```
def rational(n, d):
    def select(name):
        if name == 'n':
            return n
        elif name == 'd':
            return d
    return select

def numer(x):
    return x('n')

def denom(x):
    return x('d')
```

Эта функция отражает рациональное число

Конструктор — функция высшего порядка

Селектор вызывает функцию x

Длина. Последовательность имеет конечную длину.

Выбор элемента. Элементам последовательности соответствуют индексы — порядковые номера (неотрицательные целые начинающиеся с 0).

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8

>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30

>>> 1 in digits
True
>>> 8 in digits
True
>>> 5 not in digits
True
>>> not(5 in digits)
True
```

1. Вычислить заголовочное <выражение>, результат должен быть итерируемым значением, то есть последовательностью.
2. Для каждого элемента этой последовательности, по порядку:
   A. Связать <имя> с этим элементом в текущем фрейме.
   B. Выполнить <набор>.

Последовательность последовательностей фиксированной длины

```
>>> pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
>>> same_count = 0

>>> for x, y in pairs:
...     if x == y:
...         same_count = same_count + 1

>>> same_count
2
```

Имя для каждого элемента последовательности фиксированной длины

```
..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...
                 range(-2, 2)
```

Длина: конечное значение − начальное значение

Выбор элемента: начальное значение + индекс

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]

>>> list(range(4))
[0, 1, 2, 3]
```

Конструктор списка

По-умолчанию, диапазон начинается с нуля

Принадлежность.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Срез.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Возвращает новый объект

Значение называют *итерируемым* если оно может быть использовано для получения итератора.

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

```
>>> 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)
```

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

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

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

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

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

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

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3

def a_then_b(a, b):
    yield from a
    yield from b
>>> list(a_then_b( [3, 4], [5, 6] ))
[3, 4, 5, 6]
```

```
[<отображающее выраж.> for <имя> in <итерируемое выраж.> if <фильтрующее выраж.>]
```

Короткая версия: `[<отображающее выраж.> for <имя> in <итерируемое выраж.>]`

Сложное выражение, результат которого является списком и получается по правилам:

1. Добавить новый фрейм полагая текущий фрейм родительским.
2. Создать пустой *результирующий список* который будет значением выражения.
3. Для каждого элемента в итерируемом значении выражения <итерируемое выраж.>:
   A. Во фрейме из шага 1 связать <имя>
   B. Если <фильтрующее выражение> имеет истинное значение, тогда значение <отображающего выражения> заносится в *результирующий список.*

Для значений результат применения **repr** соответствует тому, что Python выводит в интерактивной сессии

```
>>> 12e12
12000000000000.0
```

```
>>> print(repr(12e12))
12000000000000.0
```

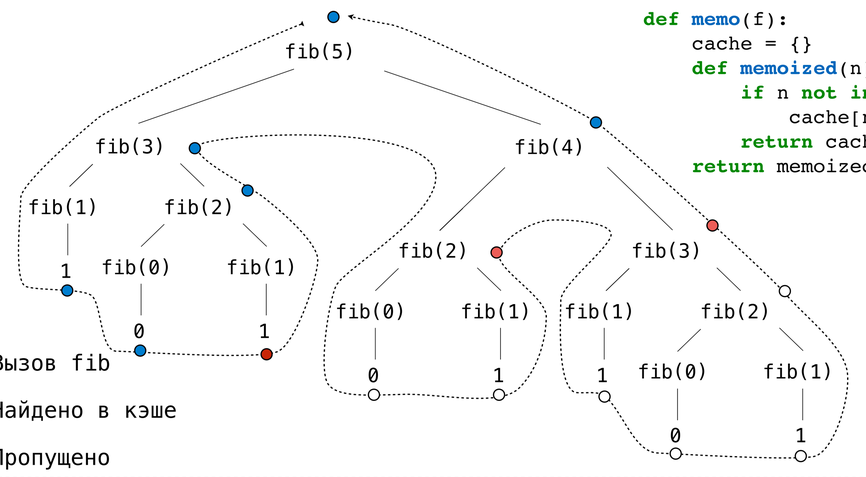
Результат применения **str** к значению выражения соответствует тому, что Python выводит при печати с помощью функции **print**:

```
>>> print(half)
1/2
```

Полиморфная функция: функция, которая способна обрабатывать множество (поли-) различных форм (морф) данных.

```
>>> half.__repr__()
'Fraction(1, 2)'
>>> half.__str__()
'1/2'
```

Функции **str** и **repr** полиморфны; они применимы к любому объекту.




```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Полиморфная функция может принимать два и более аргументов разного типа.

Диспетчеризация типов: определяется тип аргумента и выбирается поведение.

Приведение типов: преобразование значения к другому типу

```
>>> Ratio(1, 3) + 1
Ratio(4, 3)
```

```
>>> 1 + Ratio(1, 3)
Ratio(4, 3)
```

```
>>> from math import pi
>>> Ratio(1, 3) + pi
3.4749259869231266
```

$R(n) = \Theta(f(n))$ означает, что существуют положительные константы k_1 и k_2 такие, что $k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$ для всех **n** больших некоторого минимума **m**

$\Theta(b^n)$ Экспоненциальный рост. Рекурсивный fib

$\Theta(\phi^n)$ шагов, где $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Увеличение размера задачи умножает R(n) на множитель

$\Theta(n^2)$ Квадратичный рост. Например, overlap

Увеличение n увеличивает R(n) на размер задачи n

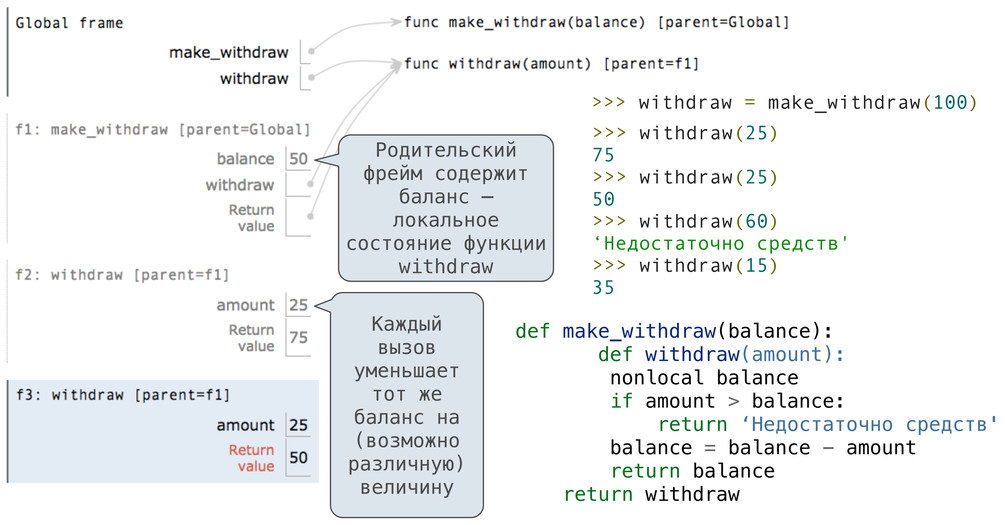
$\Theta(n)$ Линейный рост. Например, медленный factors или exp

$\Theta(\sqrt{n})$ Рост корень из n. Например, factors_fast

$\Theta(\log n)$ Логарифмический рост. Например, exp_fast

Удвоение размера задачи изменяет R(n) линейно.

$\Theta(1)$ Константный рост. Размер задачи не имеет значения.



Родительский фрейм содержит баланс — локальное состояние функции withdraw

Каждый вызов уменьшает тот же баланс на (возможно различную) величину

```
>>> withdraw = make_withdraw(100)
>>> withdraw(25)
75
>>> withdraw(25)
50
>>> withdraw(60)
'Недостаточно средств'
>>> withdraw(15)
35
```

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Недостаточно средств'
        balance = balance - amount
        return balance
    return withdraw
```

```
def make_withdraw_list(balance):
    b = [balance]
    def withdraw(amount):
        if amount > b[0]:
            return 'Insufficient funds'
        b[0] = b[0] - amount
        return b[0]
    return withdraw

withdraw = make_withdraw_list(100)
withdraw(25)
```

```
>>> a = [10]
>>> b = a
>>> a == b
True
>>> a.append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

```
>>> a = [10]
>>> b = [10]
>>> a == b
True
>>> b.append(20)
>>> a
[10]
>>> b
[10, 20]
>>> a == b
False
```

Идентичность

`<exp0> is <exp1>`

вернёт True если результаты обоих выражений <exp0> и <exp1> укажут на один объект

Равенство

`<exp0> == <exp1>`

вернет True если результаты обоих выражений <exp0> и <exp1> будут равными значениями

Идентичные объекты всегда равны

```
suits = ['монеты', 'шнурки', 'мириады', 'десятки']  # Список строк
original_suits = suits
suits.pop()                 # Выталкивает последний элемент
suits.pop()
suits.remove('шнурки')      # Удаляет первый элемент равный аргументу
suits.append('кубки')               # Добавляет элемент в конец
suits.extend(['мечи', 'трефы'])     # Добавляет все элементы в конец
suits[2] = 'пики'  # Заменяет элемент
suits
suits[0:2] = ['червы', 'бубны']  # Заменяет срез
[suit.upper() for suit in suits]
[suit[1:4] for suit in suits if len(suit) == 5]"
```

Константы: множители не влияют на порядок роста процесса

$\Theta(n)$ $\Theta(500 \cdot n)$ $\Theta(\frac{1}{500} \cdot n)$

Логарифмы: основания логарифма не влияют на порядок роста процесса

$\Theta(\log_2 n)$ $\Theta(\log_{10} n)$ $\Theta(\ln n)$

Вложения: когда внутренний процесс повторяется на каждом шаге внешнего процесса, общее число шагов равно произведению количеств шагов внешнего и внутреннего процессов

```
def overlap(a, b):
    count = 0
    for item in a:
        for item in b:
            count += 1
    return count
```

Внешний: длина a

Внутренний: длина b

Если a и b оба имеют длину **n**, то overlap потребует $\Theta(n^2)$ шагов

Члены низкого порядка: Быстрорастущая часть вычисления в итоге доминирует

$\Theta(n^2)$ $\Theta(n^2 + n)$ $\Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$

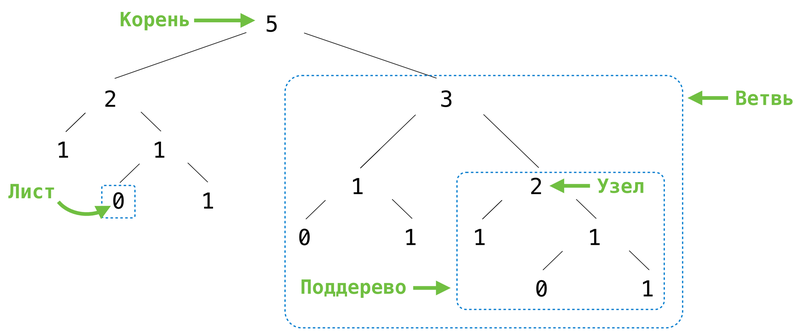
`x = 2`

Состояние	Результат
•Нет инструкции nonlocal •«x» **не** связано локально	Создается связь имени «x» с объектом 2 в первом фрейме текущего окружения
•Нет инструкции nonlocal •«x» **связано** локально	Пересвязывание имени «x» с объектом 2 в первом фрейме текущего окружения
•nonlocal x •«x» **связано** в нелокальном фрейме	Пересвязывание «x» с 2 в первом нелокальном фрейме текущего окружения, в котором присутствует «x»
•nonlocal x •«x» **не** связано в нелокальном фрейме	SyntaxError: no binding for nonlocal 'x' found
•nonlocal x •«x» **связано** в нелокальном фрейме •«x» также связано локально	SyntaxError: name 'x' is parameter and nonlocal

Дерево содержит корневое значение и набор ветвей; каждая ветвь — тоже дерево.

Дерево без ветвей называют листом.

Корневые значения поддеревьев корневого дерева часто называют узловыми значениями или узлами.

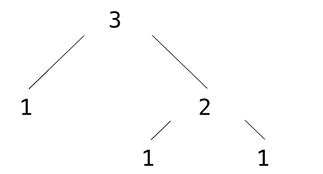



```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    return not branches(tree)
```

Ветви — это деревья!

Список из последовательности ветвей

Проверяет, что дерево сделано на списках




```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

```
def leaves(tree):
    """Возвращает список листьев дерева.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])
```

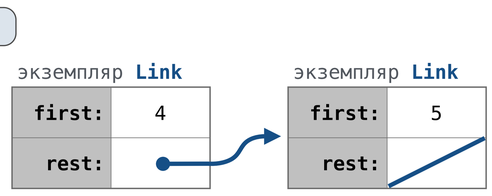
```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

def leaves(tree):
    """Возвращает список листьев дерева."""
    if tree.is_leaf():
        return [tree.label]
    else:
        return sum([leaves(b) for b in tree.branches], [])
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

```
class Link:
    """Связный список."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ', '
            self = self.rest
        return string + str(self.first) + '>'
```

Последовательность нулевой длины




```
>>> s = Link(4, Link(5))
>>> s
Link(4, Link(5))
>>> s.first
4
>>> s.rest
Link(5)
>>> print(s)
<4, 5>
>>> print(s.rest)
<5>
>>> s.rest.rest is Link.empty
True
```

```
>>> s = {'один', 'два', 'три', 'четыре', 'четыре'}
>>> s
{'четыре', 'один', 'три', 'два'}
>>> 'три' in s
True
>>> len(s)
4
>>> s.union({'один', 'пять'})
{'пять', 'три', 'четыре', 'один', 'два'}
>>> s.intersection({'шесть', 'пять', 'четыре', 'три'})
{'четыре', 'три'}
```

Ещё один контейнерный тип в Python

- Множества задаются в фигурных скобках
- Повторяющиеся элементы удаляются на этапе создания
- Элементы множества неупорядочены

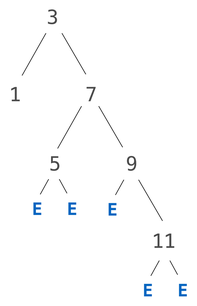
Двоичное дерево — это дерево, у которого есть левая и правая ветвь.

Идея: Заполнять отсутствующие ветви пустыми деревьями.

Идея 2: Экземпляр BinaryTree всегда будет иметь в точности две ветви.

```
class BinaryTree(Tree):
    empty = Tree(None)

    def __init__(self, label, left=empty, right=empty):
        Tree.__init__(self, label, [left, right])

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]
```

```
Bin = BinaryTree
t = Bin(3, Bin(1),
           Bin(7, Bin(5),
                  Bin(9, Bin.empty,
                         Bin(11))))
```



E: Пустое дерево

Идея: Все банковские счета включают баланс `balance` и владельца `holder`; класс `Account` должен добавлять соответствующие атрибуты каждому новому экземпляру.

```
>>> a = Account('Вася')
>>> a.holder
'Вася'
>>> a.balance
0
```

При «вызове» класса:

1. Создается новый экземпляр класса:

Экземпляр класса Account

balance: 0 holder: 'Вася'

2. Методу класса `__init__` при вызове, в качестве первого аргумента передается новый объект (`self`), вместе с ним передаются дополнительные аргументы, указанные в «вызове» класса.

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Недостаточно средств'
        self.balance = self.balance - amount
        return self.balance
```

Метод `__init__` называют «конструктор»

self всегда связан с экземпляром класса Account

Функция: все аргументы в скобочках

```
>>> type(Account.deposit)
<class 'function'>
>>> type(vasya_account.deposit)
<class 'method'>
```

```
>>> Account.deposit(vasya_account, 656)
666
>>> vasya_account.deposit(671)
1337
```

Метод: Один объект до точки и потом аргументы в скобочках

<выражение> . <имя>

<Выражение> может быть любым корректным выражением на Python.

<Имя> должно быть просто именем...

которое возвращает значение атрибута, найденное по <имени> в объекте, который является результатом <выражения>.

При выполнении выражения с точкой:

1. Выполнить <выражение> слева от точки, которое вернёт объект.
2. В атрибутах объекта ищется совпадение <имени>; если атрибут с таким именем существует, то возвращается его значение.
3. В противном случае, <имя> ищется в классе с возвратом значения атрибута.
4. Если это функция, то вместо функции Python вернет связанный метод.

Инструкция присвоения выражению с точкой слева от знака равенства изменяет атрибут объекта указанного в выражении с точкой.

- Если объект — это экземпляр, то присвоение изменяет атрибут экземпляра
- Если объект — это класс, то присвоение изменяет атрибут класса

Атрибуты класса Account

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Атрибуты экземпляра vasya_account

balance: 0
holder: 'Вася'
interest: 0.08

Атрибуты экземпляра petya_account

balance: 0
holder: 'Петя'

```
>>> vasya_account = Account('Вася')
>>> petya_account = Account('Петя')
>>> petya_account.interest
0.02
>>> vasya_account.interest
0.02
>>> Account.interest = 0.04
>>> petya_account.interest
0.04
>>> vasya_account.interest
0.04
```

```
>>> vasya_account.interest = 0.08
>>> vasya_account.interest
0.08
>>> petya_account.interest
0.04
>>> Account.interest = 0.05
>>> petya_account.interest
0.05
>>> vasya_account.interest
0.08
```

Класс `CheckingAccount` — особый вид класса `Account`

```
>>> ch = CheckingAccount('Вася')
>>> ch.interest     # Меньший процент для чекового счета
0.01
>>> ch.deposit(20)  # Пополнение точно такое же
20
>>> ch.withdraw(5)  # Снятие включает дополнительную комиссию в 1 ₽
14
```

Почти всё поведение такое же как в базовом классе Account

```
class CheckingAccount(Account):
    """Банковский счет с комиссией за снятие."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
        return super().withdraw(amount + self.withdraw_fee)
```

или

Представь, что перед каждым примером выполняется вот что:

```
s = [2, 3]
t = [5, 6]
```

Действие	Пример	Результат
append добавляет элемент в список	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
extend добавляет все элементы одного списка в другой	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
Срезы и сложение создают новые списки содержащие существующие элементы	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]
Функция **list** создаёт новый список, содержащий исходные элементы	t = list(s) s[1] = 0	s → [2, 0] t → [2, 3]
Присвоение срезу заменяет срез новыми значениями	s[0:0] = t s[3:] = t t[1] = 0	s → [5, 6, 2, 5, 6] t → [5, 0]