfo(hits=99, misses=102, maxsize=64, currsize=64)
```

Partial

Позволяет зафиксировать часть позиционных и ключевых аргументов в функции

In [50]:
```
f = functools.partial(sorted, key=lambda p: p[1])

f([("a", 4), ("b", 2)])
```

Out[50]:
```
[('b', 2), ('a', 4)]
```

In [51]:
```
g = functools.partial(sorted, [2, 3, 1, 4])
g()
```

Out[51]:
```
[1, 2, 3, 4]
```