Аргументы

Основы передачи аргументов

- Аргументы передаются путем автоматического присваивания объектов именам локальных переменных. Аргументы функций, т.е. ссылки на (возможно) разде ляемые объекты, отправленные вызывающим кодом, представляют собой еще один пример присваивания Python в действии. Поскольку ссылки реализованы в виде указателей, все аргументы в действительности передаются через указатели. Объекты, передаваемые как аргументы, никогда автоматически не копируются.

- Присваивание именам аргументов внутри функции не затрагивает вызывающий код. Когда функция выполняется, имена аргументов в заголовке функции становятся новыми локальными именами в области видимости функции. Никакого совмещения имен аргументов функции и имен переменных в области видимости вызывающего кода не происходит.

Основы передачи аргументов

- Модификация внутри функции аргумента, являющегося изменяемым объектом, может затронуть вызывающий код. с другой стороны, так как аргументам прос то присваиваются передаваемые объекты, в функциях можно модифицировать переданные изменяемые объекты на месте, в результате оказывая влияние на вызывающий код. Изменяемые аргументы могут служить входными и выходными данными для функций.

Основы передачи аргументов

- Схема передачи по присваиванию Python – не совсем то же самое, что и ссылоч ные параметры C++, но на практике она очень похожа на модель передачи аргументов в C (и других языках):

- Неизменяемые аргументы фактически передаются «по значению». Объекты, по добные целым числам и строкам, передаются по ссылке на объекты, а не путем копирования, но из-за того, что модифицировать на месте неизменяемые объек ты невозможно, эффект во многом похож на создание копий.

- Изменяемые аргументы фактически передаются «по указателю». Объекты вро де списков и словарей также передаются по ссылке на объекты, что аналогич но способу передачи массивов как указателей в языке C – изменяемые объекты можно модифицировать на месте в функции почти как массивы в C.

Основы передачи аргументов

```
>>> def f(a):     # a присваивается (устанавливается в ссылку на) переданный объект
        a = 99     # Изменяет только локальную переменную a

>>> b = 88
>>> f(b)          # a и b первоначально ссылаются на 88
>>> print(b)      # Значение b не изменилось
88


>>> def changer(a, b):     # Аргументам присваиваются ссылки на объекты
        a = 2               # Изменяет только значение локального имени
        b[0] = 'spam'       # Изменяет разделяемый объект на месте
>>> X = 1
>>> L = [1, 2]             # Вызывающий код:
>>> changer(X, L)          # Передача неизменяемого и изменяемого объектов
>>> X, L                   # X остается прежним, L отличается!
(1, ['spam', 2])
```

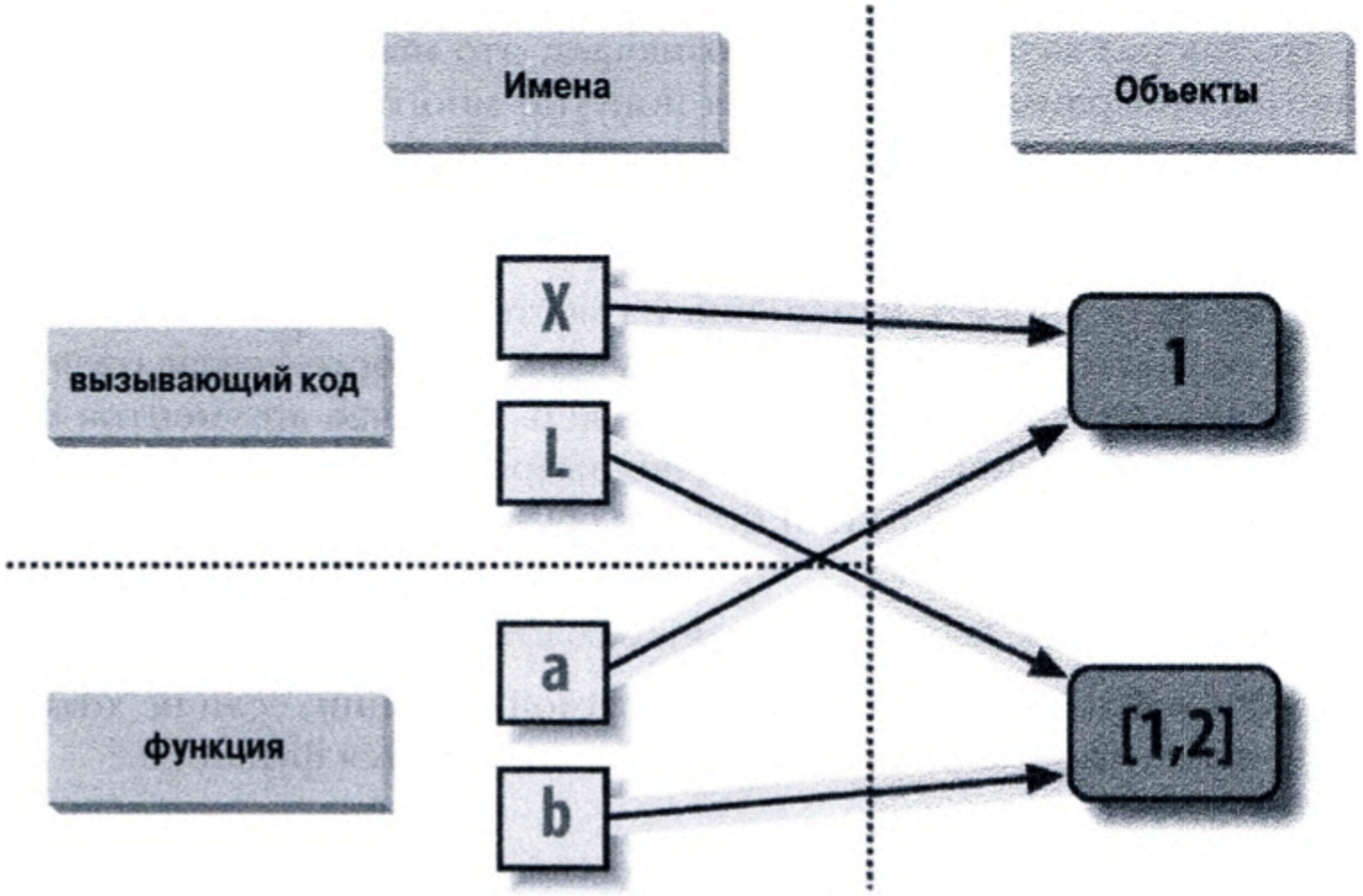
Основы передачи аргументов

- Функция changer присваивает значения самому аргументу a и компоненту *объекта*, на который ссылается аргумент b. Эти два присваивания внутри функции лишь слегка отличаются по синтаксису, но приводят к совершенно разным результатам.

- Поскольку a – имя локальной переменной в области видимости функции, первое присваивание не влияет на вызывающий код; оно просто модифицирует локальную переменную a, чтобы a ссылалась на другой объект, и не изменяет привязку имени X в области видимости вызывающего кода. Ситуация такая же, как в предыдущем примере.

- Аргумент b тоже является именем локальной переменной, но ему передан изменяемый объект (список, на который ссылается L в области видимости вызывающего кода). Так как второе присваивание представляет собой изменение объекта на месте, результат присваивания b[0] в функции оказывает воздействие на значение L после возврата управления из функции.

Основы передачи аргументов



Основы передачи аргументов

```
>>> X = 1
>>> a = X    # Разделяют тот же самый объект
>>> a = 2    # Переустанавливает только a, значением X по-прежнему будет 1
>>> print(X)
1

>>> L = [1, 2]
>>> b = L              # Разделяют тот же самый объект
>>> b[0] = 'spam'      # Изменение на месте: L тоже увидит изменение
>>> print(L)
['spam', 2]
```

Основы передачи аргументов

Избегайте модификации изменяемых аргументов

Если вносить изменения на месте внутри функций, воздействуя на передаваемые объекты, нежелательно, тогда можно просто создавать явные копии изменяемых объектов

```
L = [1, 2]
changer(X, L[:])      # Передача копии, так что L не изменяется

def changer(a, b):
    b = b[:]          # Создание копии входного списка,
                      # чтобы не влиять на вызывающий код
    a = 2
    b[0] = 'spam'     # Изменяет только копию списка
```

Основы передачи аргументов

Эмуляция выходных параметров имножественных результатов

```
>>> def multiple(x, y):
        x = 2                       # Изменяет только локальные имена
        y = [3, 4]
        return x, y                 # Возвращение множества новых значений в кортеже

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)   # Присваивание результатов именам в вызывающем коде
>>> X, L
(2, [3, 4])
```

Режимы сопоставления аргументов

Позиционные: сопоставляются слева направо

Нормальный сценарий, который мы главным образом использовали до сих пор, предусматривает сопоставление переданных значений аргументов с именами аргументов в заголовке функции по позиции, слева направо.

Ключевые: сопоставляются по имени аргумента

В качестве альтернативы в вызывающем коде можно указывать, какой аргумент функции получает значение, за счет применения имени аргумента в вызове посредством синтаксиса *имя=значение*.

Стандартные: указывают значения для необязательных аргументов, которым значения не передавались

Сами функции могут задавать стандартные значения для аргументов, которые они получат, если в вызове передается слишком мало значений, снова с использованием синтаксиса *имя-значение*.

Режимы сопоставления аргументов

Сбор переменного количества аргументов: собирает произвольно много позиционных и ключевых аргументов

В функциях могут применяться специальные аргументы, предваренные одним или двумя символами ,* для сбора произвольного количества возможных добавочных аргументов. Такую возможность часто называют *переменным, количеством аргументов (varargs)* в честь списка аргументов переменной длины в языке C; в Python аргументы собираются в нормальный объект.

Распаковка переменного количества аргументов:
передает произвольно много позиционных и ключевых аргументов

В вызывающем коде синтаксис * можно также использовать для распаковки коллекций аргументов в отдельные аргументы. Это противоположность * в за головке функции — синтаксис * в заголовке означает сбор произвольно большо го числа аргументов, тогда как в вызове он означает распаковку произвольно большого количества аргументов и их передачу по отдельности как обособлен ных значений.

Режимы сопоставления аргументов

Аргументы с передачей только по ключевым словам: аргументы, которые должны передаваться по имени

В Python 3.X (но не в Python 2.X) функции также допускают указание аргумен тов, которые должны передаваться по имени с помощью ключевых аргументов, а не по позиции. Такие аргументы обычно применяются для определения кон фигурационных параметров в дополнение к фактическим аргументам.

Режимы сопоставления аргументов

Синтаксис	Местоположение	Интерпретация
`func(значение)`	Вызывающий код	Нормальный аргумент: сопоставляется по позиции
`func(имя=значение)`	Вызывающий код	Ключевой аргумент: сопоставляется по имени
`func(*итерируемый_объект)`	Вызывающий код	Передает все объекты в *итерируемом_объекте* как отдельные позиционные аргументы
`func(**словарь)`	Вызывающий код	Передает все пары ключ/значение в *словаре* как отдельные ключевые аргументы
`def func(имя)`	Функция	Нормальный аргумент: сопоставляется с любым переданным значением по позиции или по имени
`def func(имя=значение)`	Функция	Стандартное значение аргумента, если значение в вызове не передавалось
`def func(*имя)`	Функция	Сопоставляет и собирает оставшиеся позиционные аргументы в кортеж
`def func(**имя)`	Функция	Сопоставляет и собирает оставшиеся ключевые аргументы в словарь
`def func(*остальные, имя)`	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову (Python 3.X)
`def func(*,имя=значение)`	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову (Python 3.X)

Режимы сопоставления аргументов

Если вы решили использовать и комбинировать специальные режимы сопоставления аргументов, тогда Python потребует соблюдать описанные далее правила упорядочения для необязательных компонентов.

- В вызове функции аргументы должны указываться в следующем порядке: любые позиционные аргументы (значение), за ними комбинация любых ключевых аргументов (имя=значение) и формы *итерируемый_объект, а затем форма **словарь.

- В заголовке функции аргументы должны указываться в следующем порядке: любые нормальные аргументы (имя), за ними любые стандартные аргументы (имя=значение), далее форма *имя (или * в Python 3.X), затем аргументы с передачей только по ключевым словам имя или имя=значение (в Python 3.X) и, наконец, форма **имя.

Режимы сопоставления аргументов

Шаги, внутренне выполняемые Python для сопоставления аргументов перед присваиванием, могут быть грубо описаны следующим образом:

1. Присваивание неключевых аргументов по позиции.
2. Присваивание ключевых аргументов по совпадающим именам.
3. Присваивание добавочных неключевых аргументов кортежу *имя.
4. Присваивание добавочных ключевых аргументов словарю *и*мя.
5. Присваивание стандартных значений неприсвоенным аргументам в заголовке.

Режимы сопоставления аргументов

```
>>> def f(a, b=2, c=3): print(a, b, c)     # Аргумент a обязательный,
                                           # b и c необязательные


>>> f(1)          # Использование стандартных значений
1 2 3
>>> f(a=1)
1 2 3
>>> f(1, 4)       # Переопределение стандартных значений
1 4 3
>>> f(1, 4, 5)
1 4 5

 >>> f(1, c=6)          # Выбор стандартных значений
 1 2 6
```

Режимы сопоставления аргументов

```
def func(spam, eggs, toast=0, ham=0):   # Первые два аргумента обязательны
    print((spam, eggs, toast, ham))

func(1, 2)                              # Вывод: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                  # Вывод: (1, 0, 0, 1)
func(spam=1, eggs=0)                    # Вывод: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)           # Вывод: (3, 2, 1, 0)
func(1, 2, 3, 4)                        # Вывод: (1, 2, 3, 4)
```

Произвольное количество аргументов

```
>>> def f(*args): print(args)

>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
>>> def f(**args): print(args)

>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}

>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Распаковка аргументов

```
>>> def func(a, b, c, d): print(a, b, c, d)

>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)       # То же, что и func(1, 2, 3, 4)
1 2 3 4
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)      # То же, что и func(a=1, b=2, c=3, d=4)
1 2 3 4

>>> func(*(1, 2), **{'d': 4, 'c': 3})   # То же, что и func(1, 2, d=4, c=3)
1 2 3 4
>>> func(1, *(2, 3), **{'d': 4})        # То же, что и func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, c=3, *(2,), **{'d': 4})     # То же, что и func(1, 2, c=3, d=4)
1 2 3 4
>>> func(1, *(2, 3), d=4)               # То же, что и func(1, 2, 3, d=4)
1 2 3 4
>>> func(1, *(2,), c=3, **{'d':4})      # То же, что и func(1, 2, c=3, d=4)
1 2 3 4
```

Функция min

- необходимо написать функцию, которая способна находить минимальное значение в произвольном наборе аргументов и произвольном наборе типов данных. То есть функция обязана принимать ноль и более аргументов — столь ко, сколько передается.

- Кроме того, функция должна работать со всеми типами объектов Python: числами, строками, списками, списками словарей, файлами и даже `None`.

Функция min

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res
def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)    # Или в Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print(min1(3, 4, 1, 2))
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

Концепции проектирования функций

- Связность: используйте аргументы для входных данных и оператор return для выходных данных. Как правило, вы должны стремиться сделать функцию независимой от вещей, находящихся за ее пределами. Аргументы и оператор return часто будут наилучшими способами изоляции внешних зависимостей небольшим количеством хорошо известных мест в коде.

- Связность: применяйте глобальные переменные, только когда они понастоящему нужны. Глобальные переменные (т.е. имена во включающем модуле) обычно являются неудачным способом взаимодействия для функций. Они могут создать зависимости и проблемы синхронизации, которые затрудняют отладку, измене ние и многократное использование программ.

Концепции проектирования функций

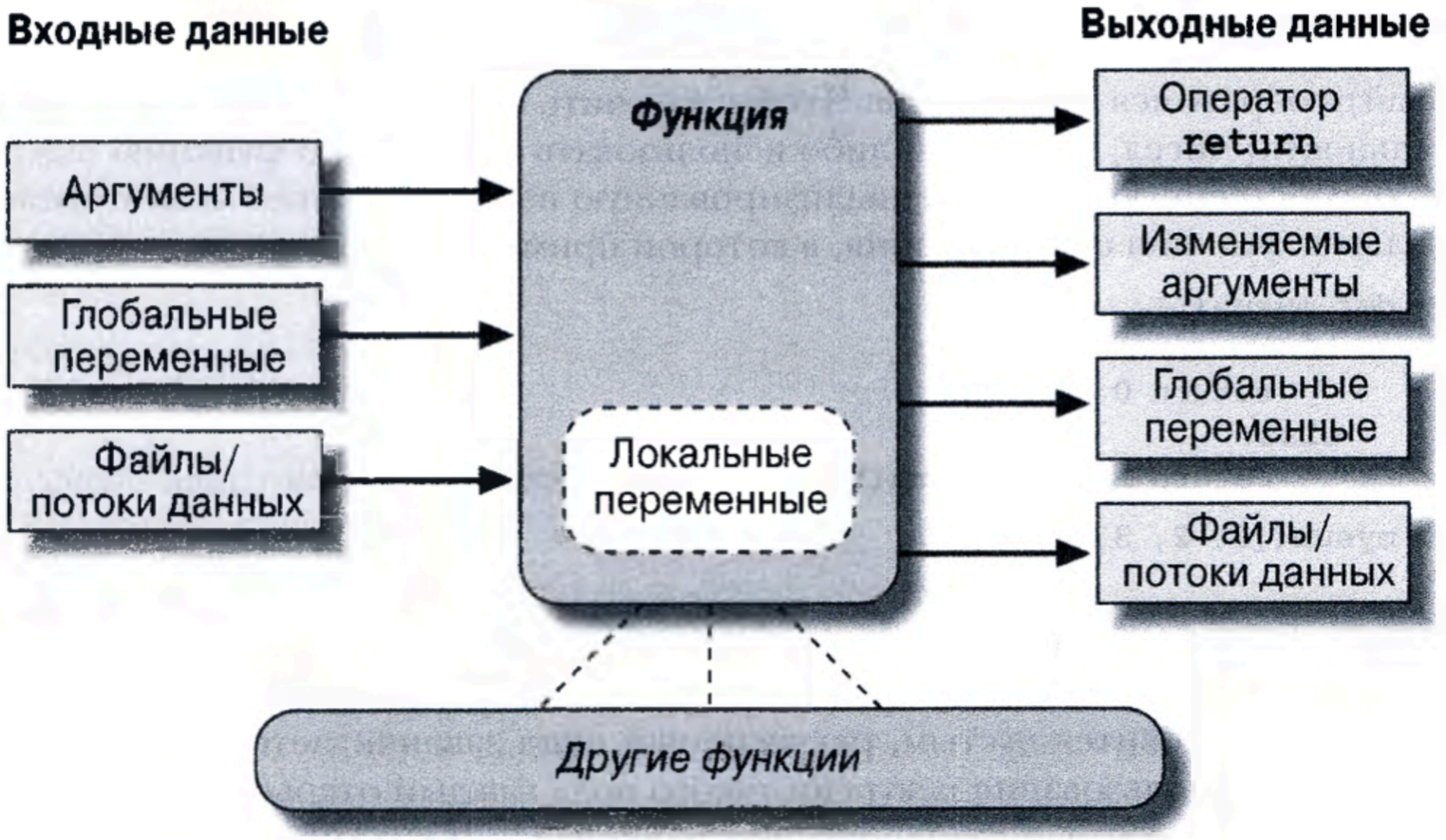
- Связность: не модифицируйте изменяемые аргументы, если только такое изменение не ожидается вызывающим кодом. Функции могут модифицировать части передаваемых изменяемых объектов, но это (как и глобальные переменные) создает сильную связность между вызывающим и вызываемым кодом, которая может сделать функцию слишком специфичной и хрупкой.

- Сцепление: каждая функция должна иметь единственное унифицированное назначение. При надлежащем проектировании каждая функция должна делать что-то одно - то, что может быть резюмировано в простом повествовательном предложении. Если такое предложение оказывается слишком широким (например, “данная функция реализует всю мою программу") или содержит много союзов (скажем, “данная функция дает сотруднику повышение и отправляет заказ пиццы”), тогда имеет смысл подумать о разбиении функции на несколько отдельных более простых функций. В противном случае не удастся повторно применять код, который лежит в основе действий, смешанных в функции.

- Размер: каждая функция должна быть относительно небольшой. Цель естественным образом следует из предыдущей цели, но если ваши функции начали занимать несколько страниц на экране редактора, то видимо пришло время их раз бить. Прежде всего, учитывая присущую коду Python лаконичность, длинная или глубоко вложенная функция зачастую служит признаком проблем с проектным решением. Сохраняйте функции простыми и короткими.

- Связность: избегайте прямого изменения переменных из другого файла модуля. Тем не менее, для справки помните о том, что изменение переменных через границы файлов приводит к появлению связности между модулями подобно тому, как глобальные переменные связывают функции — модули становятся труднее понимать и многократно использовать. При любой возможности применяйте функции доступа вместо прямых операторов присваивания.

Концепции проектирования функций



Рекурсивные функции

Суммирование с помощью рекурсии

```
>>> def mysum(L):
        if not L:
            return 0
        else:
            return L[0] + mysum(L[1:])    # Рекурсивный вызов самой себя

>>> mysum([1, 2, 3, 4, 5])
15
>>> def mysum(L):
        print(L)     # Трассировка уровней рекурсии
        if not L:    # На каждом уровне L становится короче
            return 0
        else:
            return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Рекурсивные функции

Суммирование с помощью рекурсии

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])          # Использование
                                                        # тернарного выражения

def mysum(L):
  return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Любой тип,
                            # предполагая наличие хотя бы одного элемента

def mysum(L):
  first, *rest = L
  return first if not rest else first + mysum(rest) # Применение расширенного
                                    # присваивания последовательностей Python 3.X
```

Рекурсивные функции

Суммирование с помощью рекурсии

```
>>> mysum([1])           # mysum([]) терпит неудачу в последних двух версиях
1
>>> mysum([1, 2, 3, 4, 5])
15
>>> mysum(('s', 'p', 'a', 'm'))    # Но теперь разрешены разнообразные типы
'spam'
>>> mysum(['spam', 'ham', 'eggs'])
'spamhameggs'
```

```
>>> def mysum(L):
        if not L: return 0
        return nonempty(L)    # Вызов функции nonempty, которая вызывает mysum
>>> def nonempty(L):
        return L[0] + mysum(L[1:])    # Косвенная рекурсия
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```