при поддержке
{а}вижн

Лекция 22

Перевод и адаптация материалов Джона ДеНиро (John DeNero). Используется с разрешения автора.

Списки

Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

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

Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

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

Действие

Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

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

Действие	Пример

Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

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

Действие	Пример	Результат

Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

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

Действие	Пример	Результат
append добавляет элемент в список		

Списки на диаграммах окружения

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`	

Списки на диаграммах окружения

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`

Списки на диаграммах окружения

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		

Списки на диаграммах окружения

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`	

Списки на диаграммах окружения

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]`

Списки на диаграммах окружения

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		

Списки на диаграммах окружения

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		

Списки на диаграммах окружения

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		

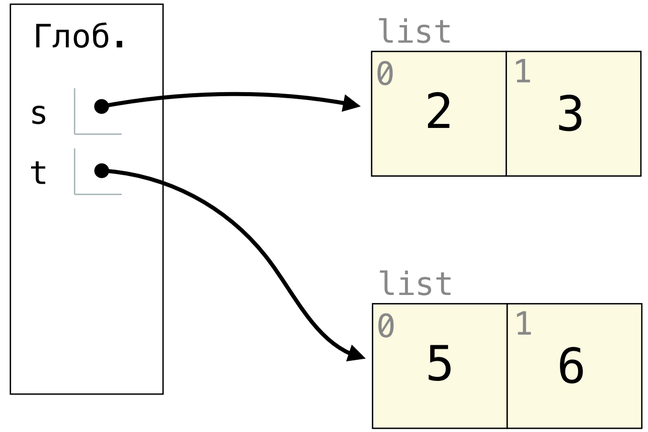
Глоб.

Списки на диаграммах окружения

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)`	`s → [2, 3, [5, 6]]`
	`t = 0`	`t → 0`
extend добавляет все элементы одного списка в другой	`s.extend(t)`	`s → [2, 3, 5, 6]`
	`t[1] = 0`	`t → [5, 0]`
Срезы и сложение создают новые списки содержащие существующие элементы		

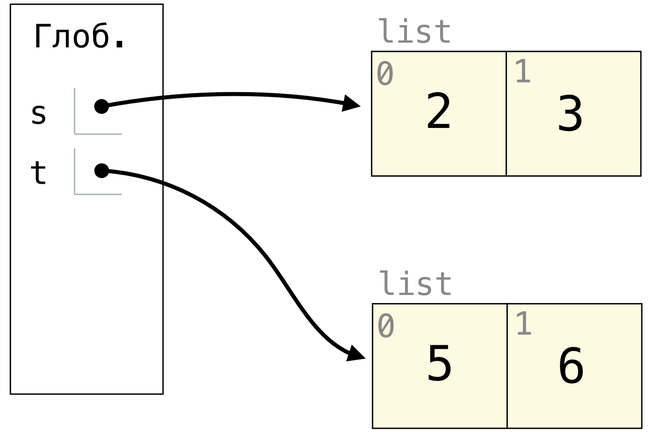


Списки на диаграммах окружения

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]`	

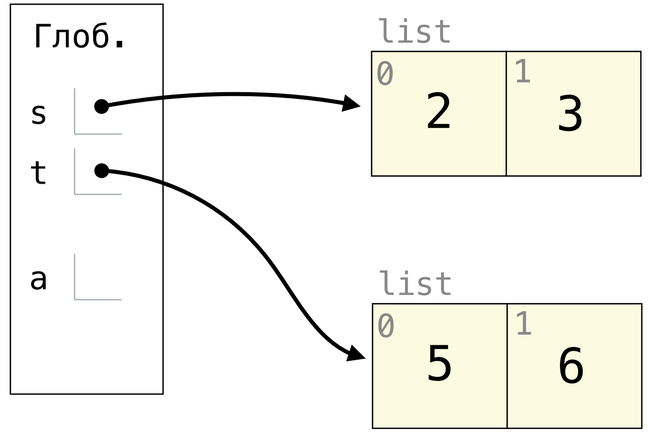


Списки на диаграммах окружения

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]`	

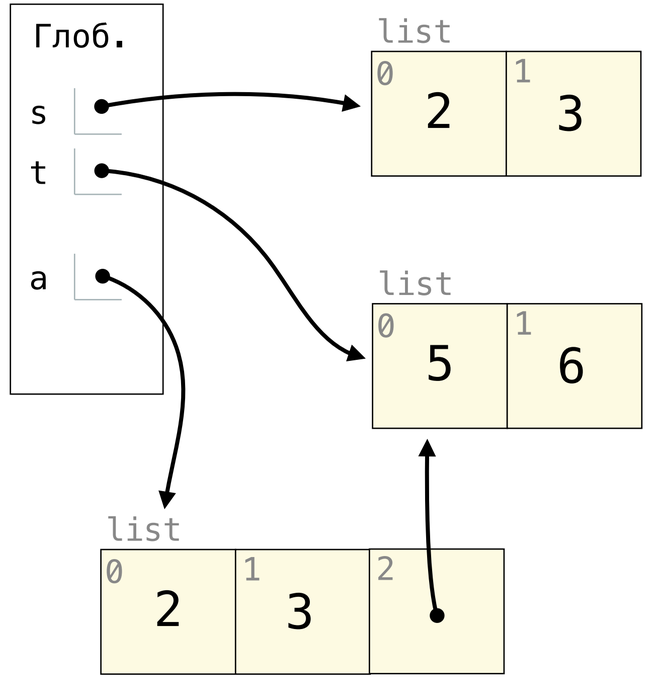


Списки на диаграммах окружения

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]`	

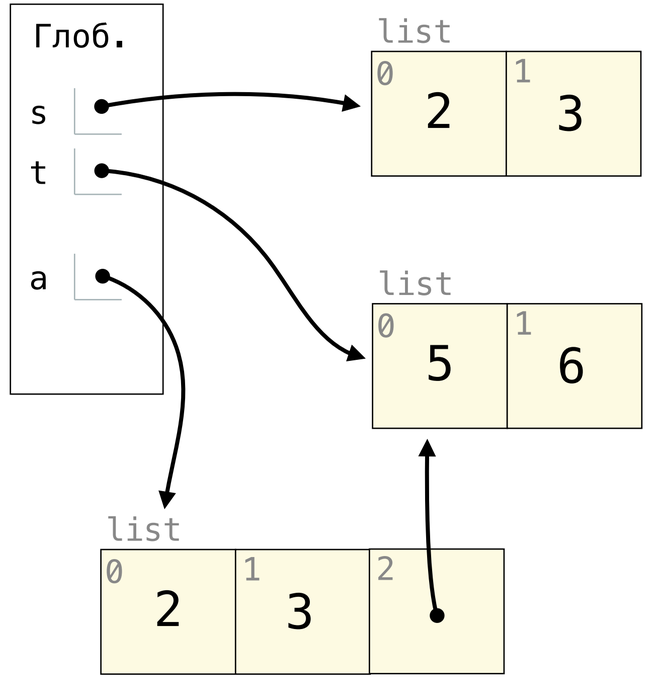


Списки на диаграммах окружения

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:]`	

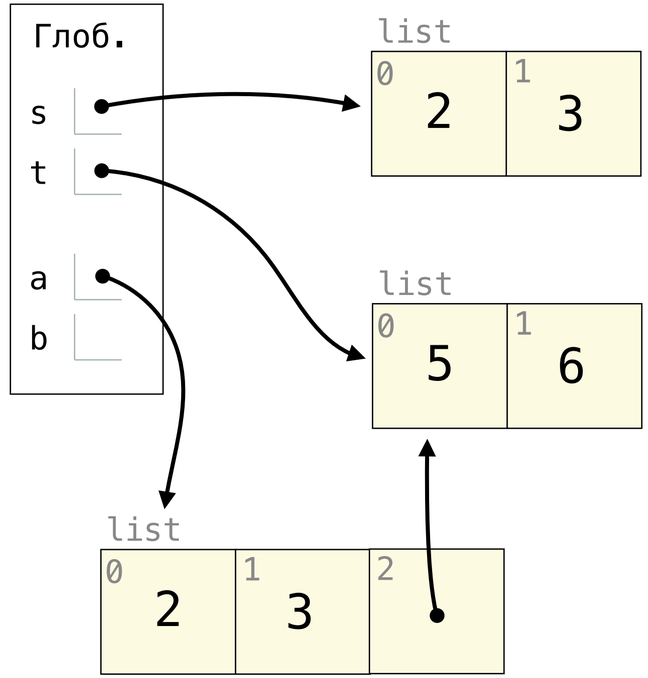


Списки на диаграммах окружения

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:]`	

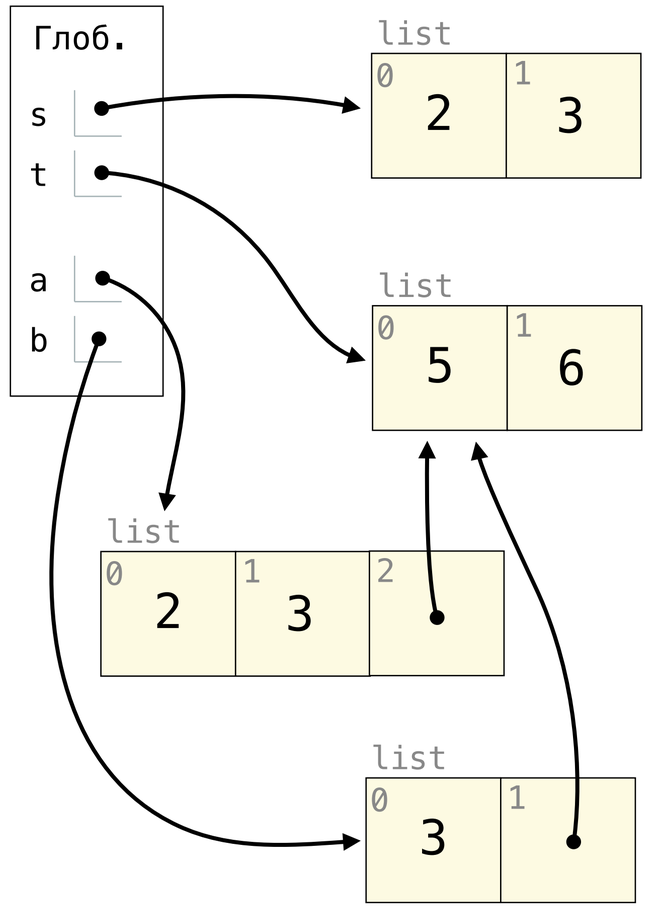


Списки на диаграммах окружения

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:]`	



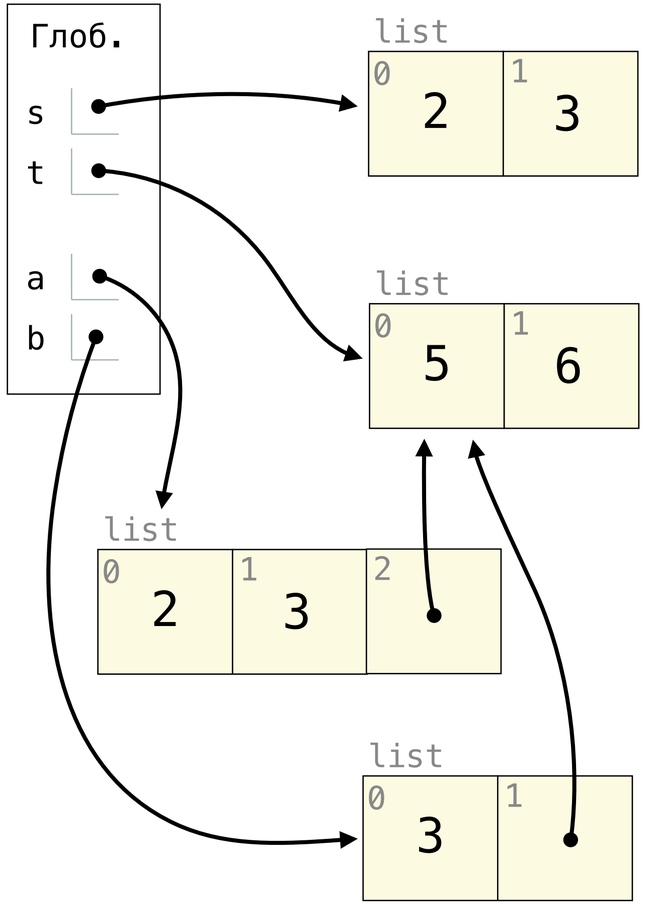
Списки на диаграммах окружения

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`	



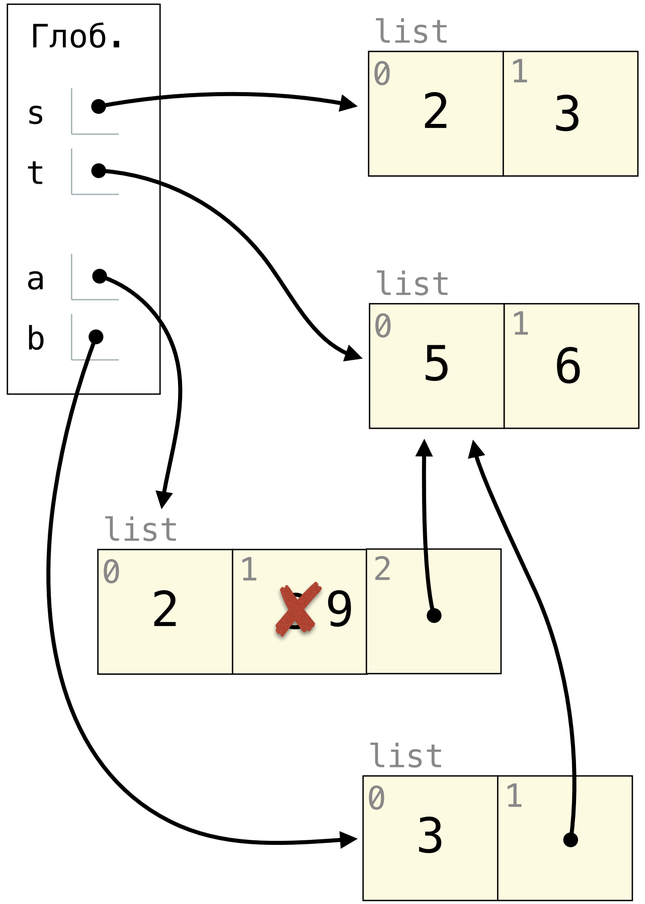
Списки на диаграммах окружения

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`	



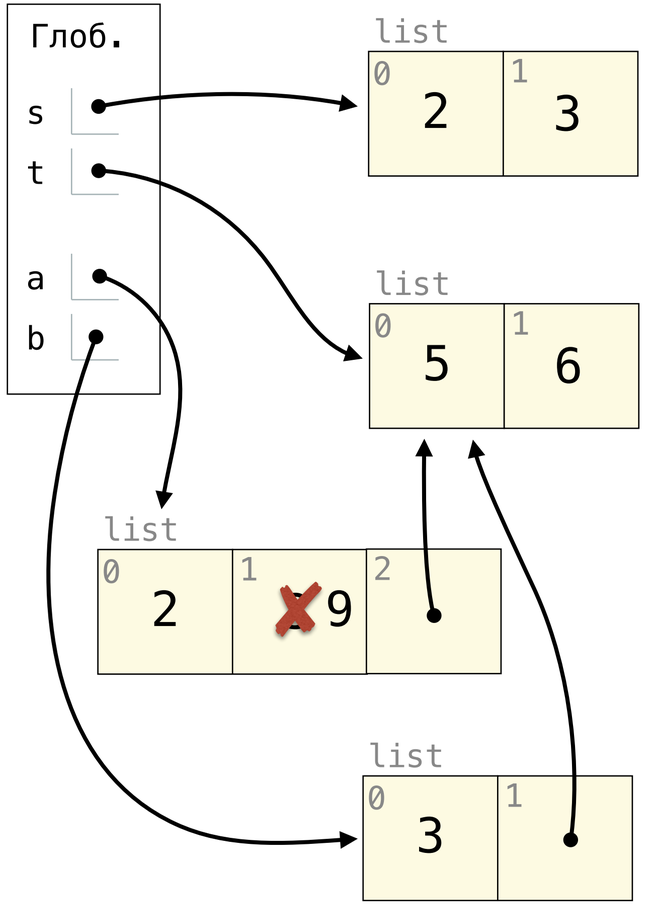
Списки на диаграммах окружения

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`	



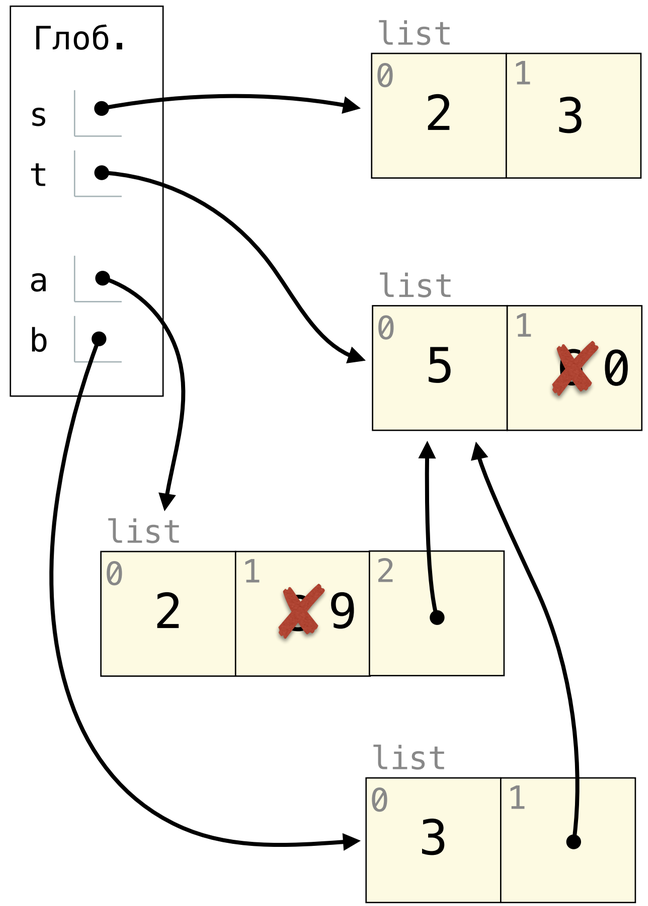
Списки на диаграммах окружения

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`	



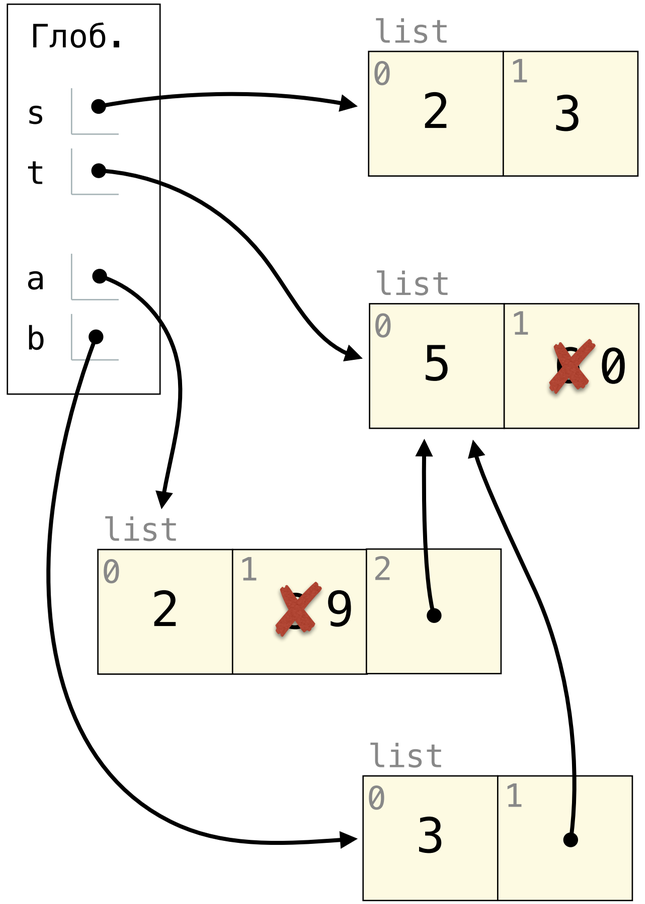
Списки на диаграммах окружения

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]]`



Списки на диаграммах окружения

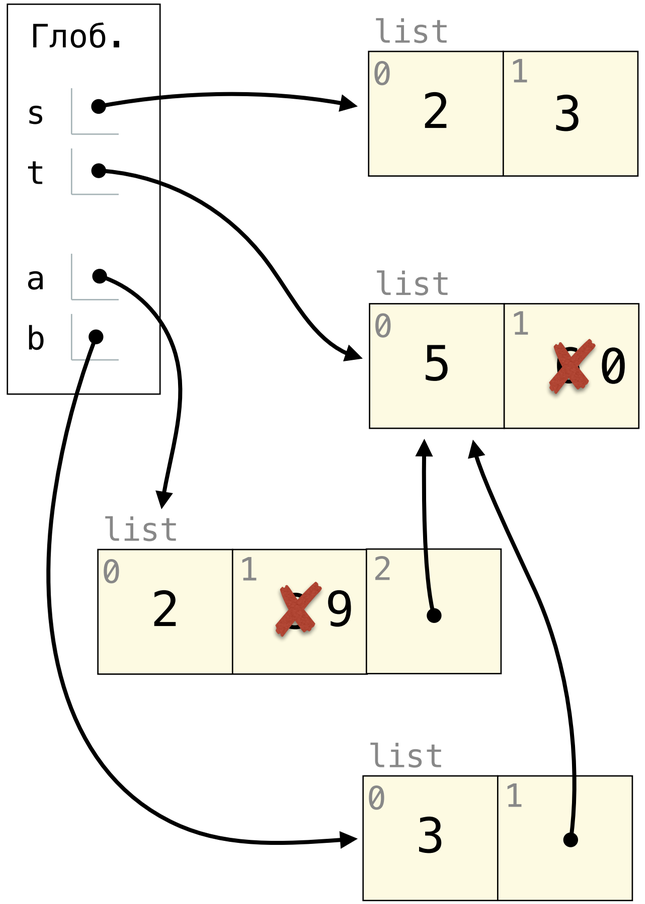
Представь, что перед каждым примером выполняется вот что:

```
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		



Списки на диаграммах окружения

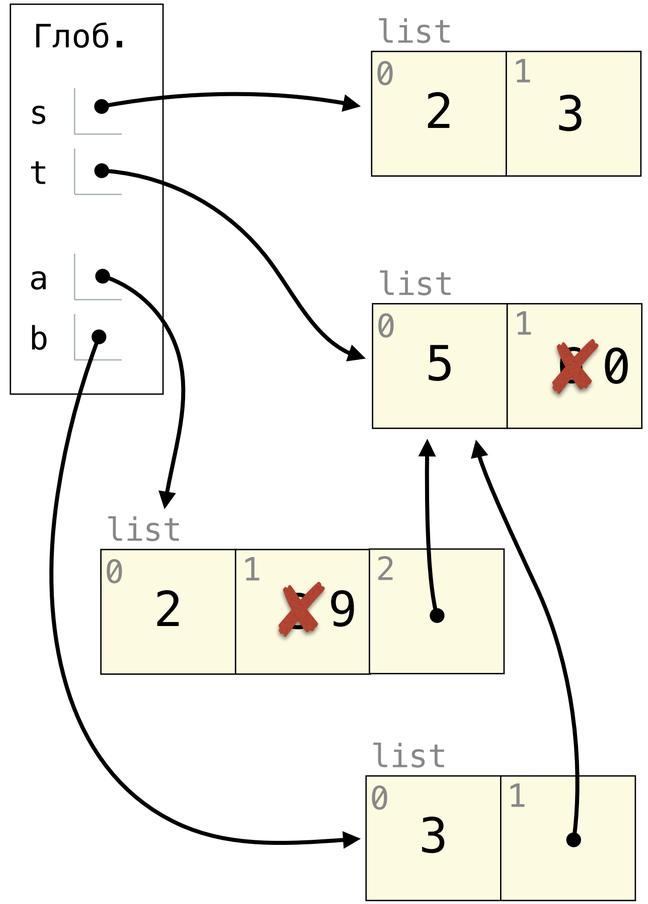
Представь, что перед каждым примером выполняется вот что:

```
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`	



Списки на диаграммах окружения

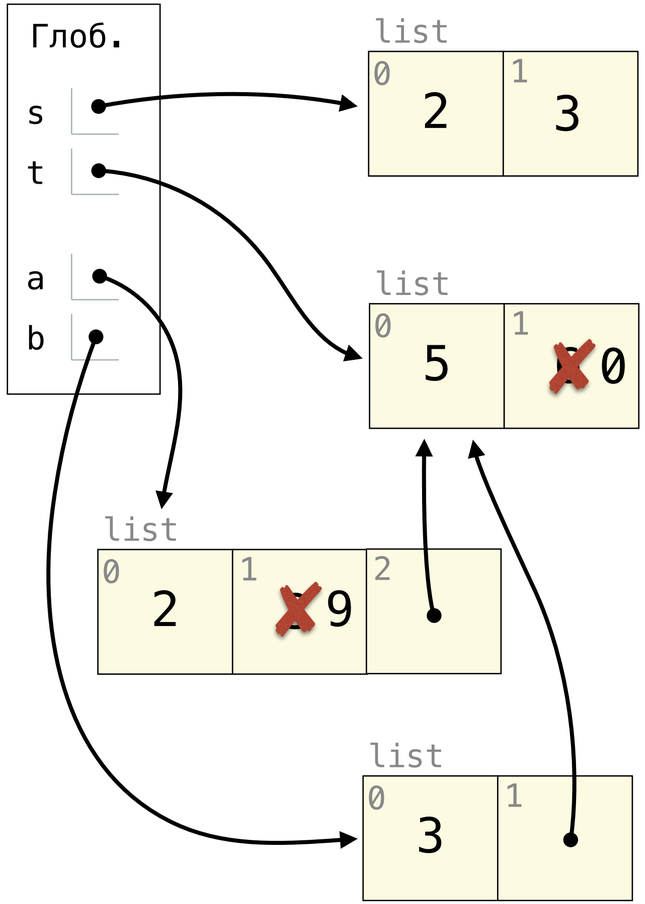
Представь, что перед каждым примером выполняется вот что:

```
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]`



Списки на диаграммах окружения

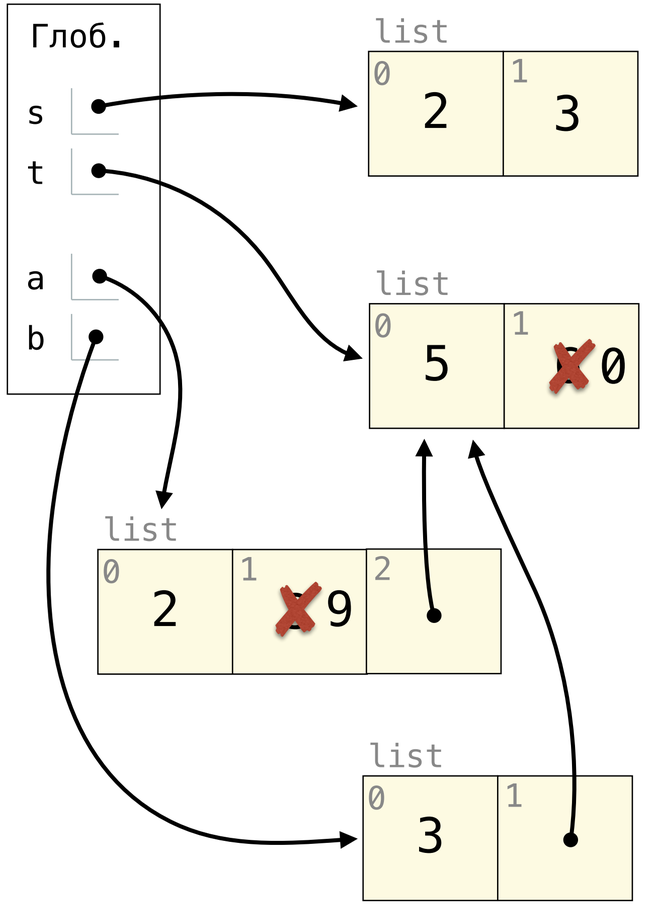
Представь, что перед каждым примером выполняется вот что:

```
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		



Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

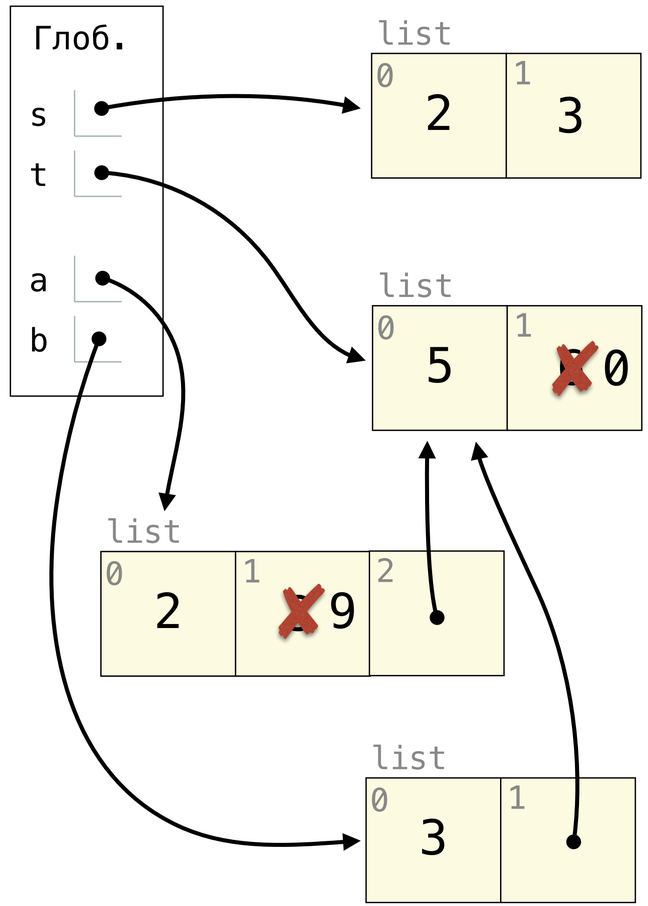
```
s = [2, 3]
t = [5, 6]
```

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`	



Списки на диаграммах окружения

Представь, что перед каждым примером выполняется вот что:

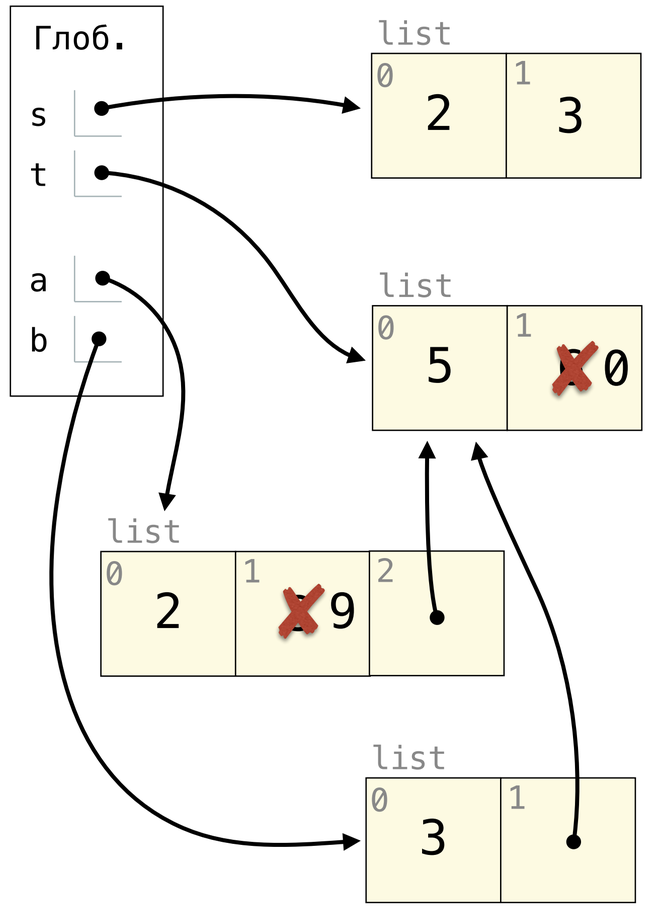
```
s = [2, 3]
t = [5, 6]
```

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]`



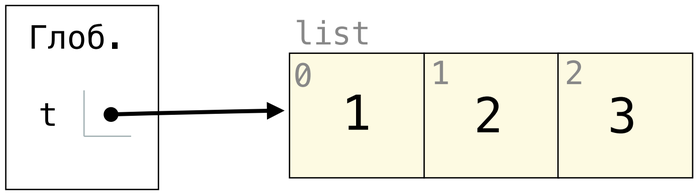
Списки в списках на диаграммах окружения

Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
```

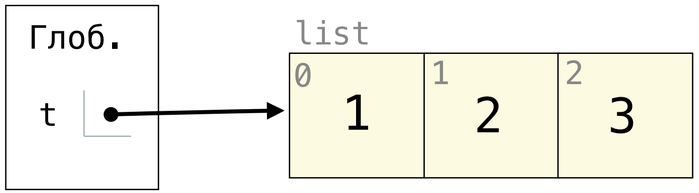
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
```



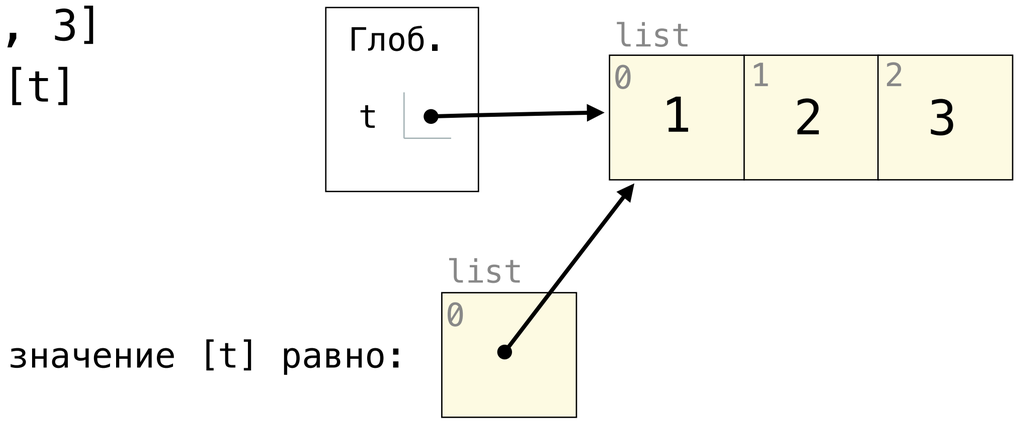
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
```



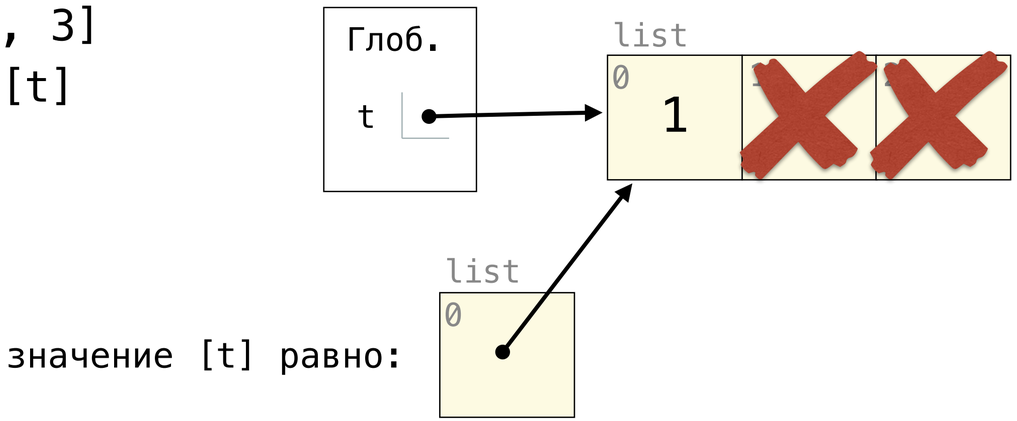
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
```



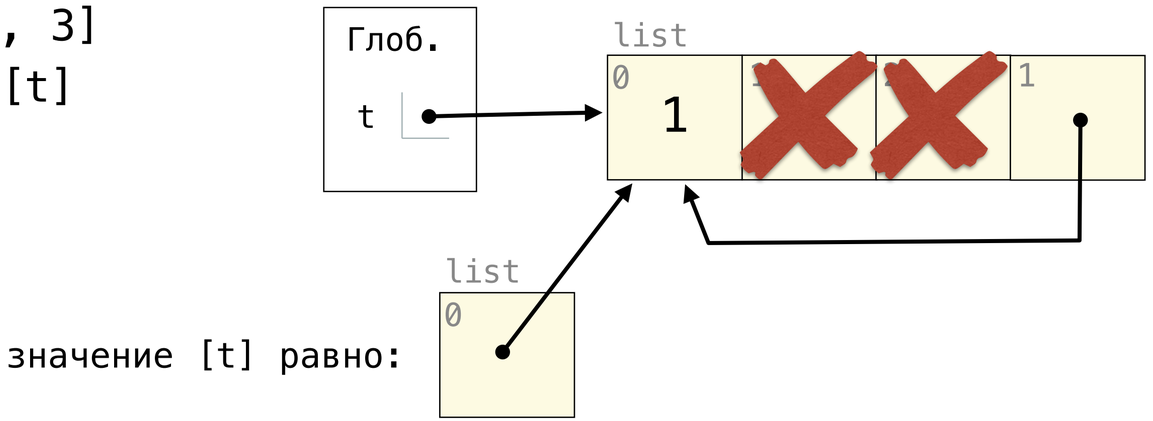
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
```



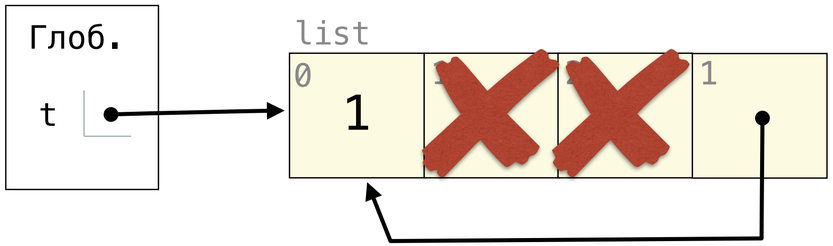
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
```



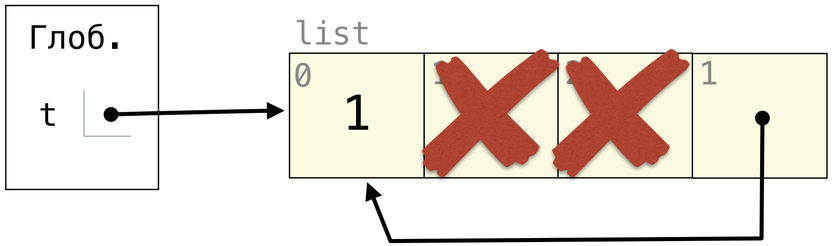
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
```



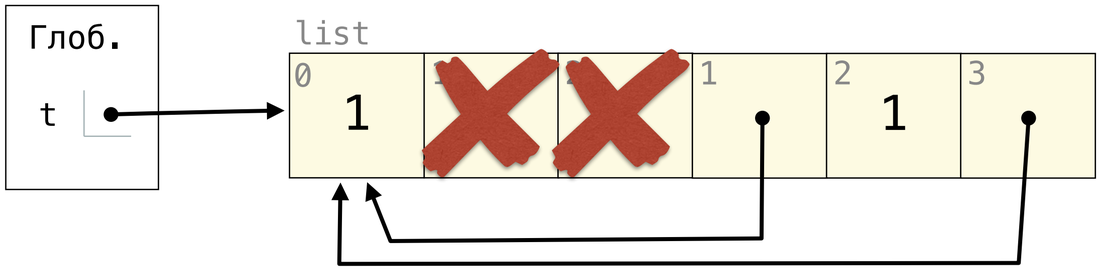
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



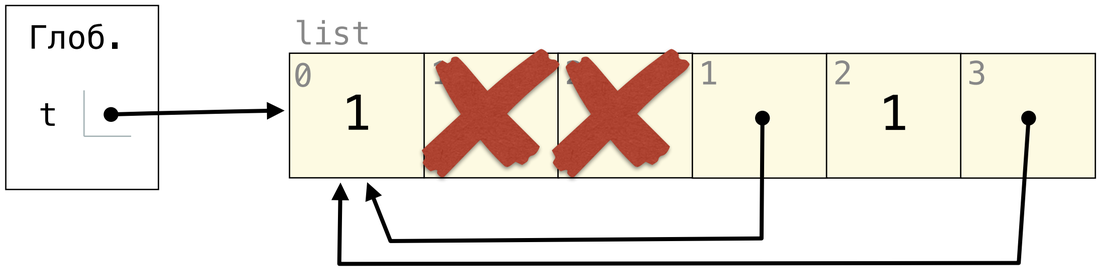
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



Списки в списках на диаграммах окружения

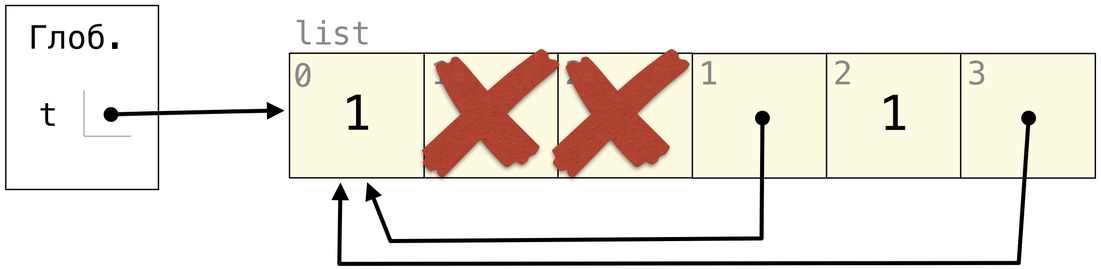
```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```




```
[1, [...], 1, [...]]
```

Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```

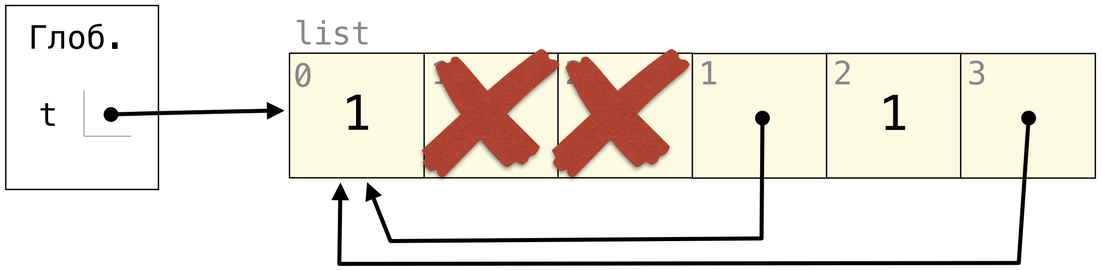



```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
```

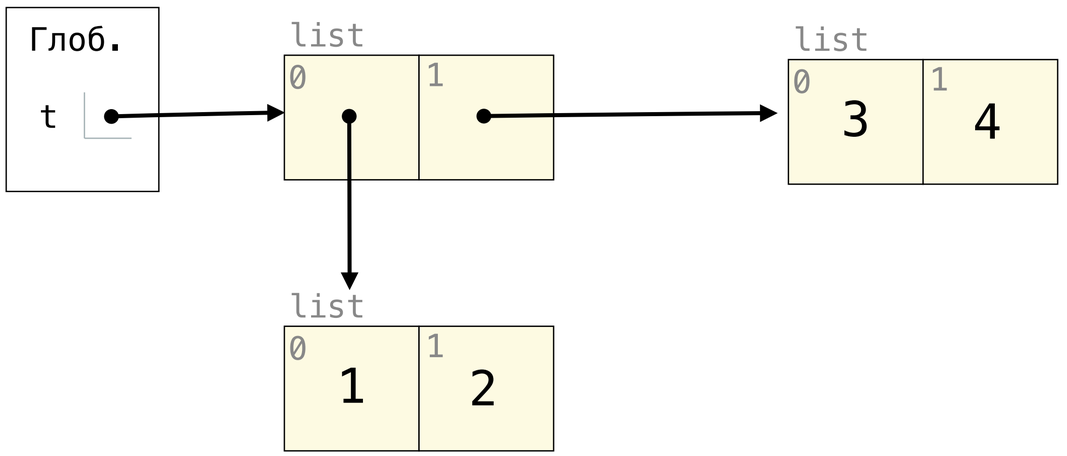
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



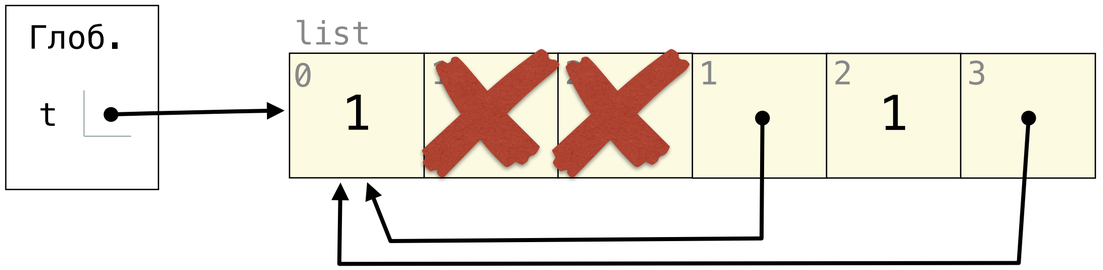

```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
```



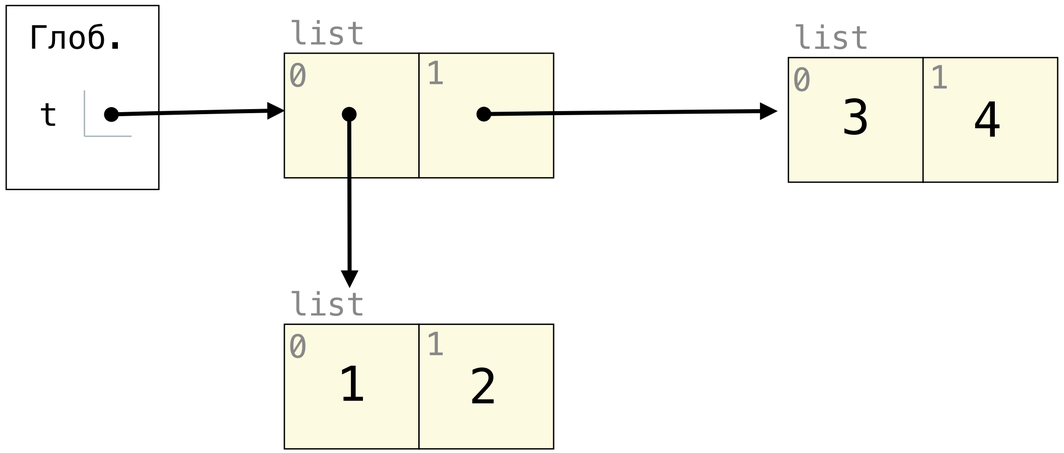
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



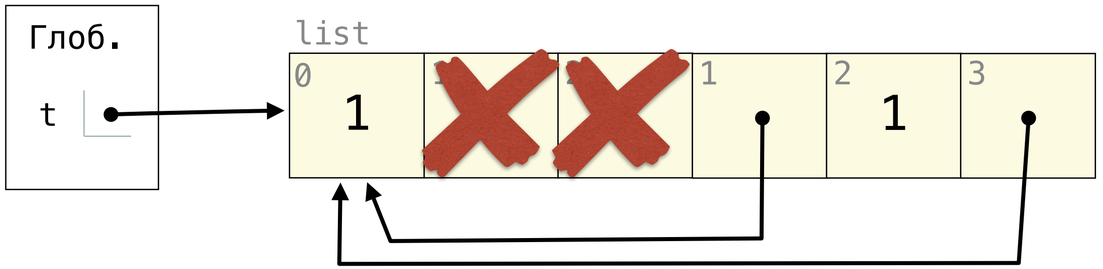

```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
t[0].append(t[1:2])
```



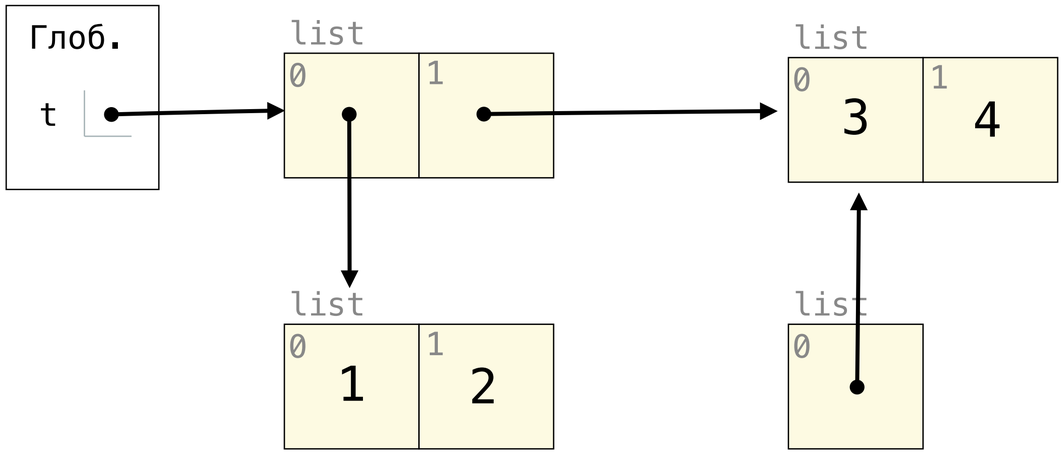
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



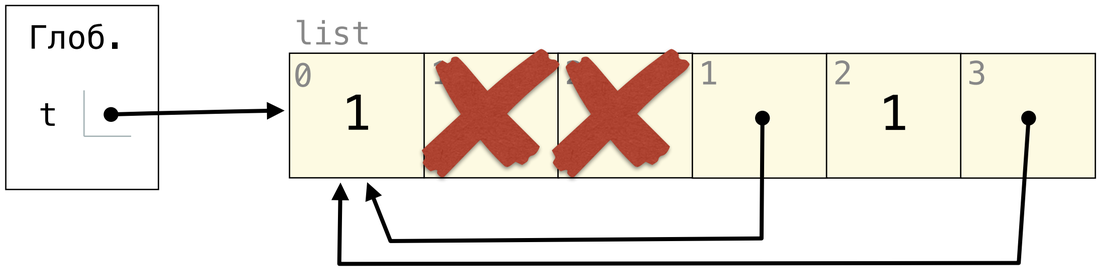

```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
t[0].append(t[1:2])
```



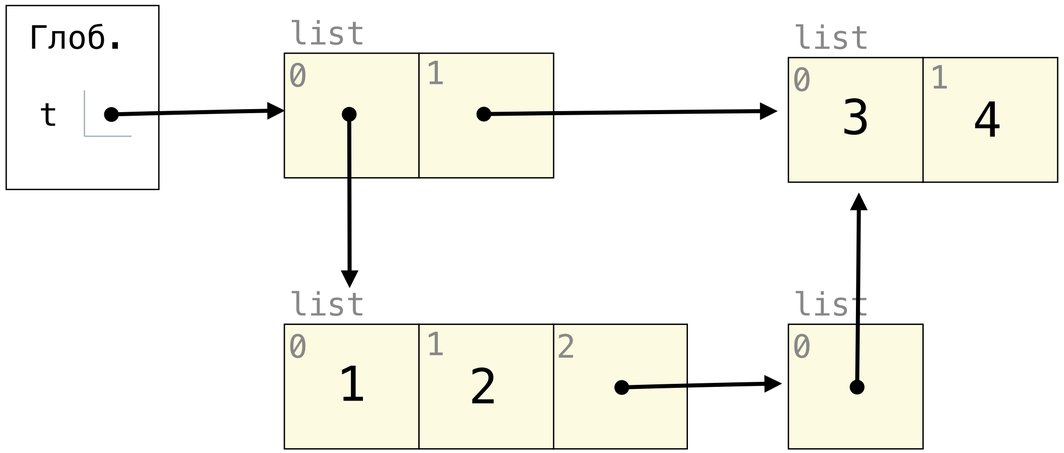
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



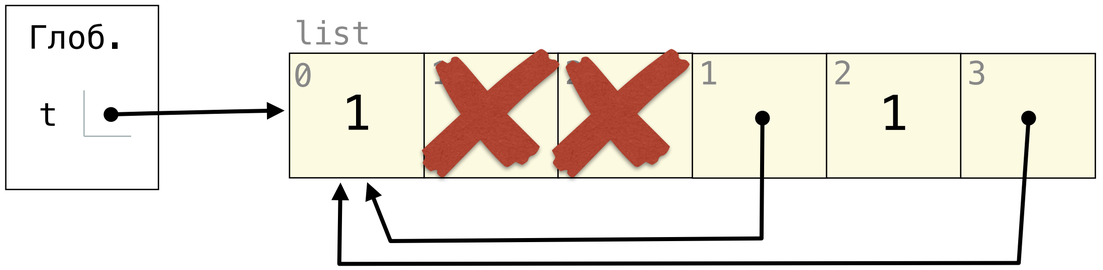

```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
t[0].append(t[1:2])
```



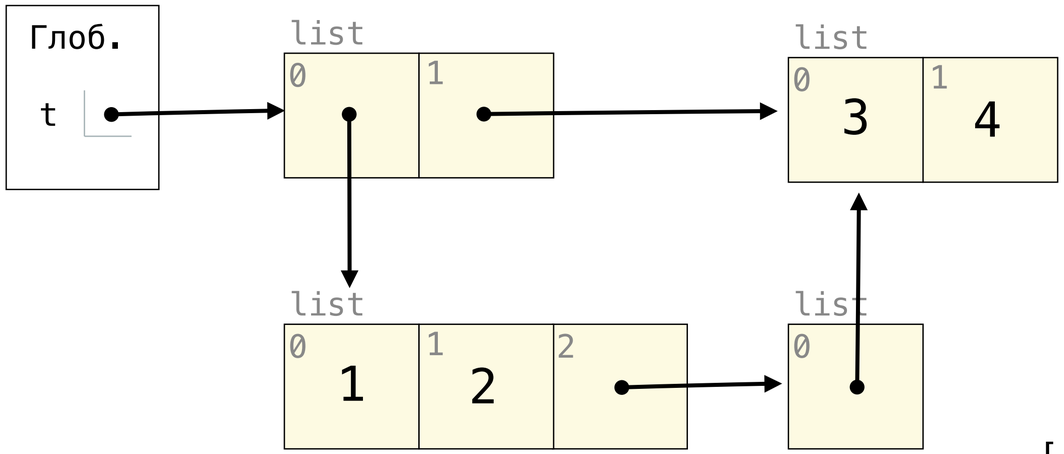
Списки в списках на диаграммах окружения

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```




```
[1, [...], 1, [...]]
```

```
t = [[1,2], [3,4]]
t[0].append(t[1:2])
```




```
[[1, 2, [[3, 4]]], [3, 4]]
```

Объекты

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
```

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
```

<class Worker>

greeting: 'Сэр'

<class Bourgeoisie>

greeting: 'Работник'

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'
```

<class Worker>

greeting: 'Сэр'

<class Bourgeoisie>

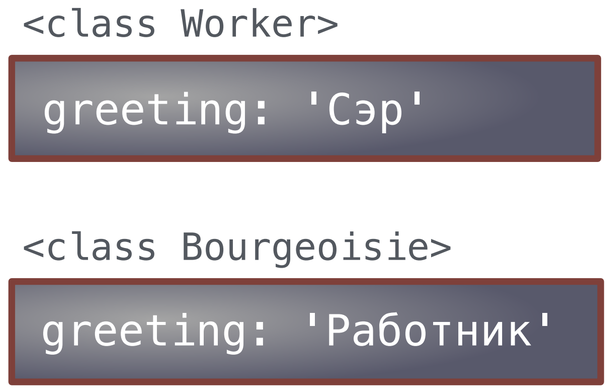
greeting: 'Работник'

Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
```

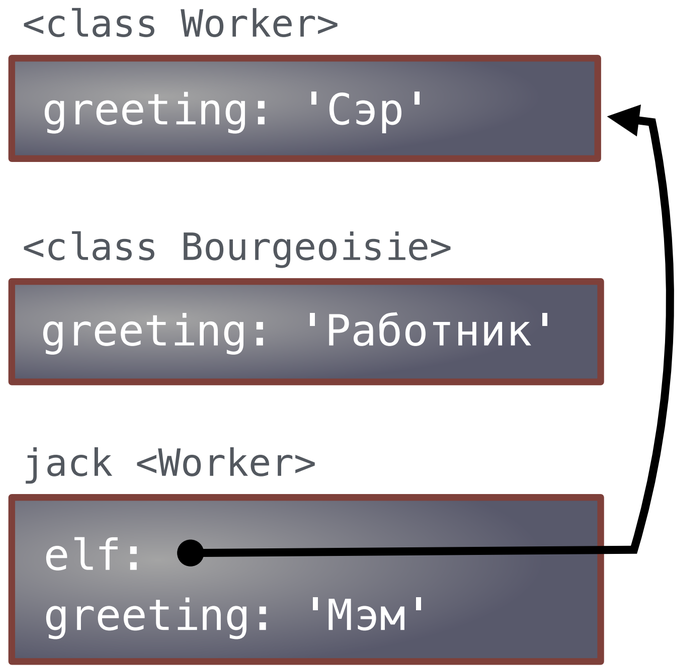


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
```

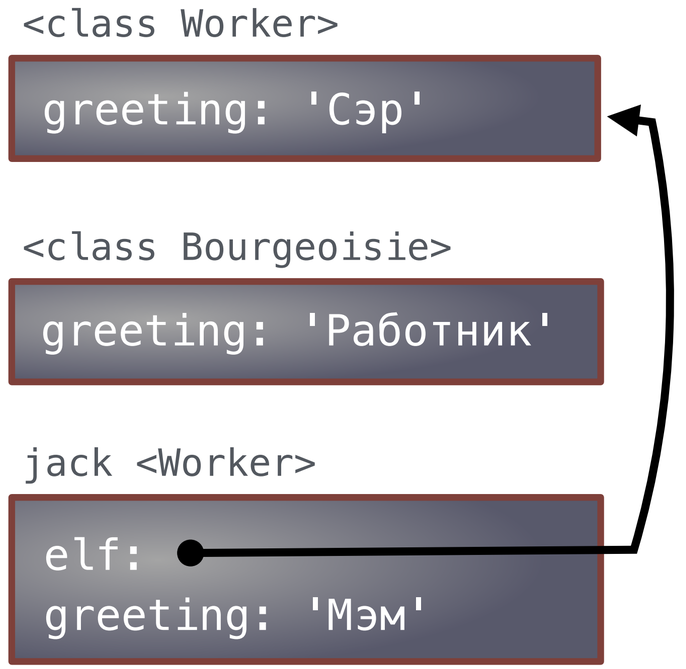


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'
```

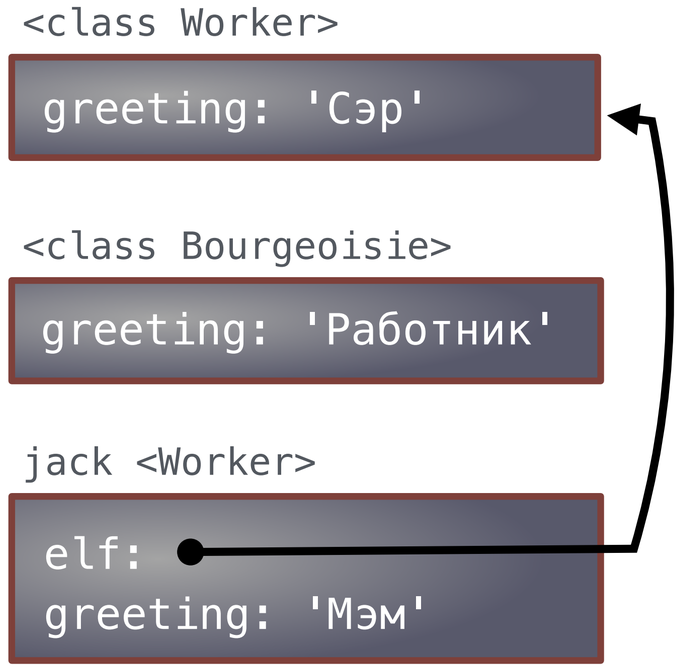


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
```

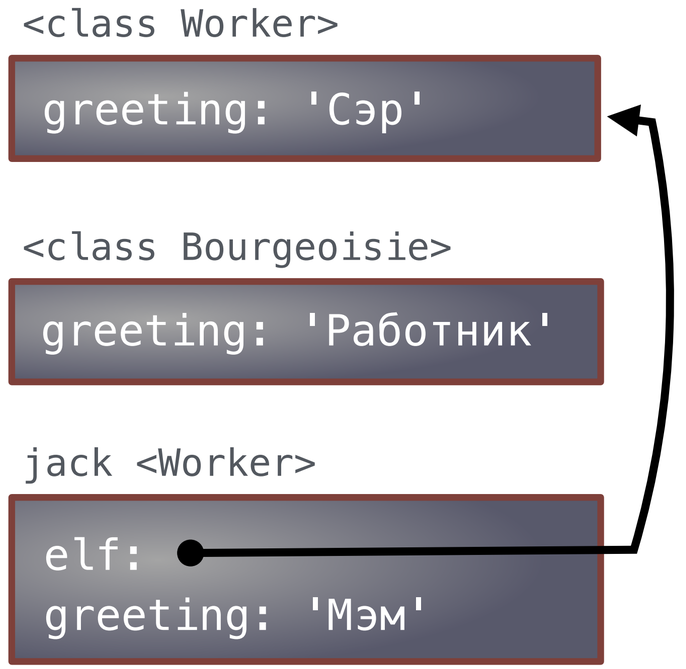


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'
```

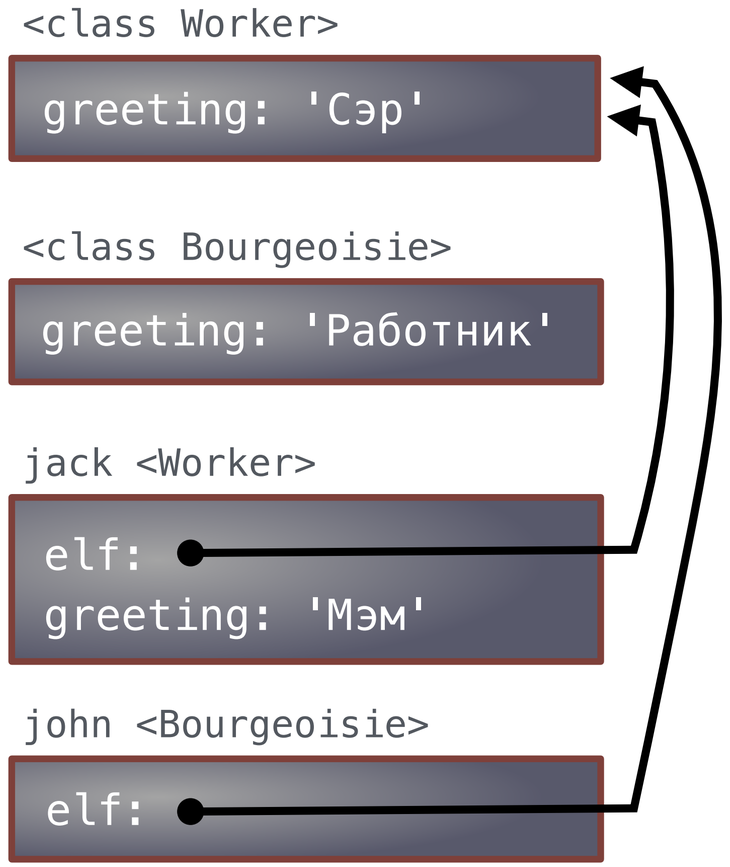


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'

>>> john.work()
```

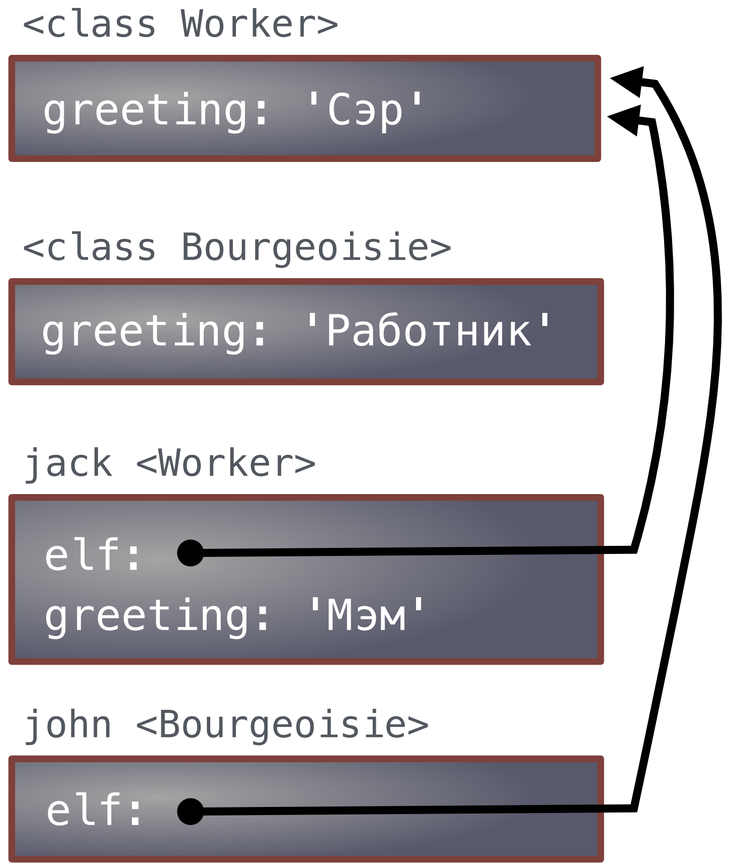


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'

>>> john.work()
Работник, я работаю
```

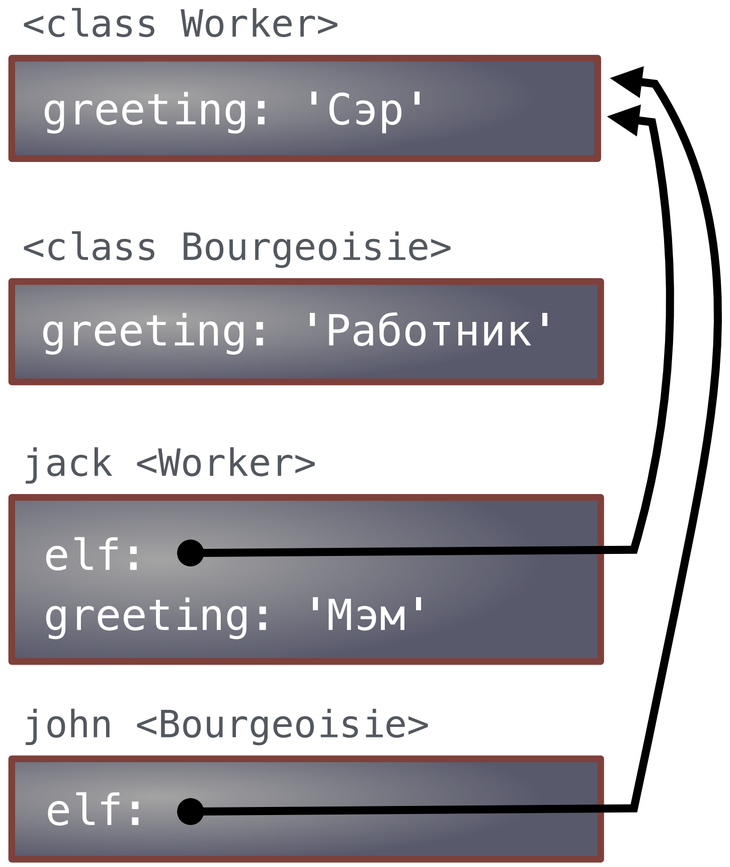


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'

>>> john.work()
Работник, я работаю
'Я богатею'
```

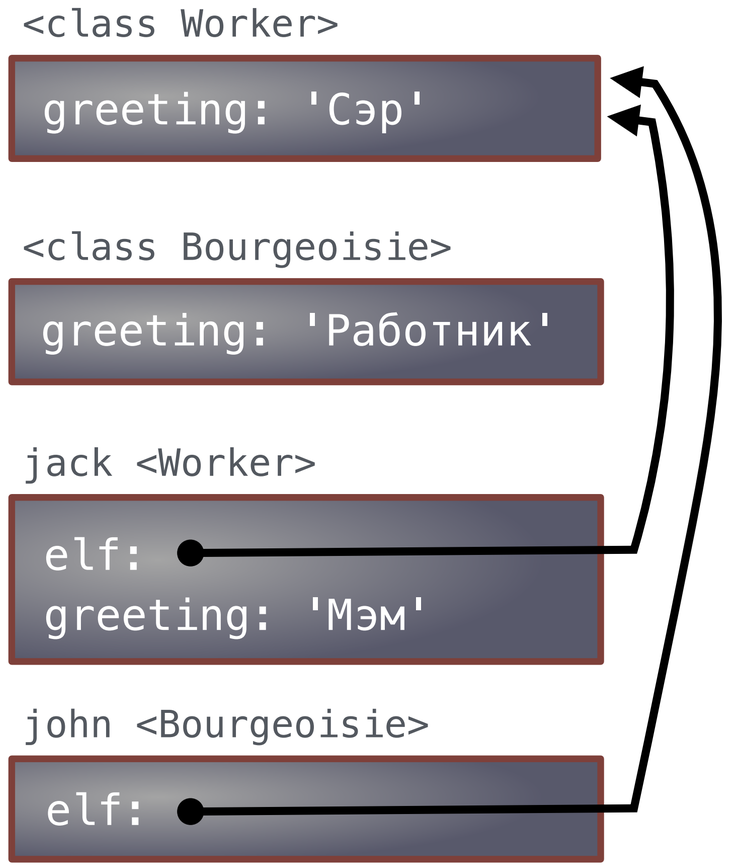


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'

>>> john.work()
Работник, я работаю
'Я богатею'


>>> john.elf.work(john)
```

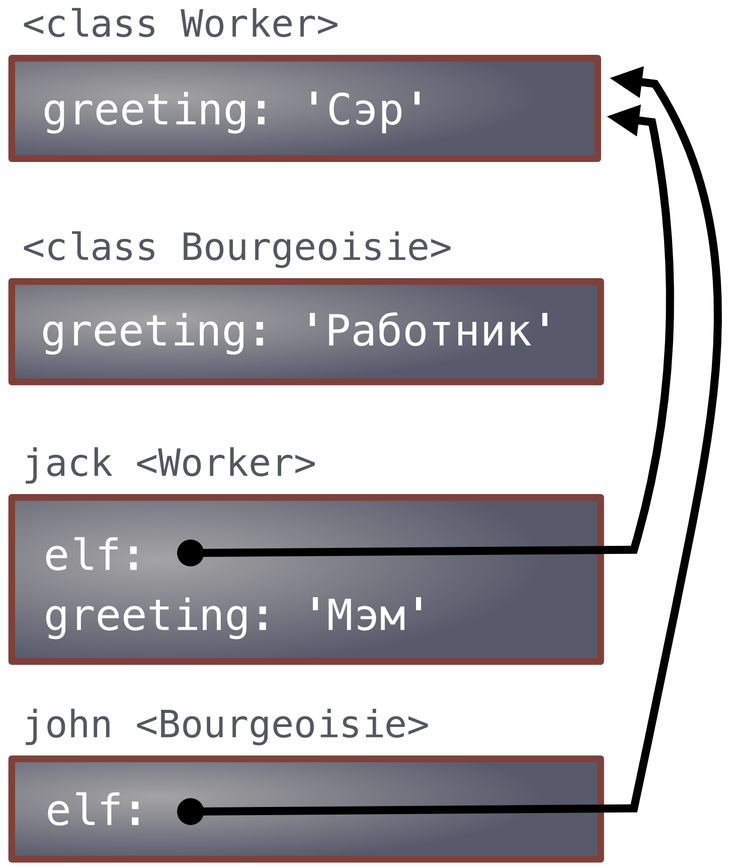


Лэндлорды-с

Поиск имён сначала происходит в атрибутах экземпляра, затем в атрибутах класса; атрибуты класса наследуются.

```
class Worker:
    greeting = 'Сэр'
    def __init__(self):
        self.elf = Worker
    def work(self):
        return self.greeting + ', я работаю'
    def __repr__(self):
        return Bourgeoisie.greeting

class Bourgeoisie(Worker):
    greeting = 'Работник'
    def work(self):
        print(Worker.work(self))
        return 'Я богатею'


jack = Worker()
john = Bourgeoisie()
jack.greeting = 'Мэм'
```

```
>>> Worker().work()
'Сэр, я работаю'

>>> jack
'Работник'

>>> jack.work()
'Мэм, я работаю'

>>> john.work()
Работник, я работаю
'Я богатею'


>>> john.elf.work(john)
'Работник, я работаю'
```



Связные списки

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

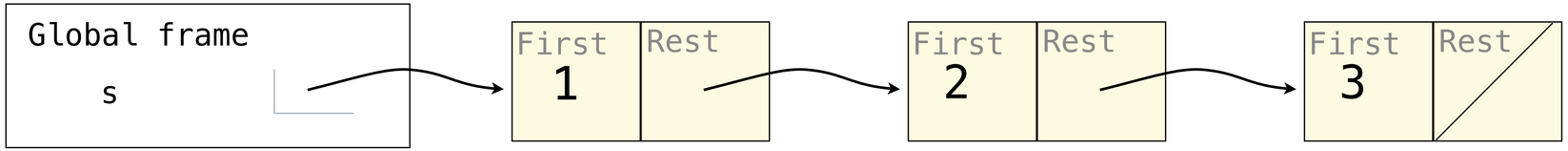
```
>>> s = Link(1, Link(2, Link(3)))
```

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
```

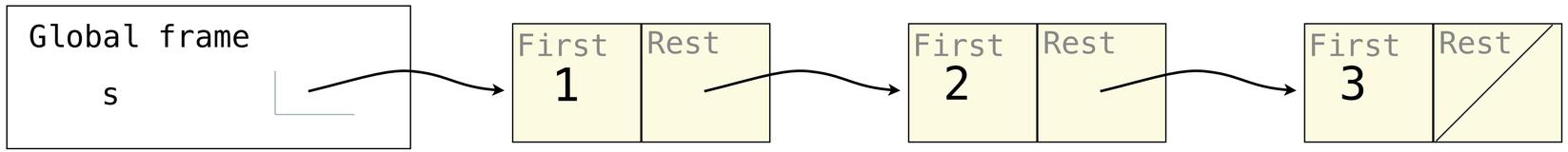


Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
```



Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
```

Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

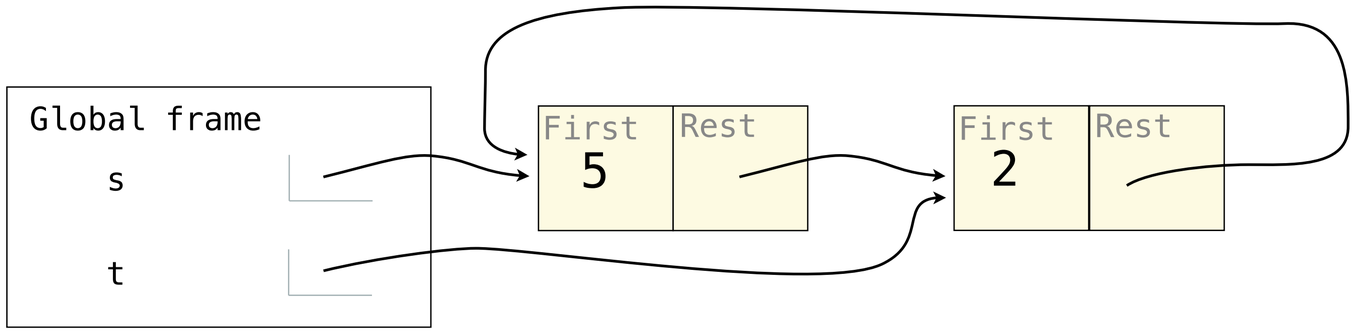
Внимание: реальная диаграмма окружения куда как более сложна.

Рекурсивные списки могут изменяться

Инструкции присвоения атрибутам могут изменять атрибуты first и rest экземпляра класса Link.

Атрибут rest связного списка может содержать связный список в качестве подсписка.

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```



Внимание: реальная диаграмма окружения куда как более сложна.

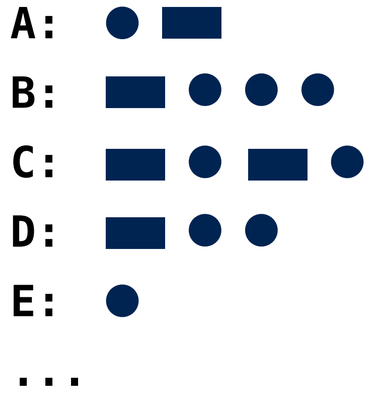
Деревья

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Код Морзе

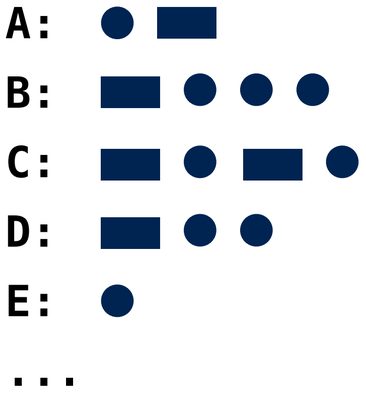
Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).



Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала

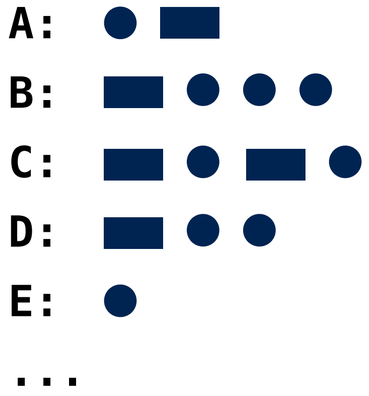


Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала

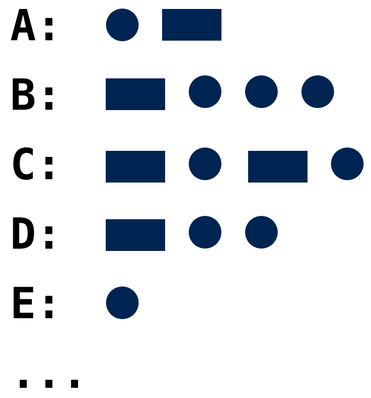
```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```



Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала



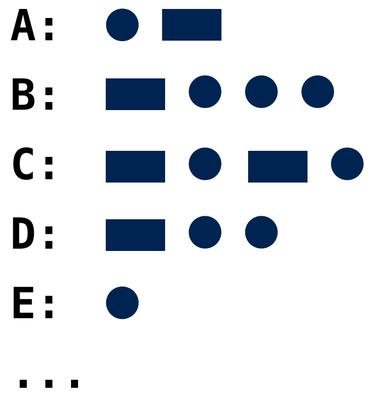

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала




```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

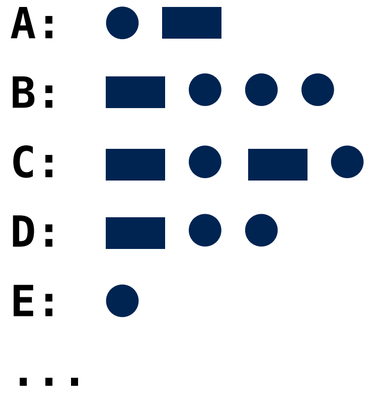
```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала




```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

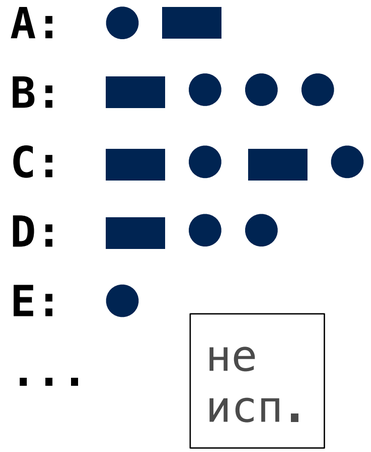
```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала




```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

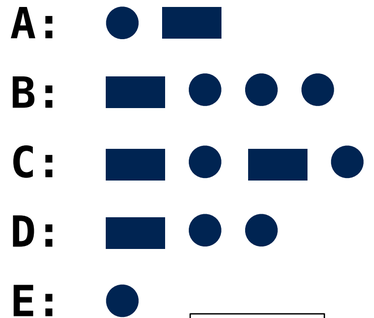
```
def morse(code):
    ....
```

```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала




```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

не исп.

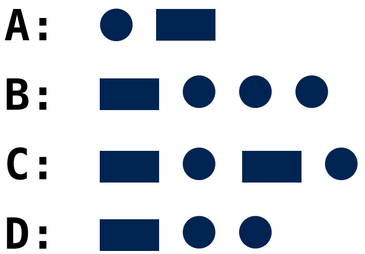
'.'

Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

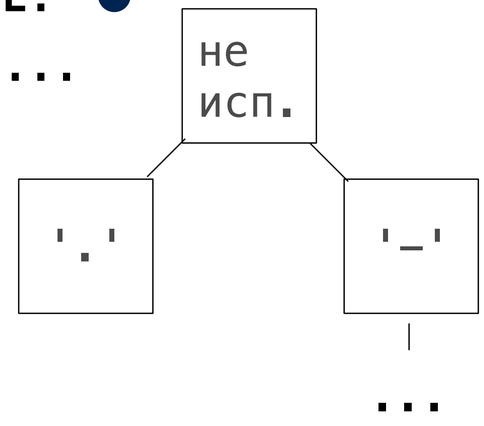
Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```




```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

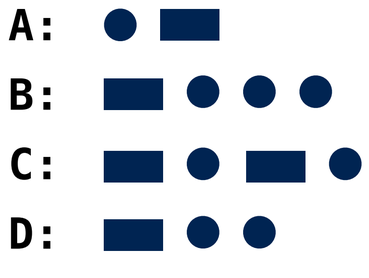


Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

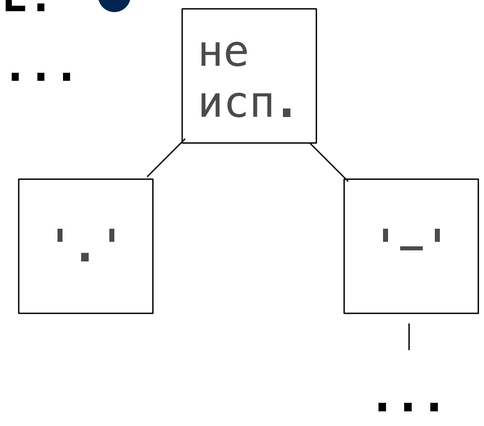
Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```




```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

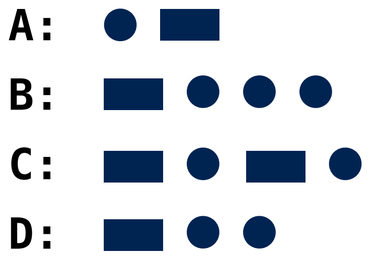


Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

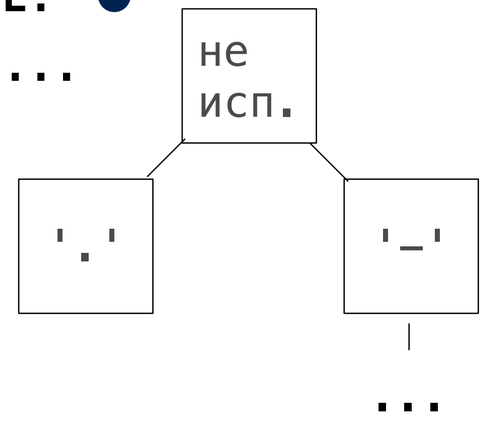
Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```




```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```

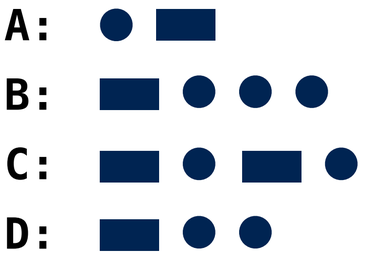


Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

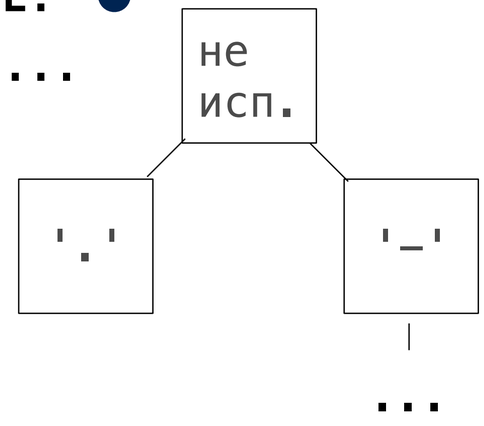
Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```




```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```



Код Морзе

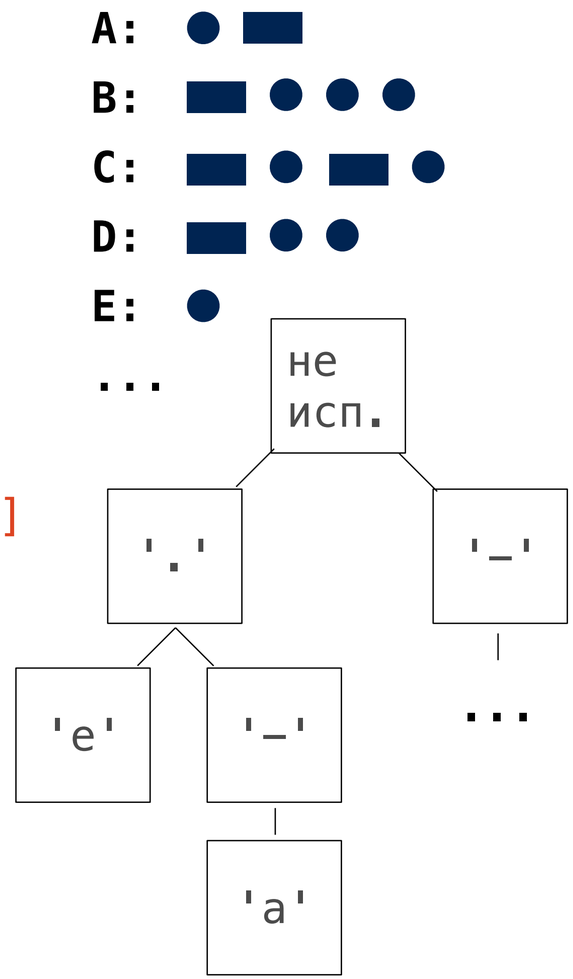
Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

```
def morse(code):
    ....
```



Код Морзе

Код Морзе – способ кодирования – представление символов последовательностью сигналов: длинных («тире») и коротких («точек»).

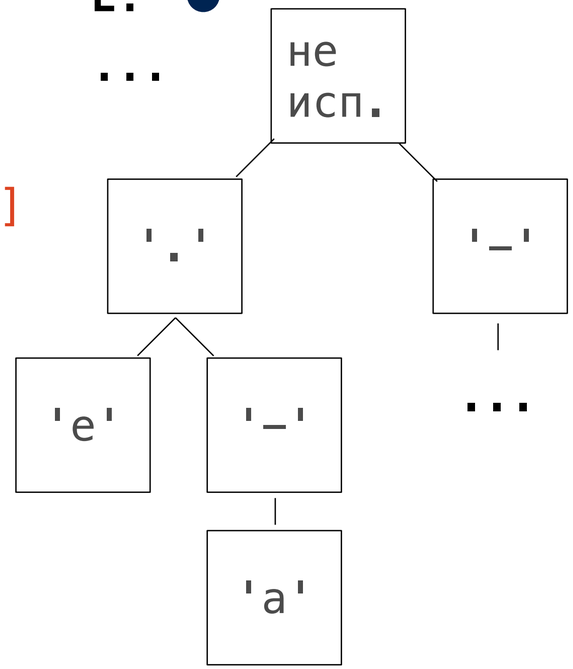
Задача: Создать функцию **morse**, чтобы **decode** верно работала

```
abcde = {'a': '.-', 'b': '-...', 'c': '-.-.', 'd': '-..', 'e': '.'}
```

```
def decode(signals, tree):
    """Декодирование сигналов в букву.

    >>> t = morse(abcde)
    >>> [decode(s, t) for s in ['-..', '.', '-.-.', '.-', '-..', '.']]
    ['d', 'e', 'c', 'a', 'd', 'e']
    """
    for signal in signals:
        tree = [b for b in tree.branches if b.label == signal][0]
    leaves = [b for b in tree.branches if b.is_leaf()]
    assert len(leaves) == 1
    return leaves[0].label
```

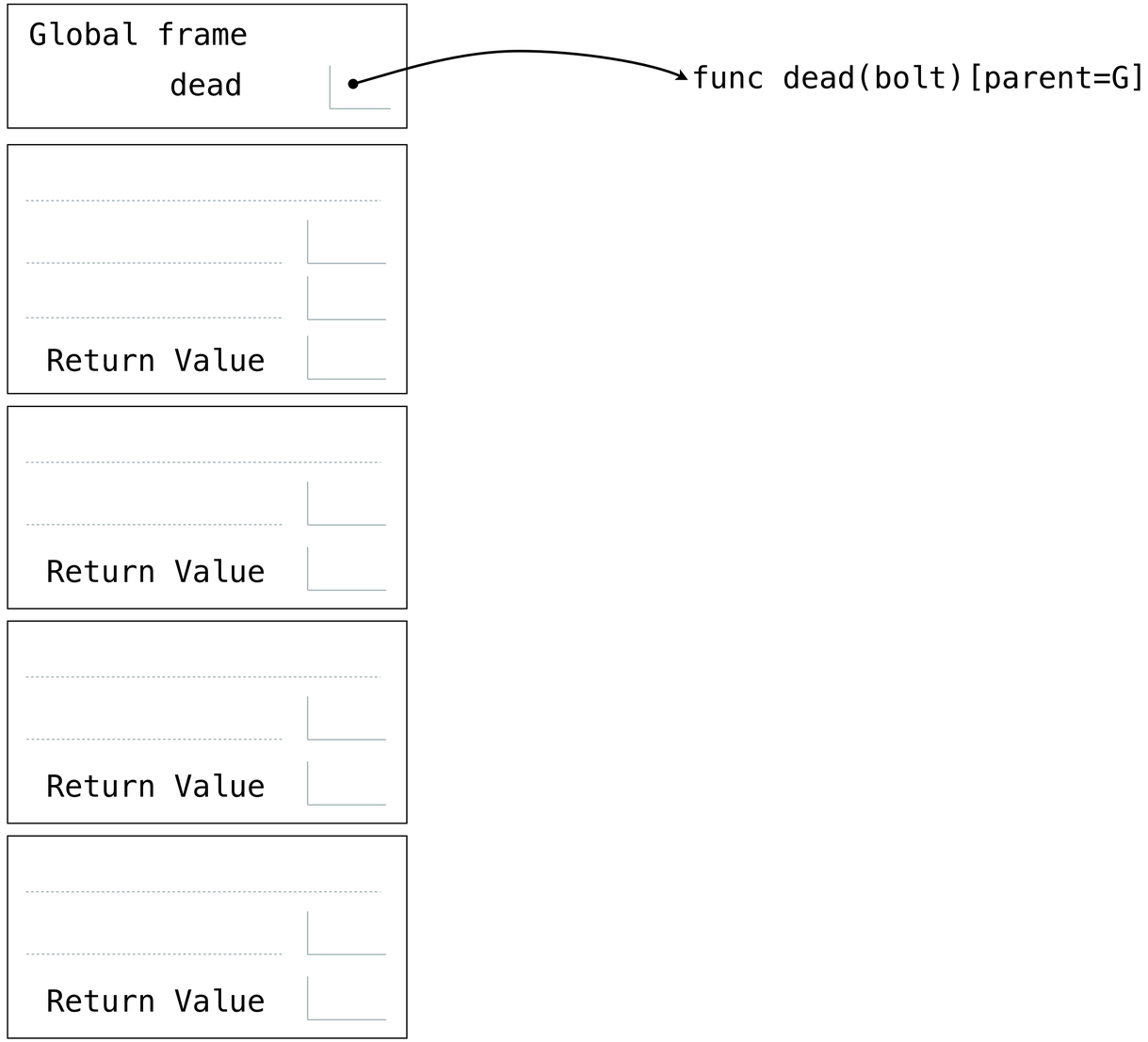
```
def morse(code):
    ....
```

A: ● ▬
B: ▬ ● ● ●
C: ▬ ● ▬ ●
D: ▬ ● ●
E: ●
...



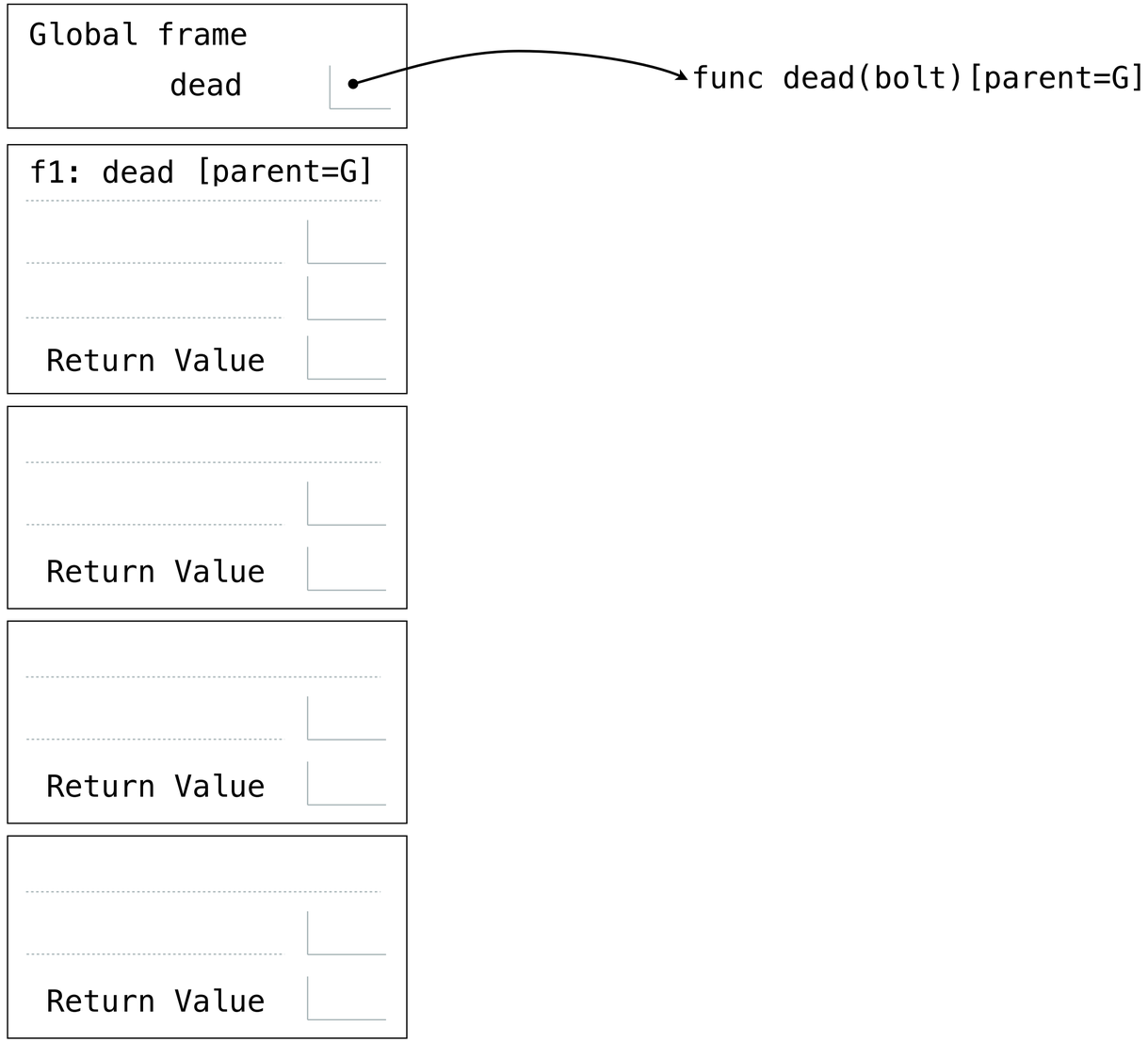
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



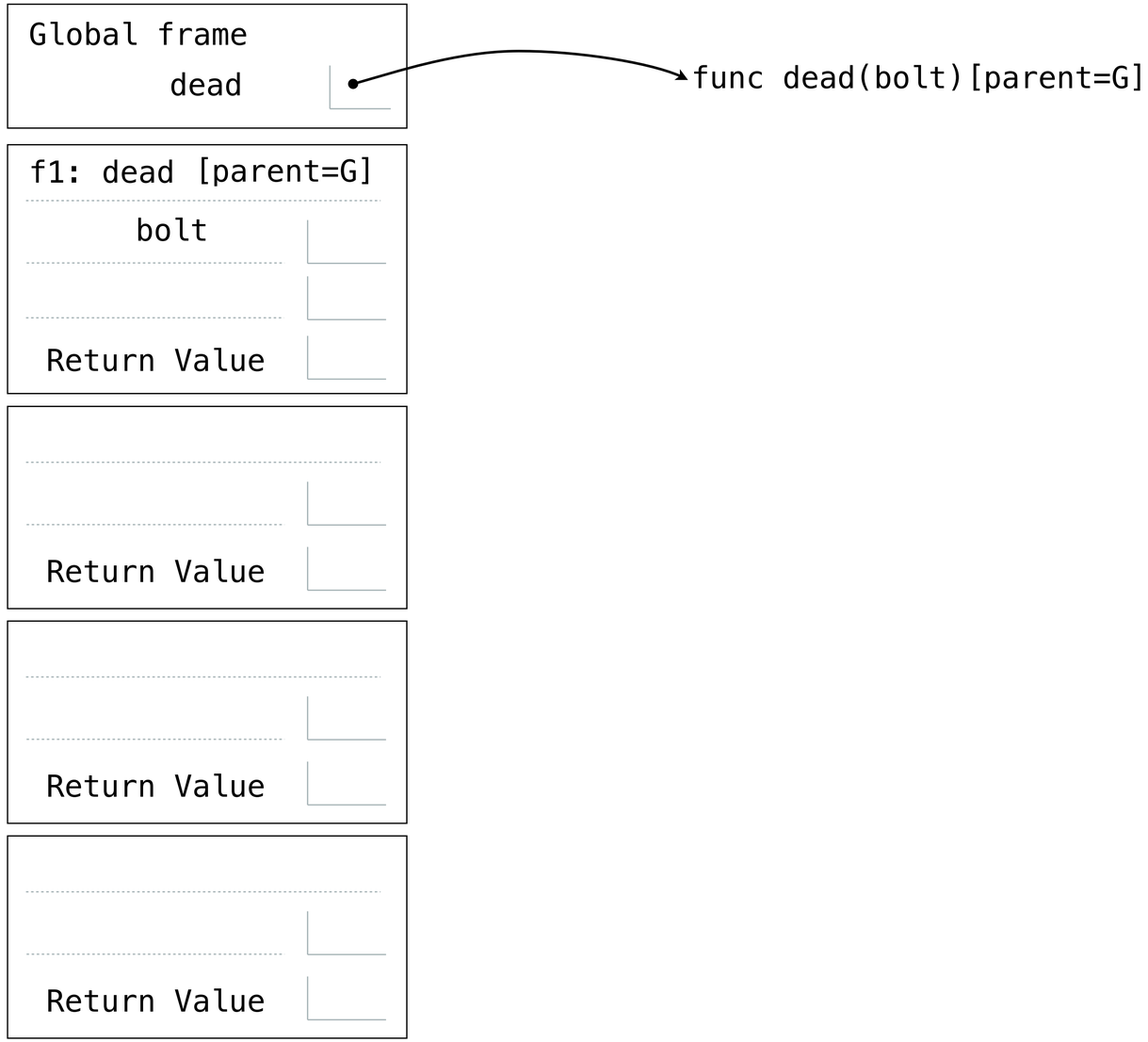
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



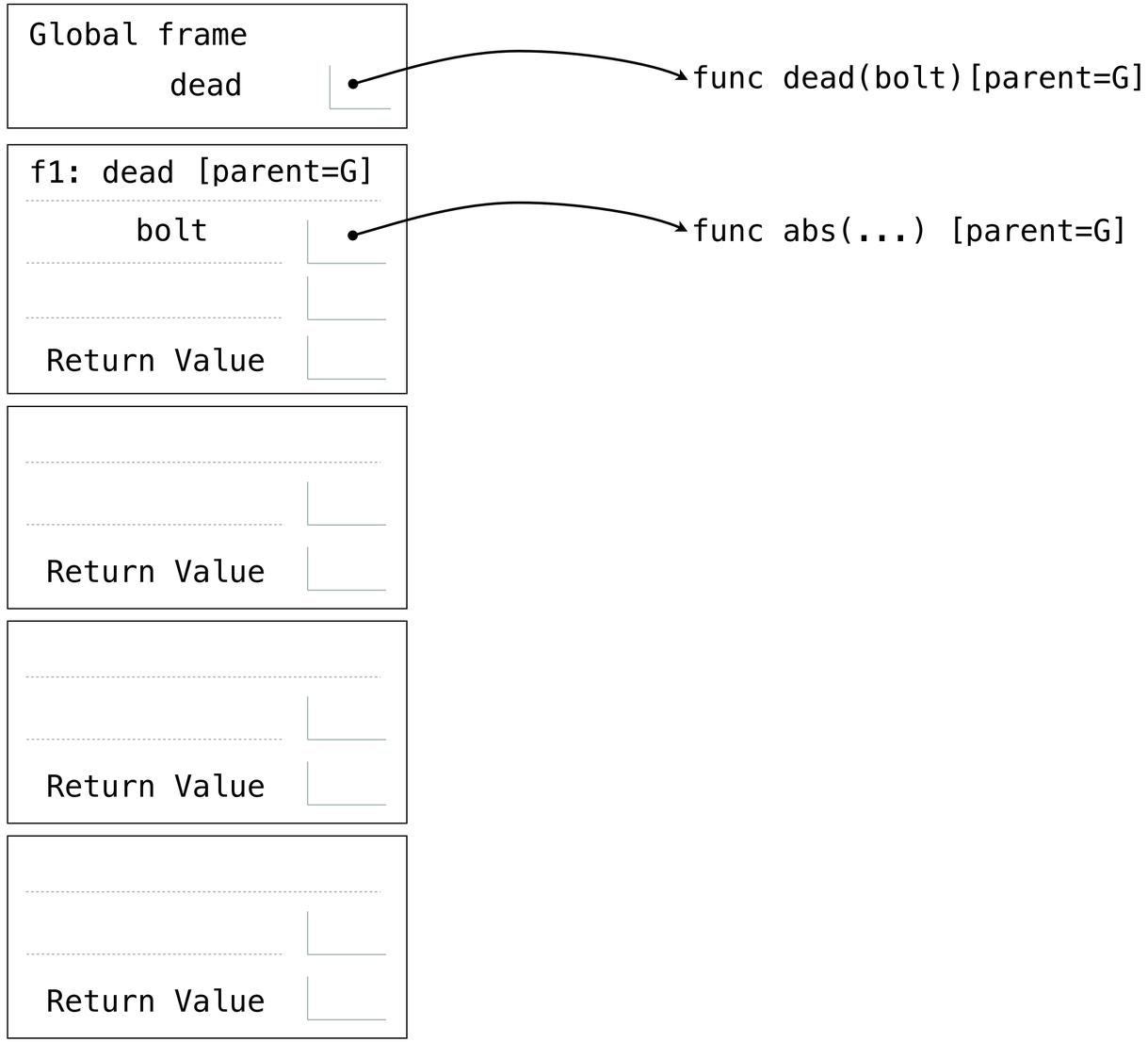
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



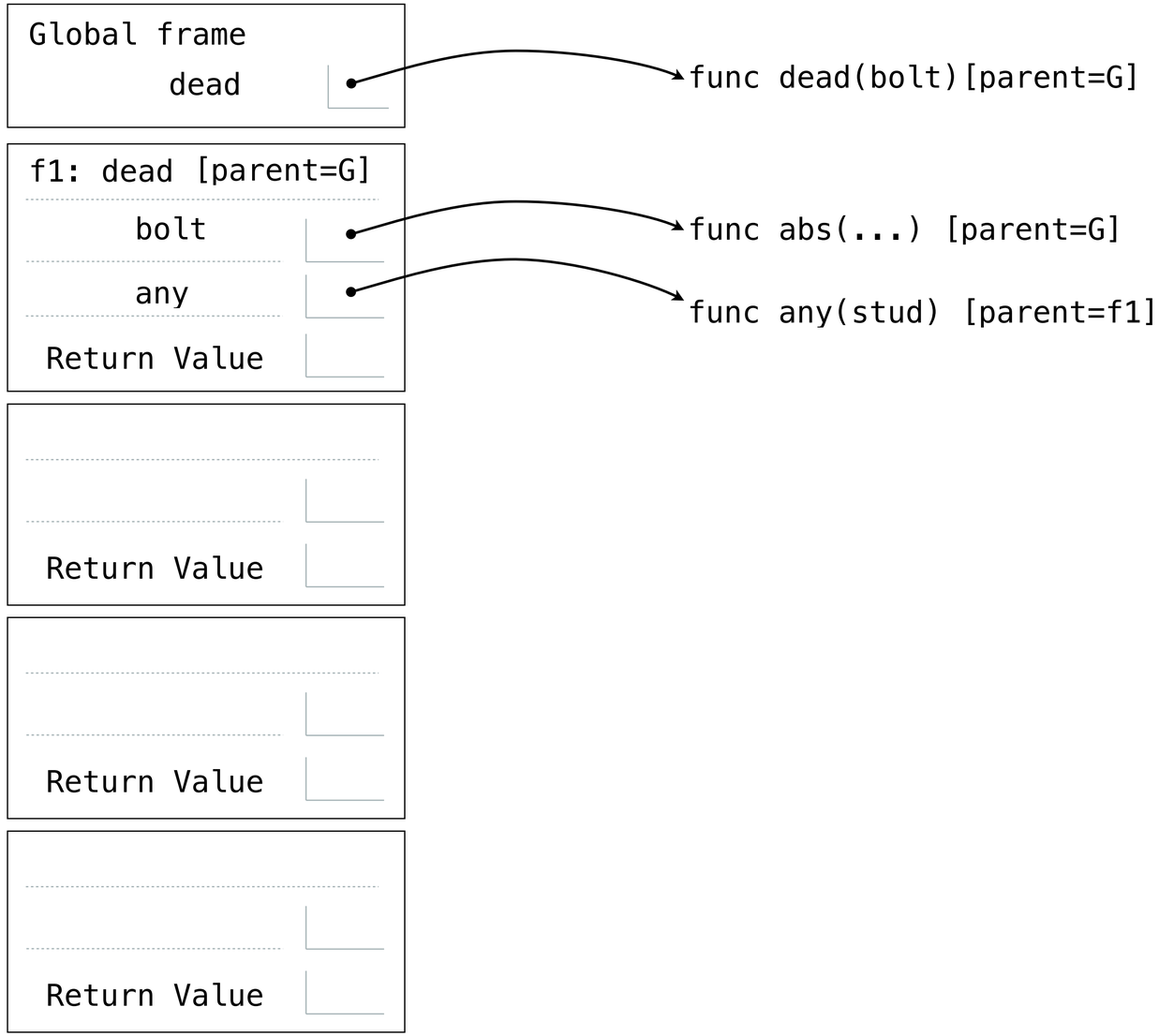
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



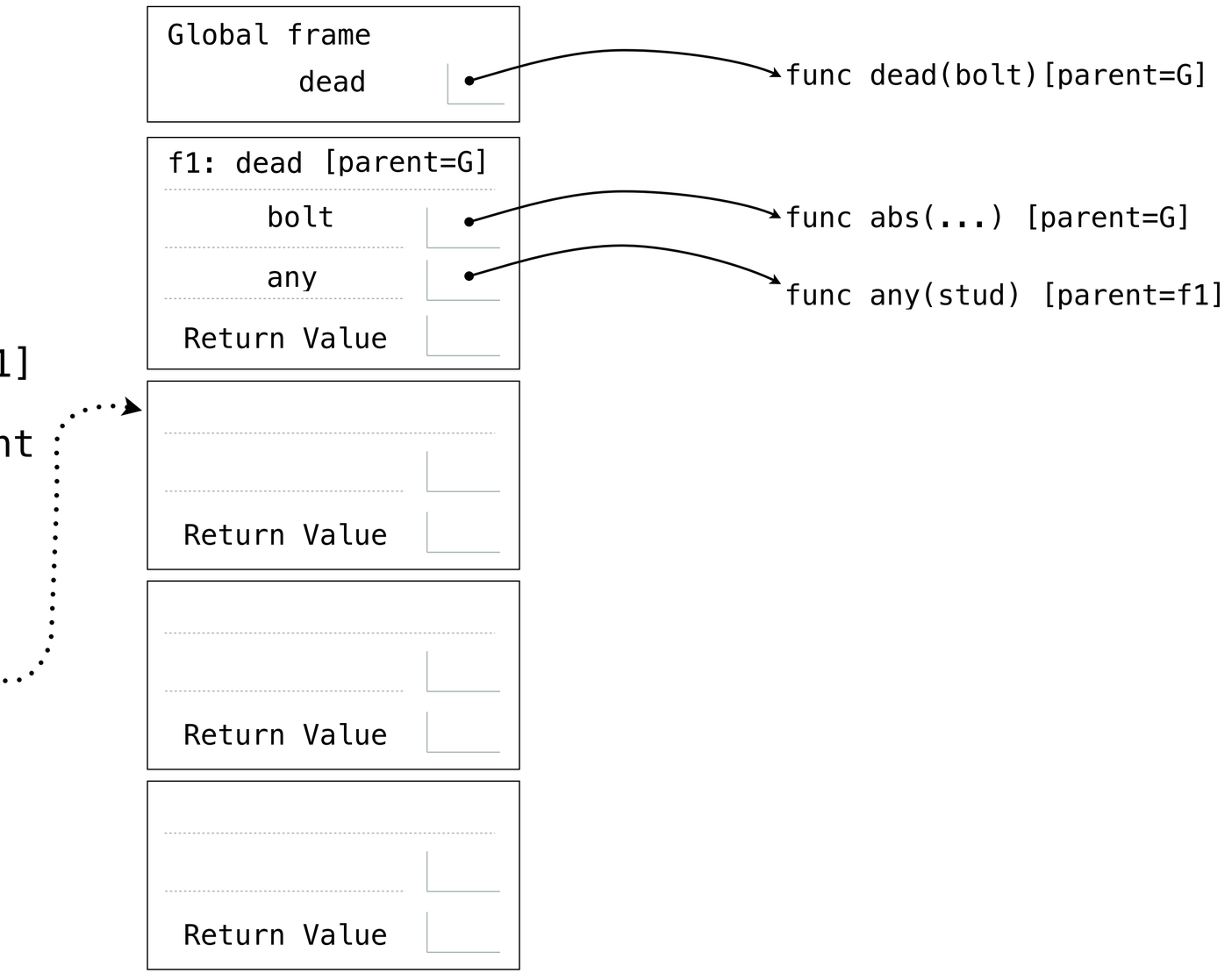
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



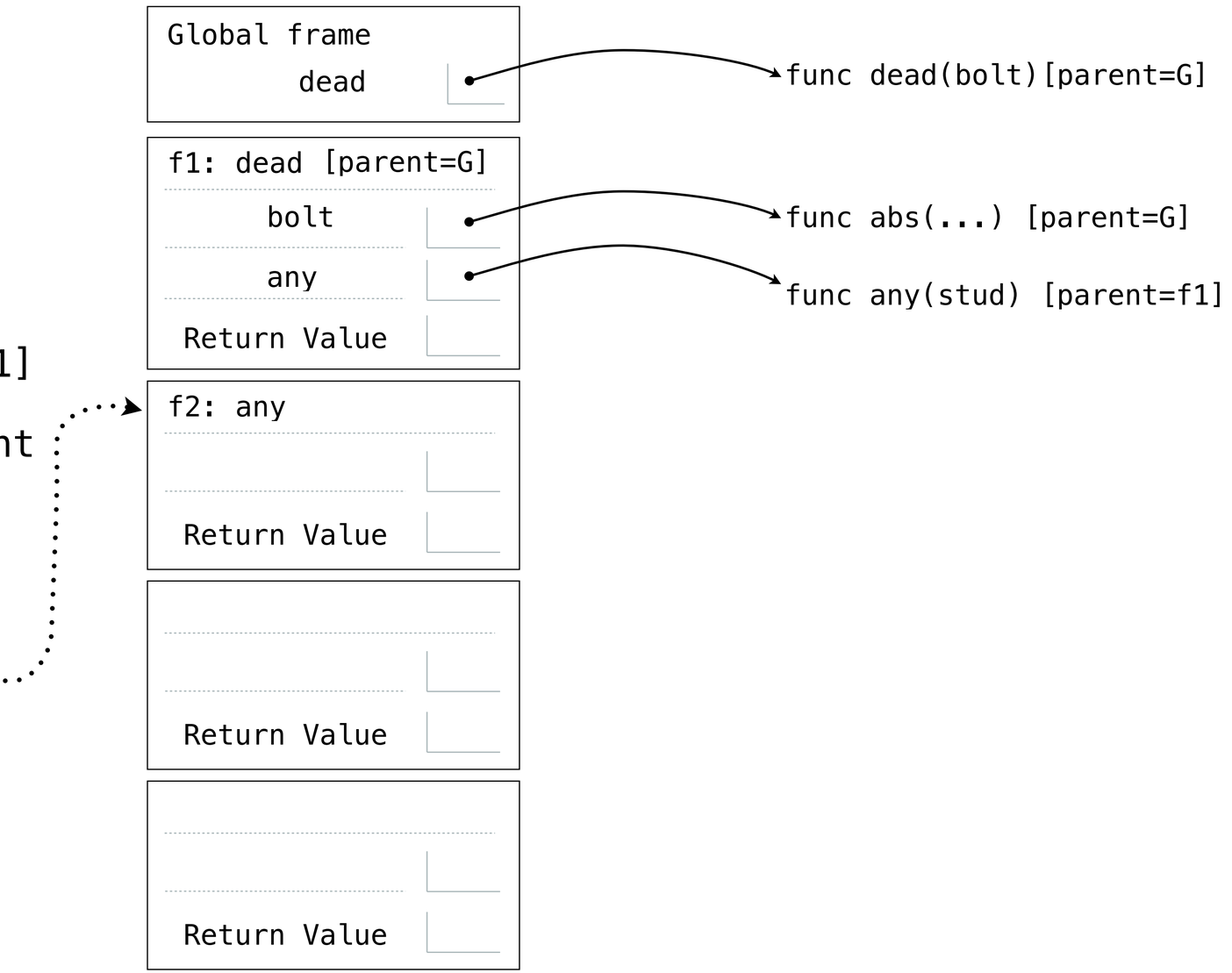
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



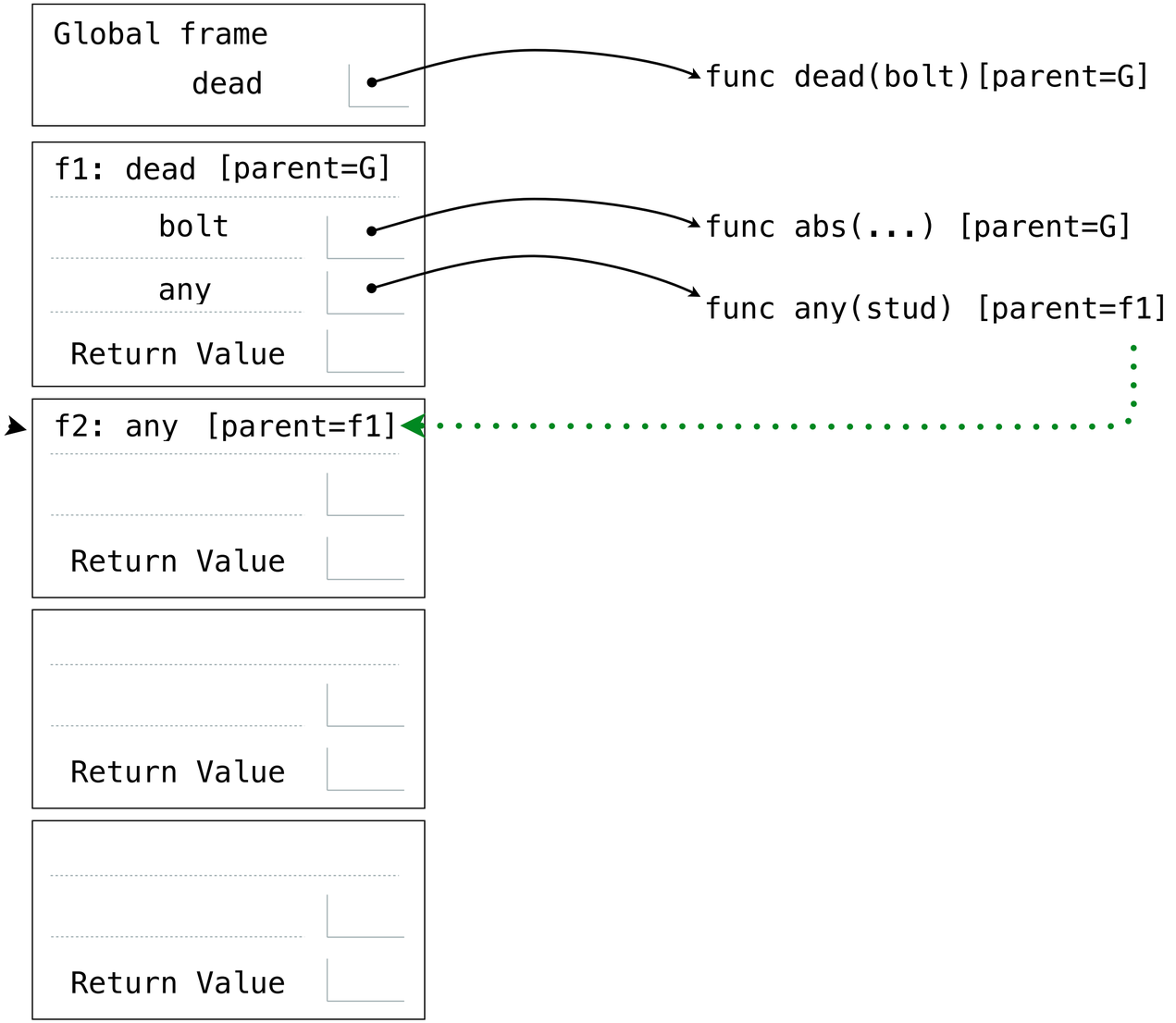
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



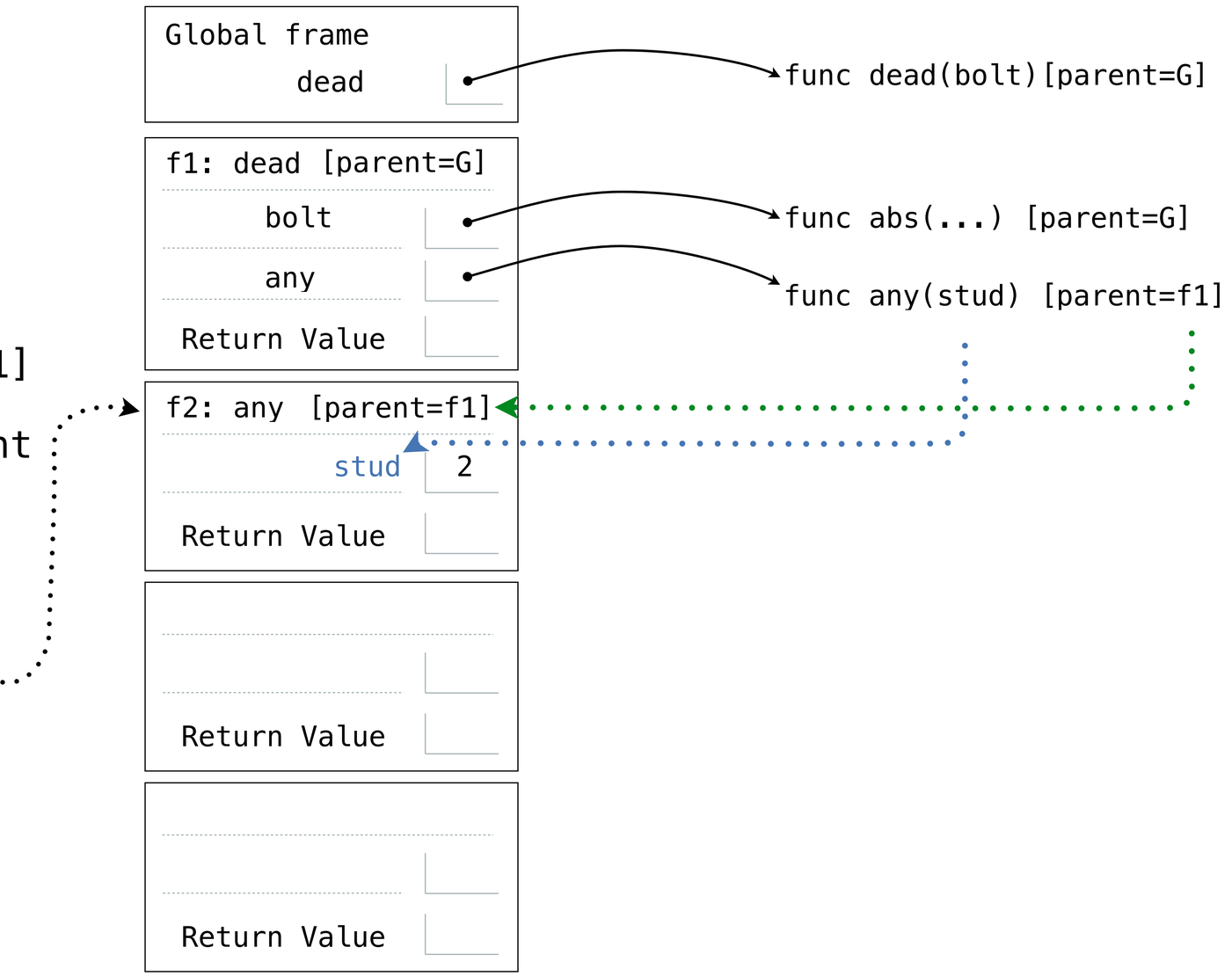
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



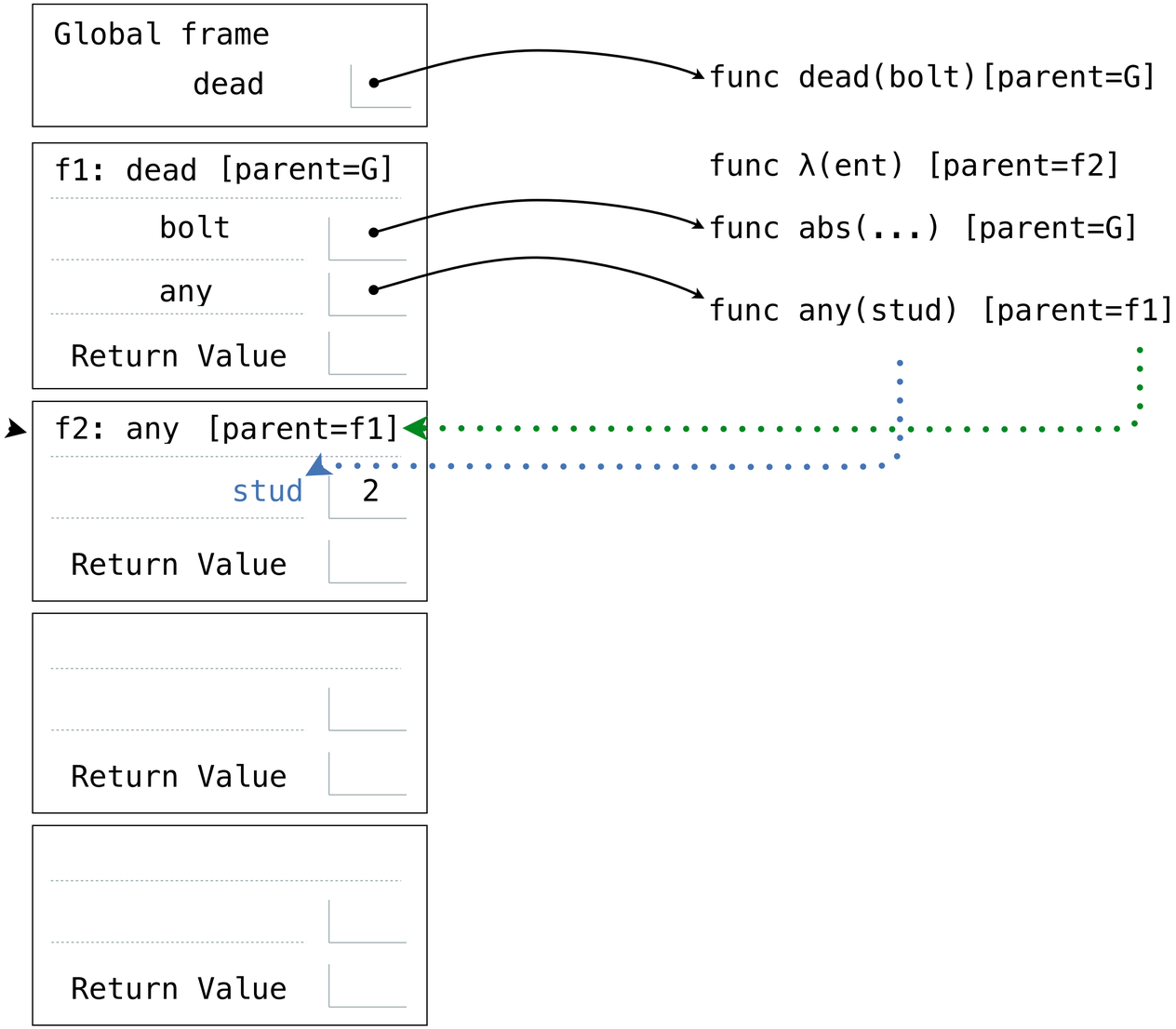
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



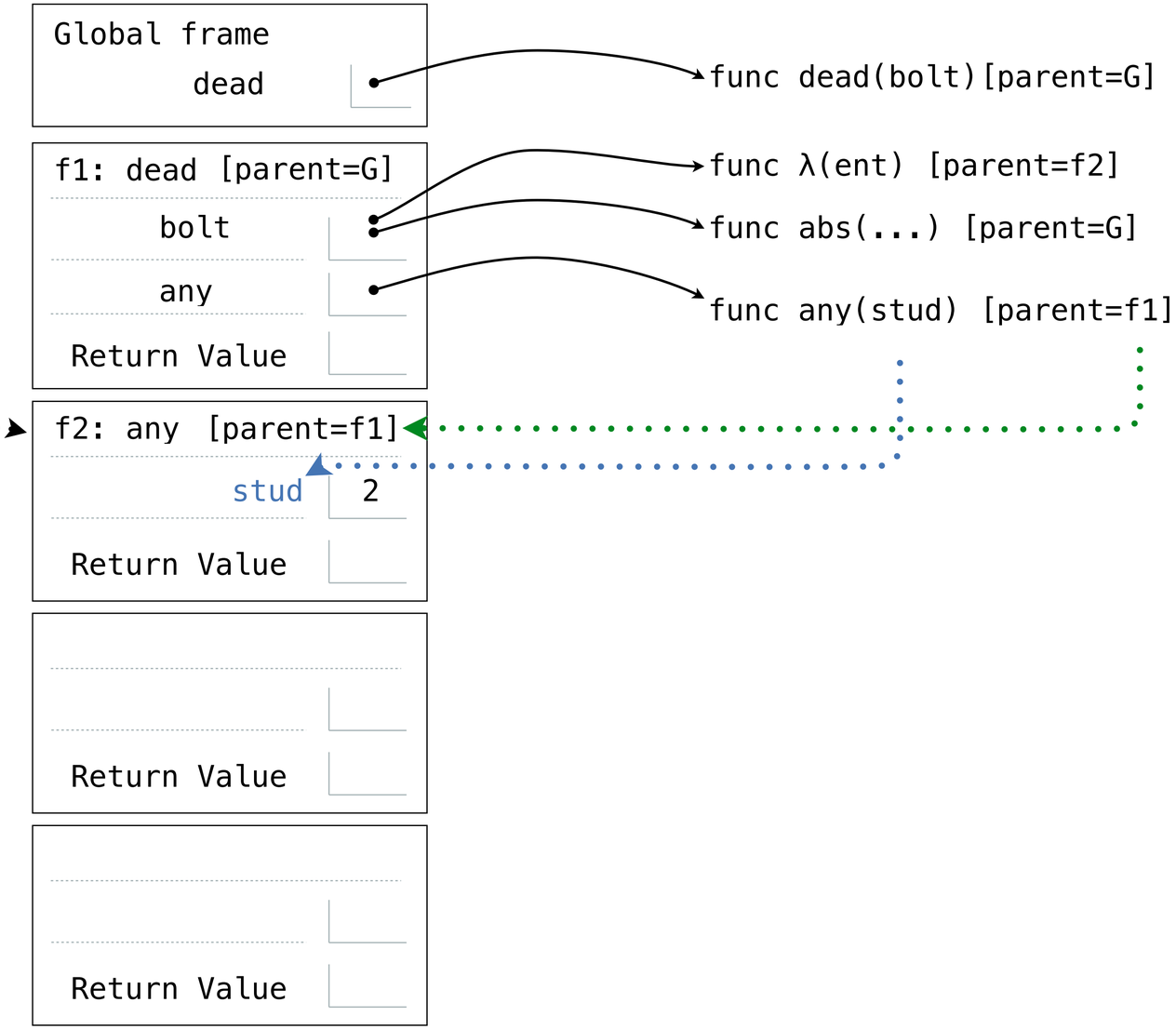
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



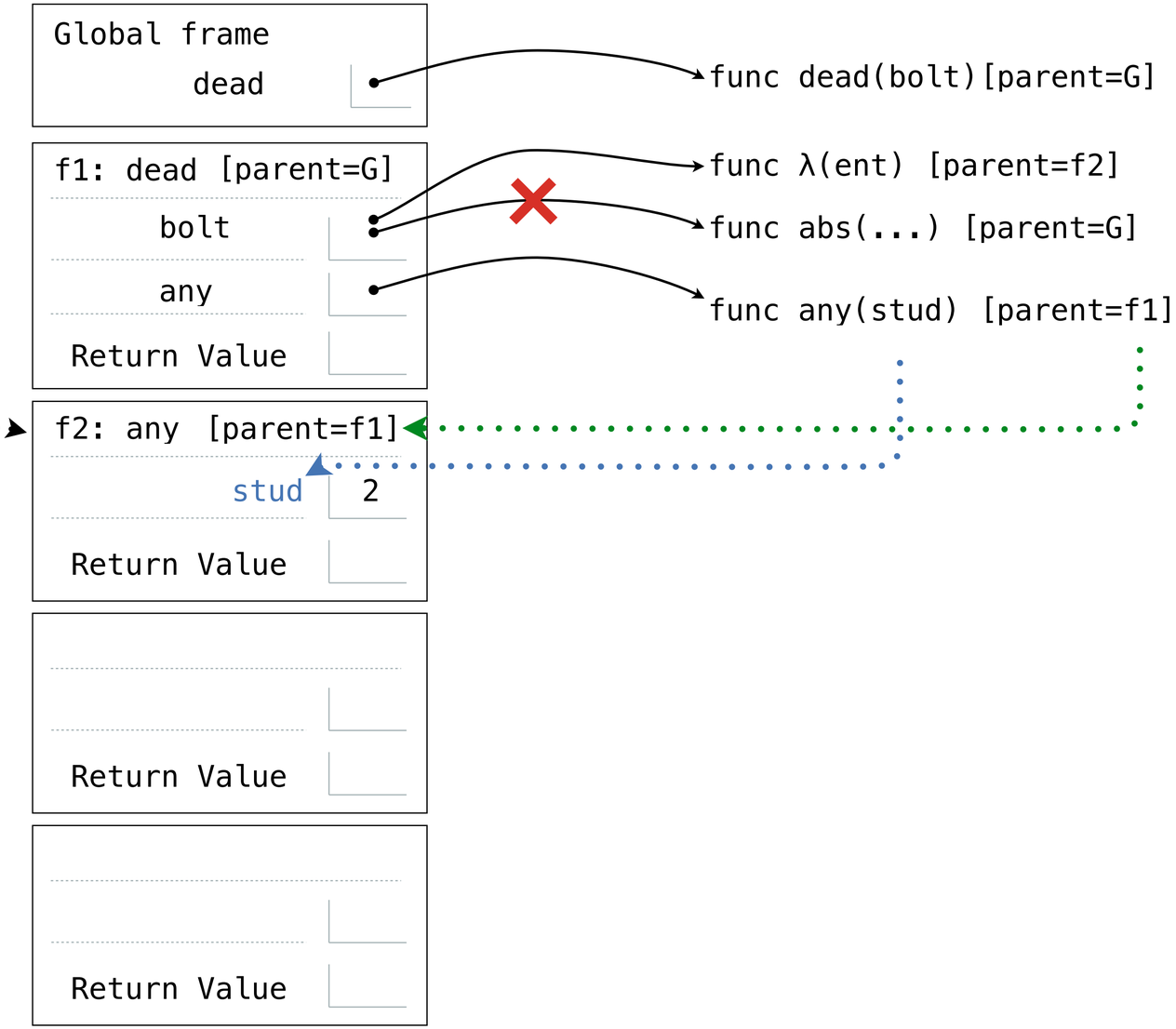
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



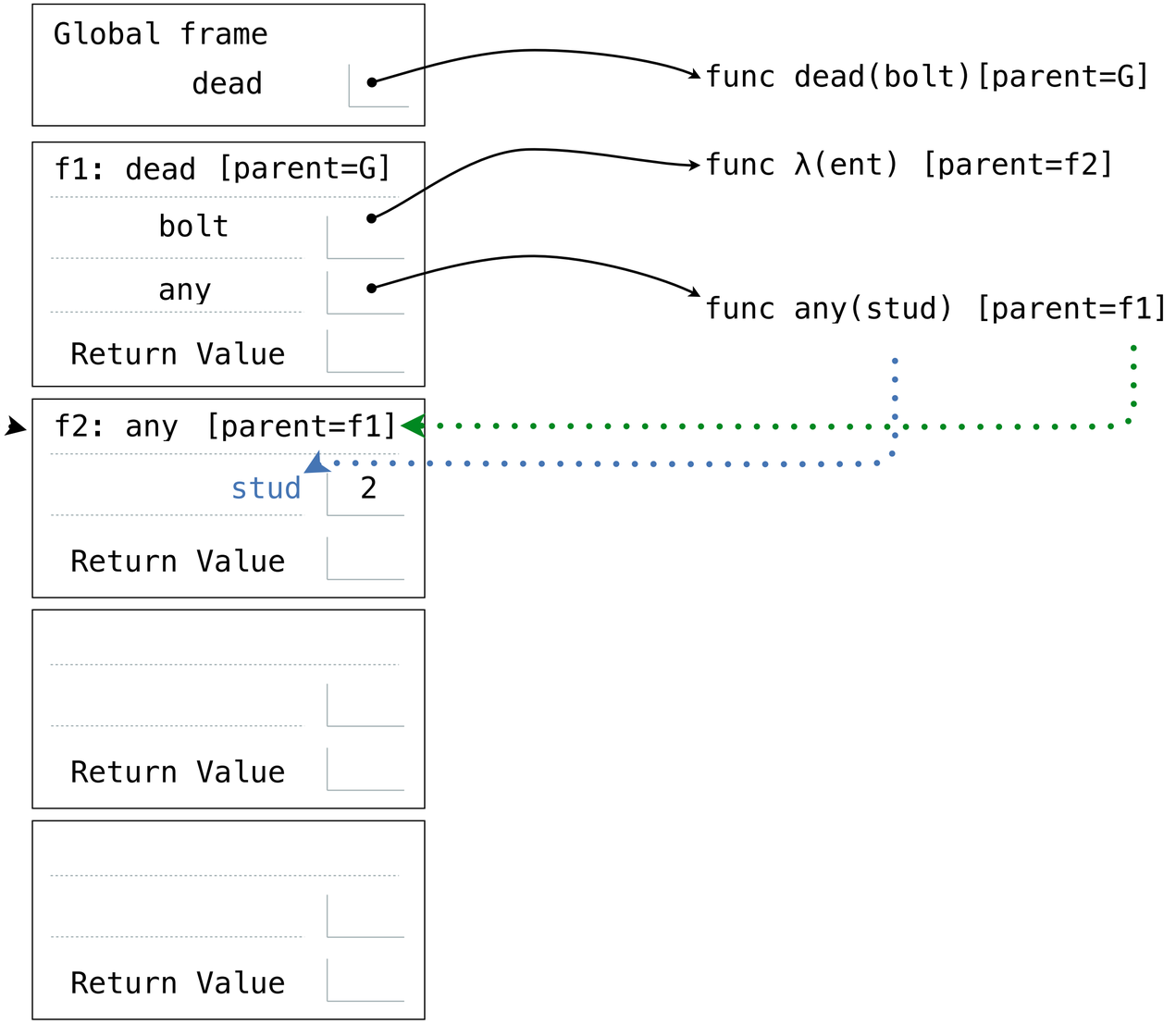
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



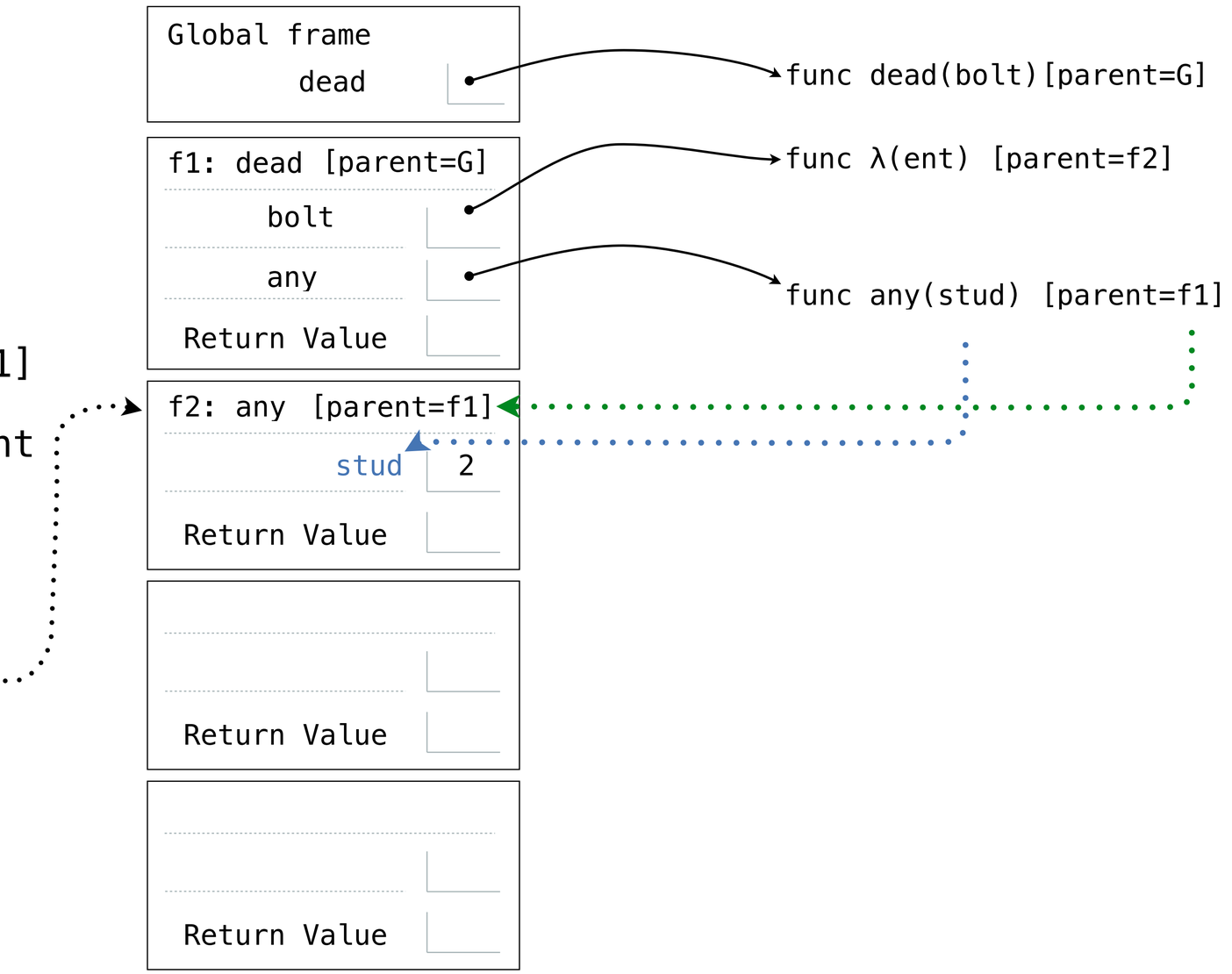
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



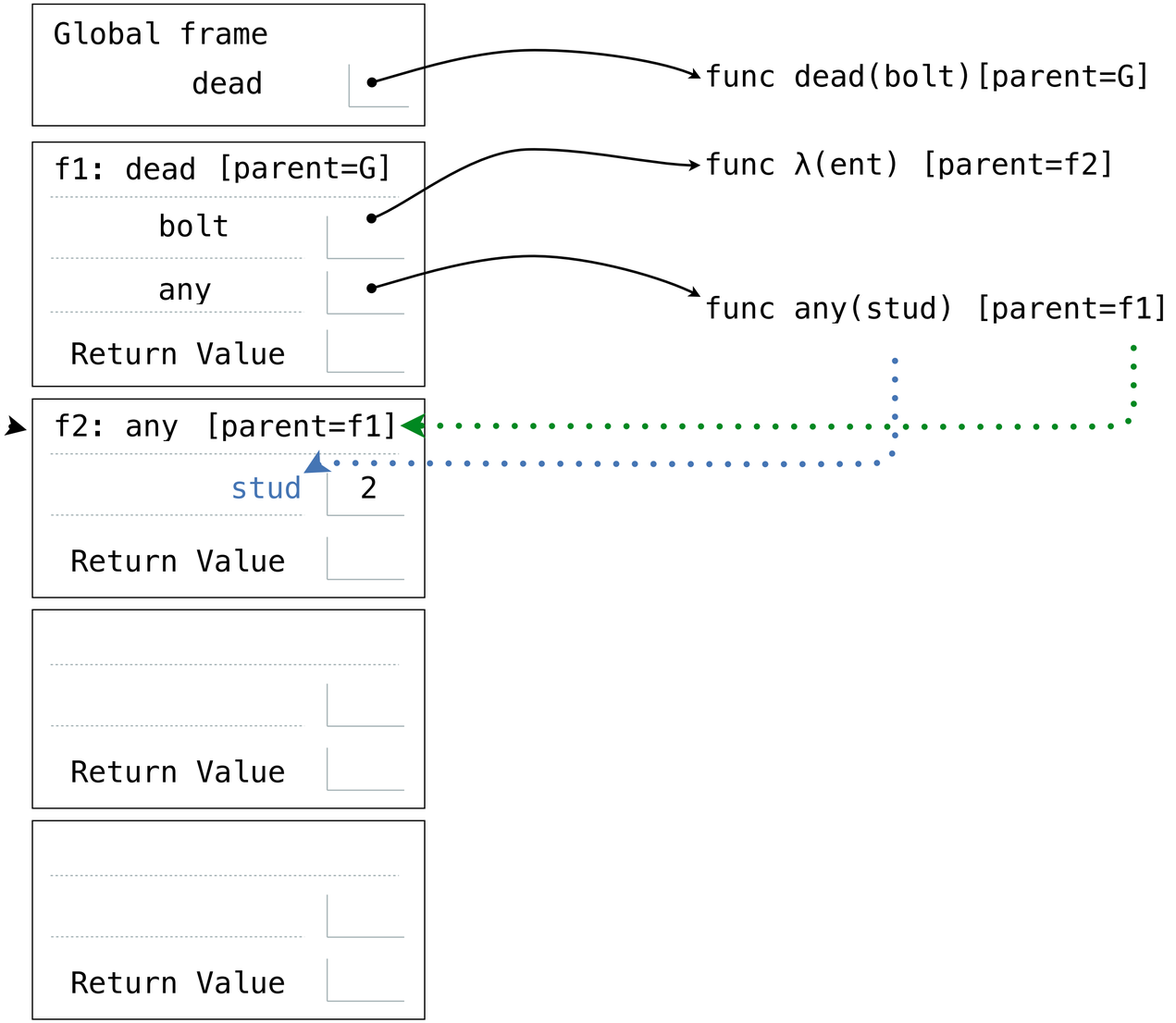
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



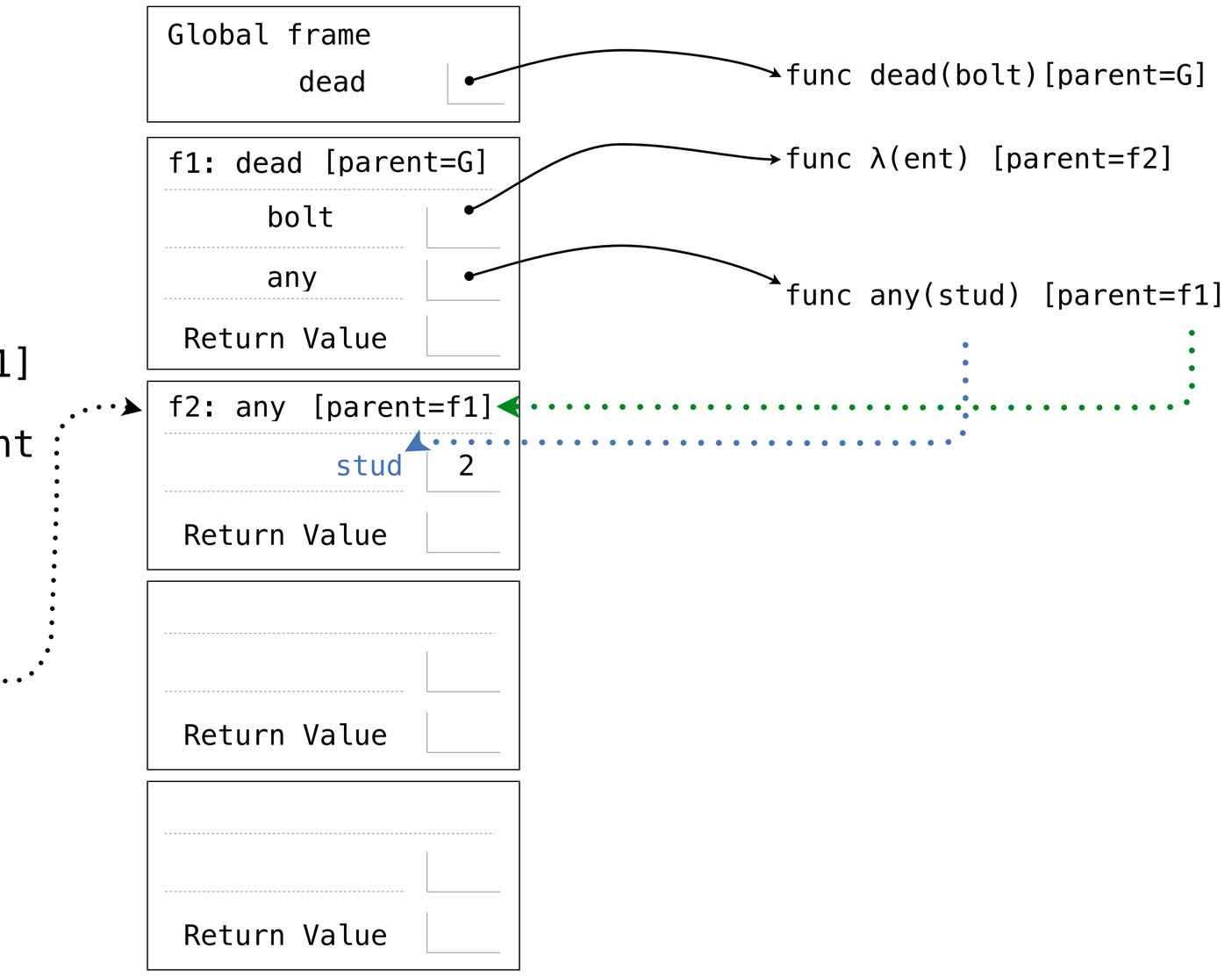
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



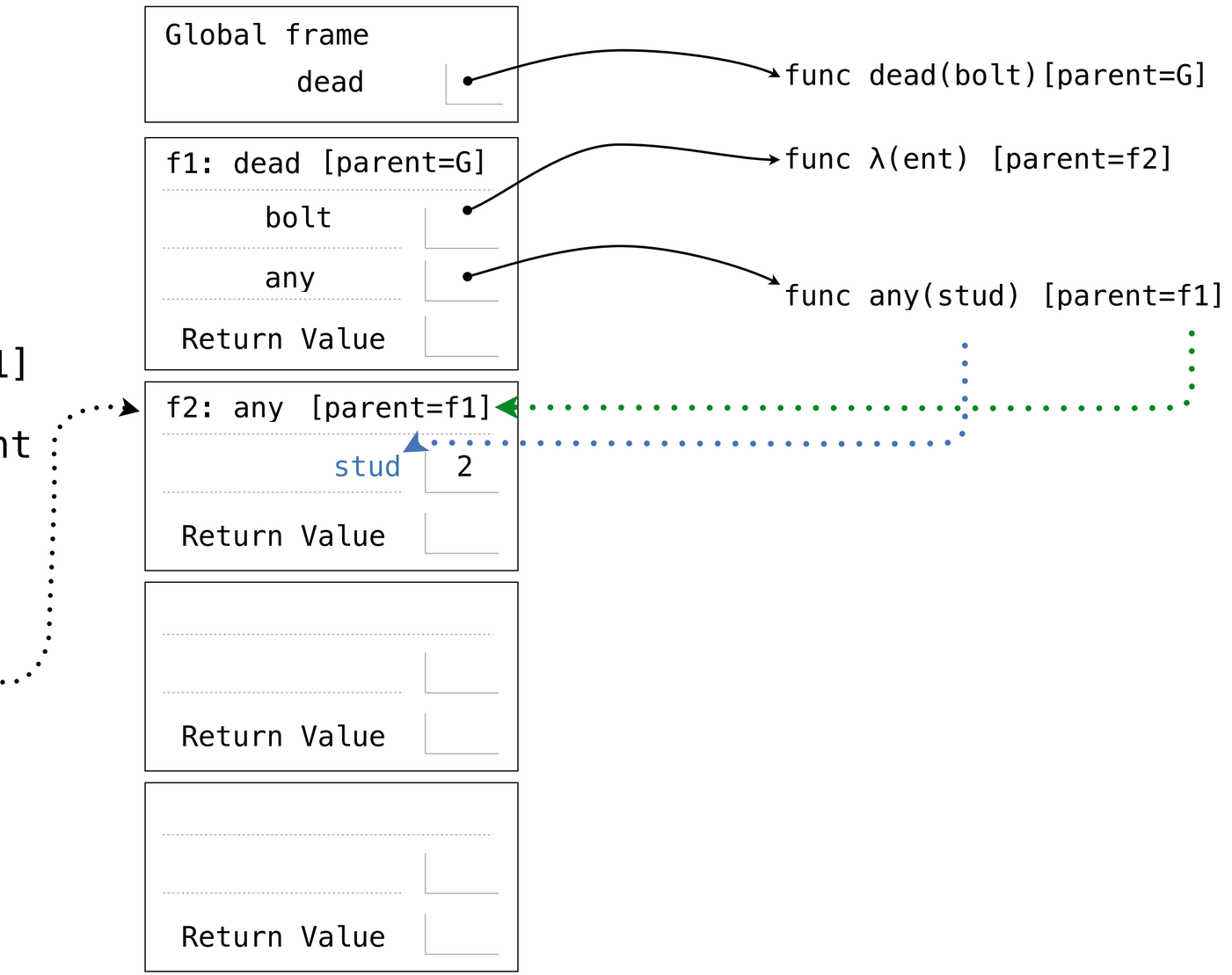
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



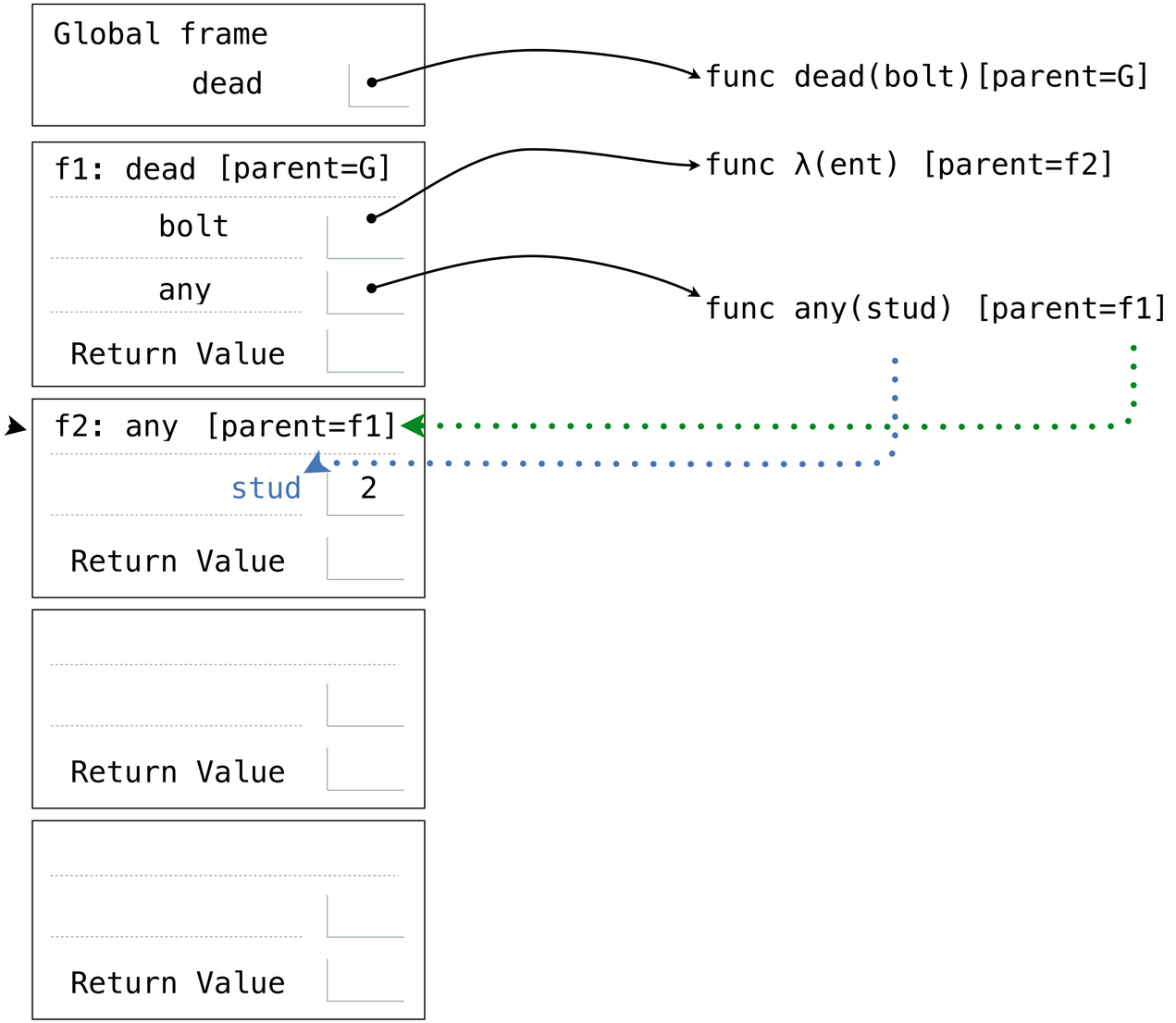
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



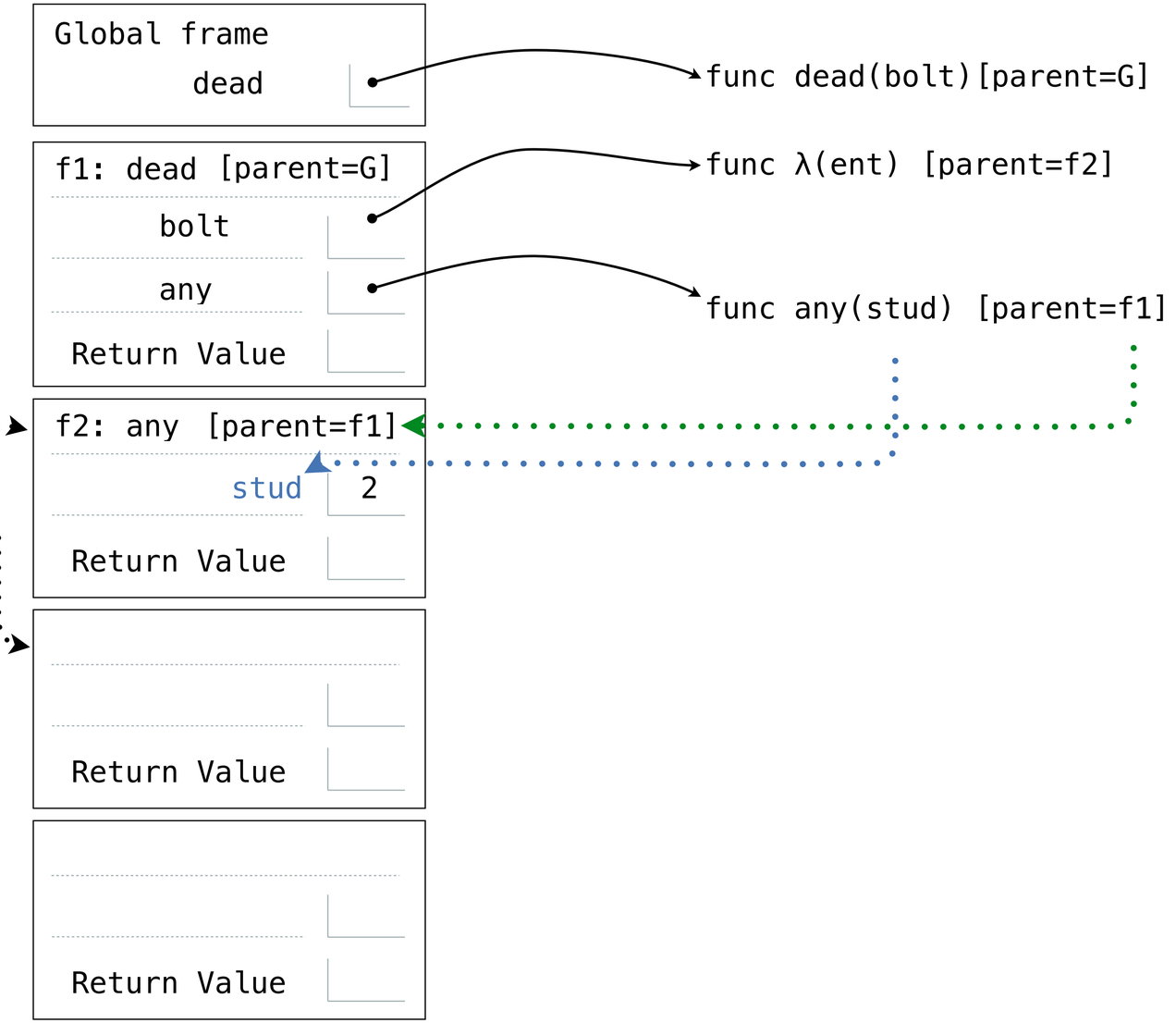
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



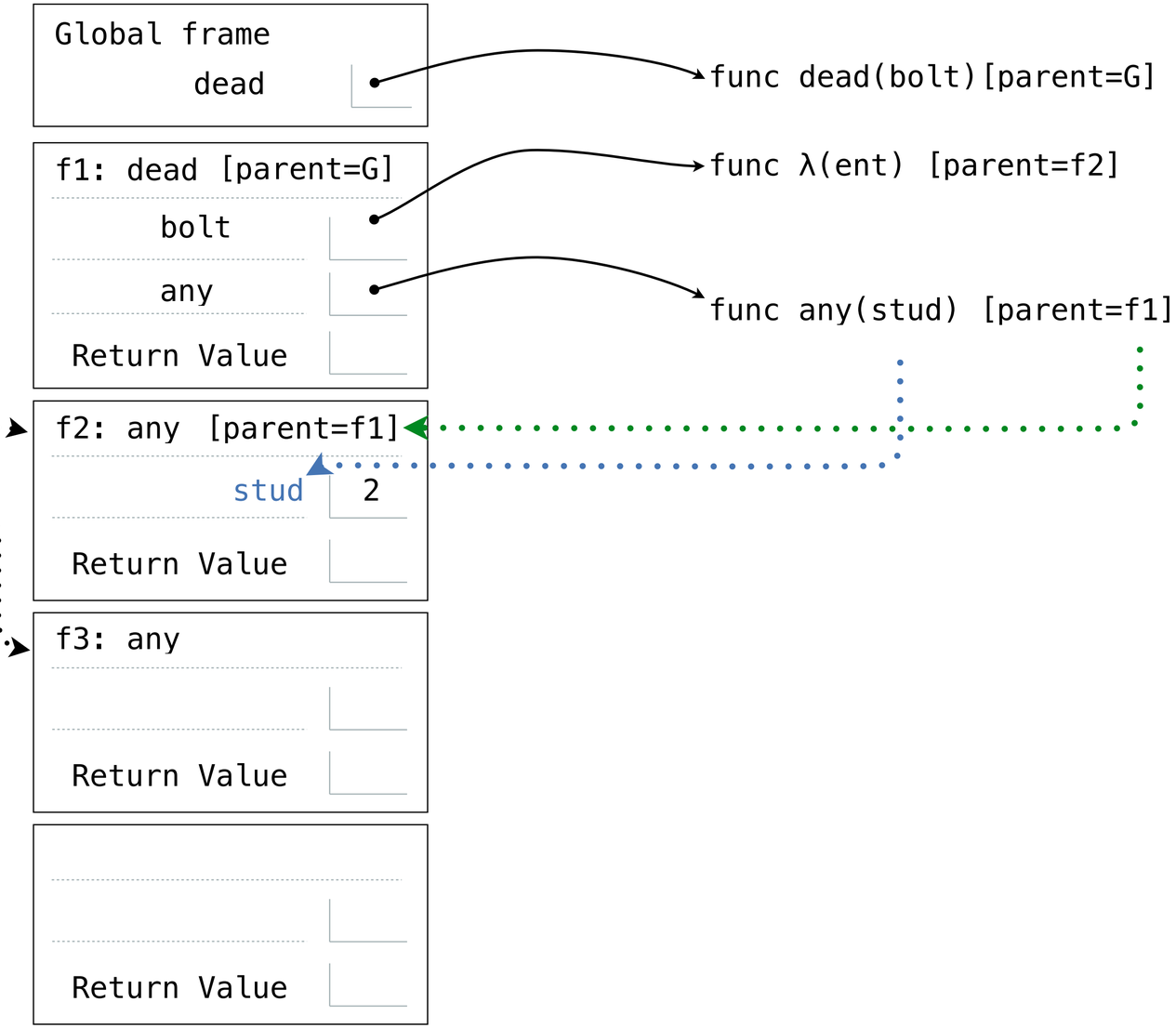
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



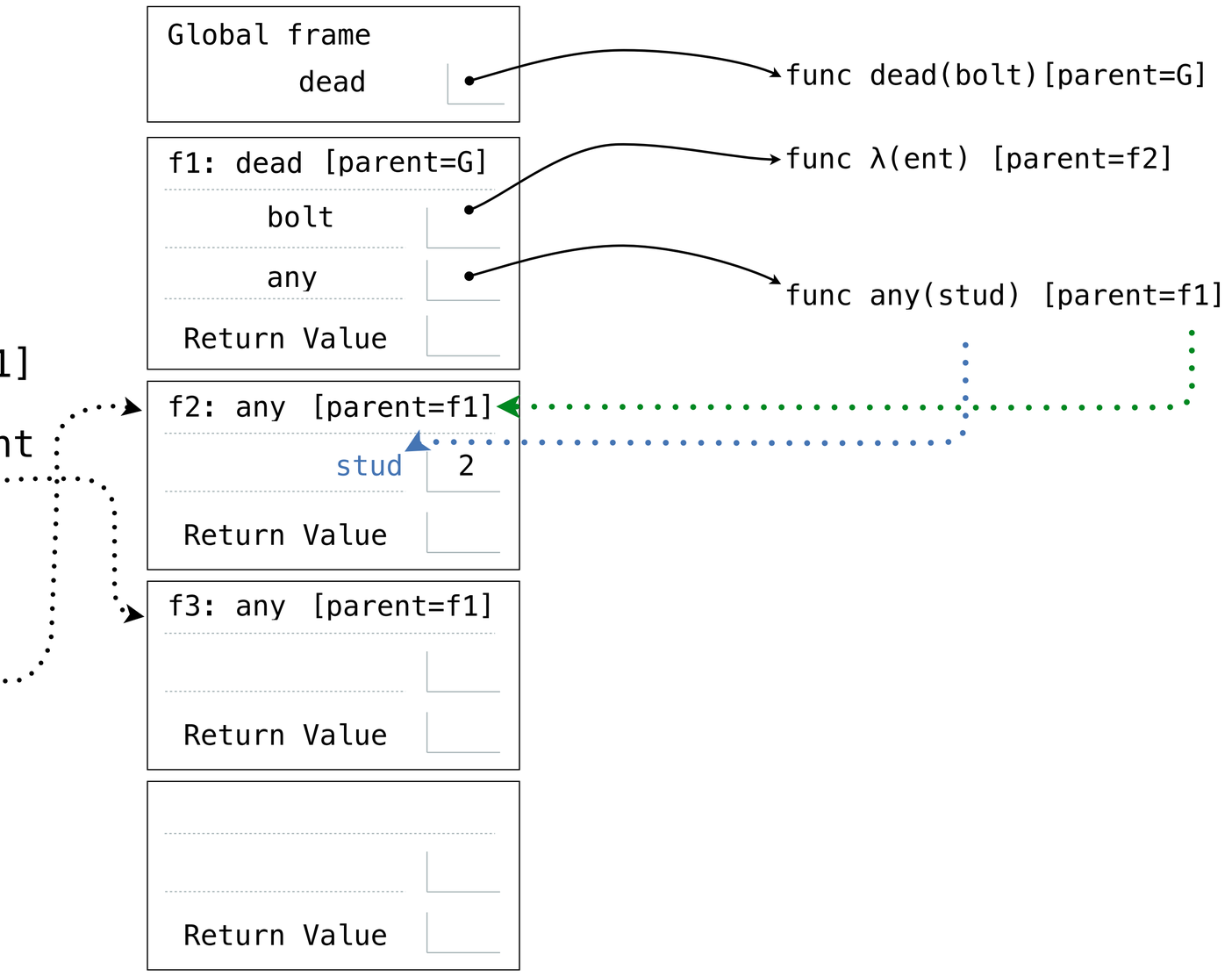
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



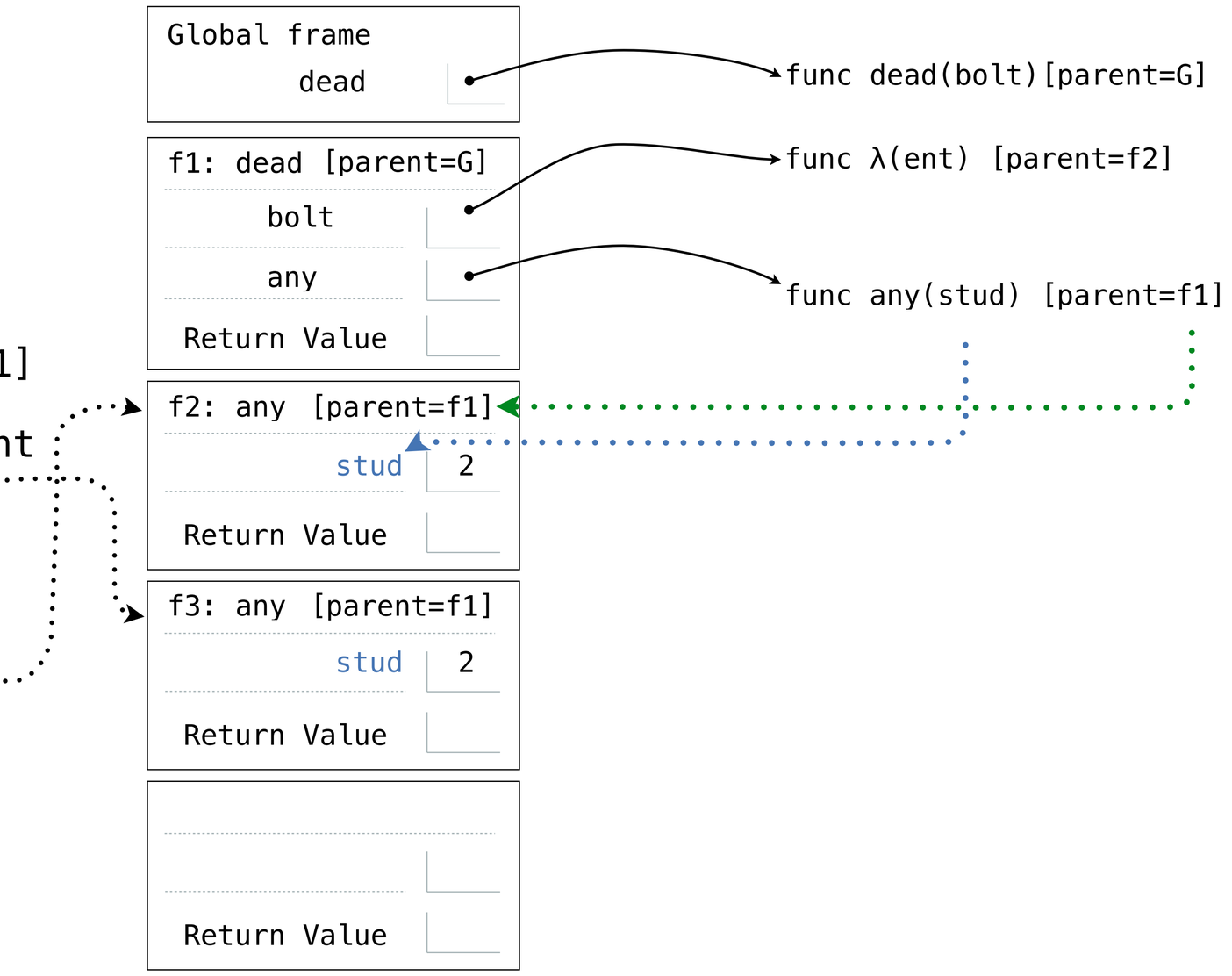
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



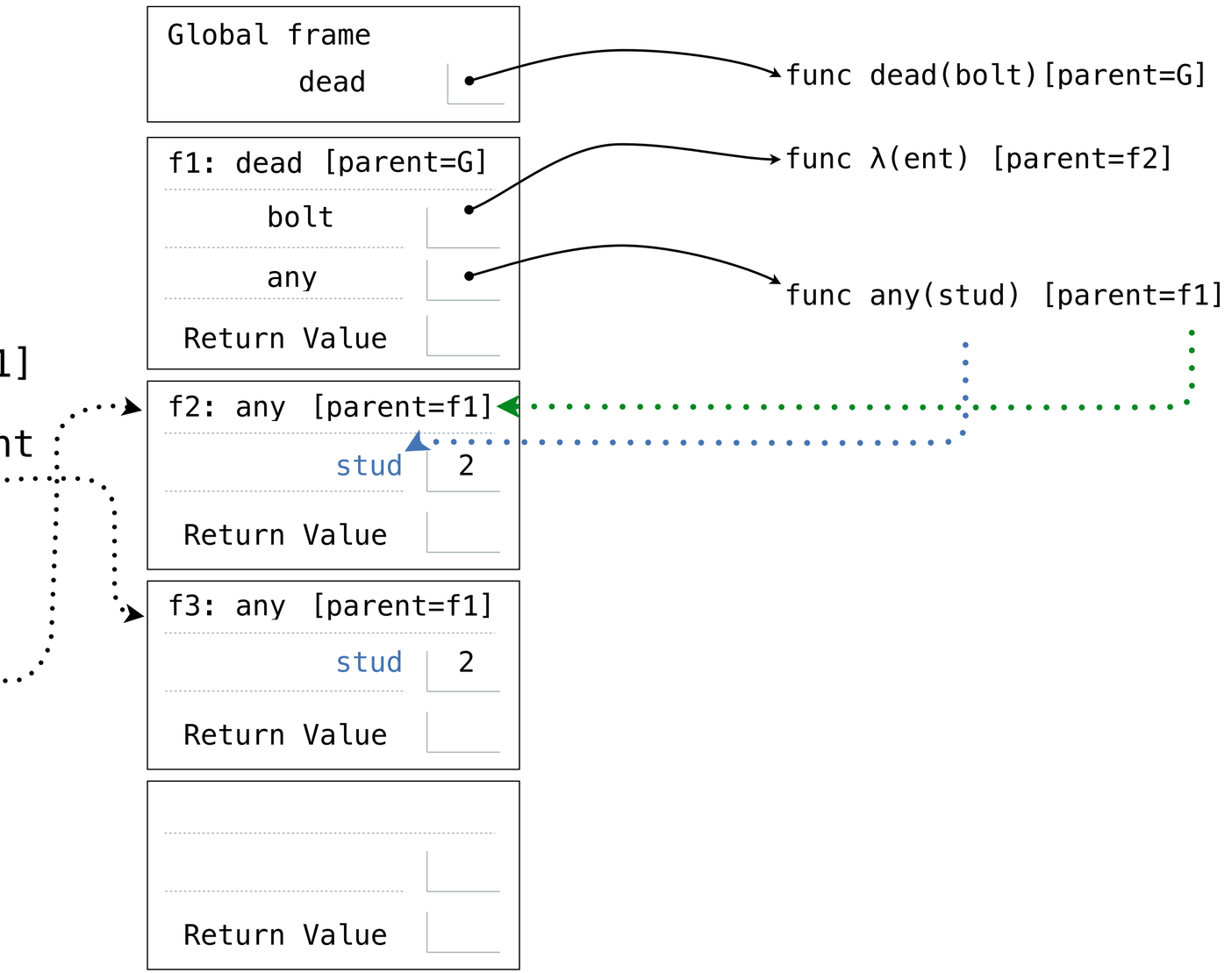
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



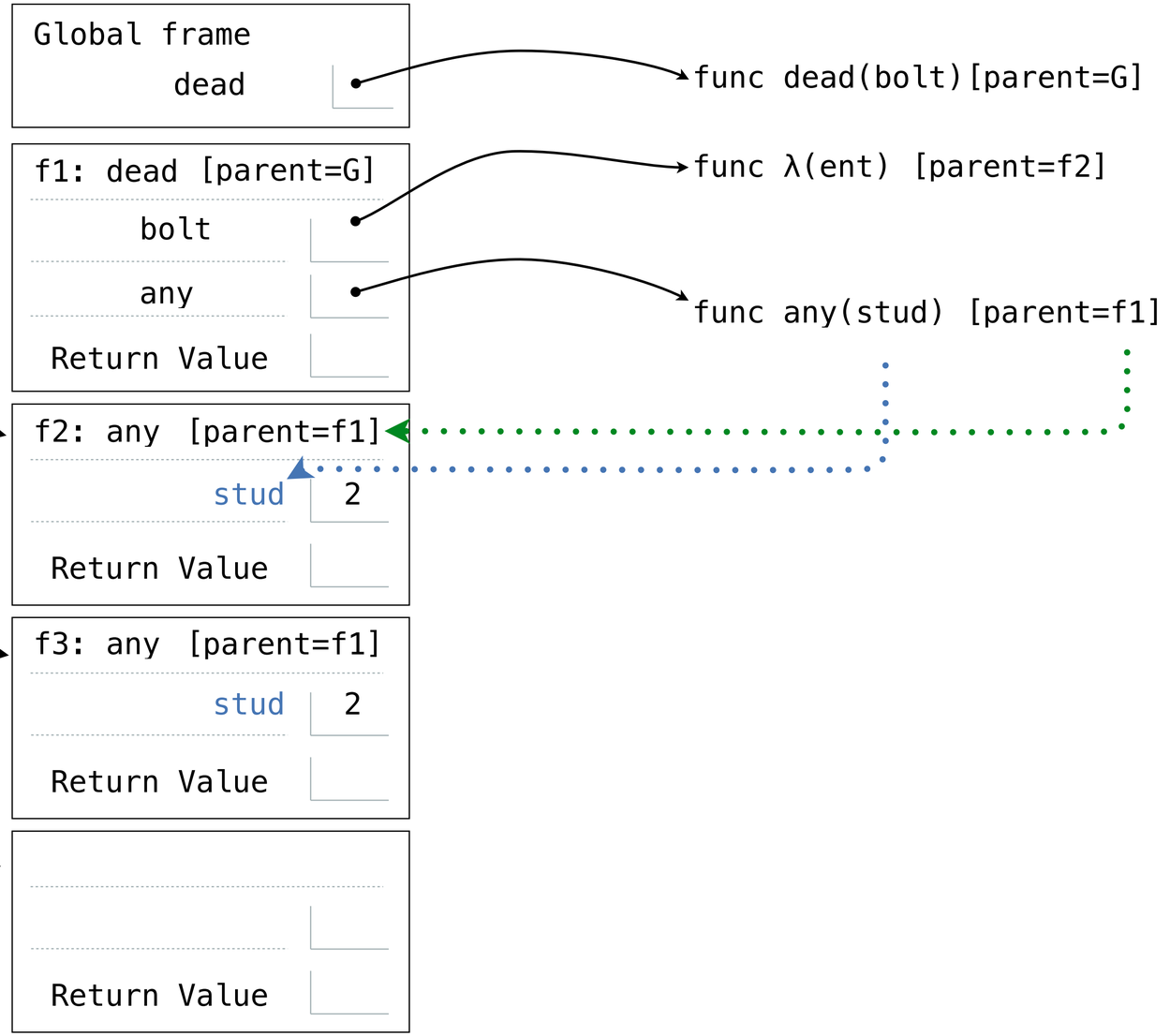
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



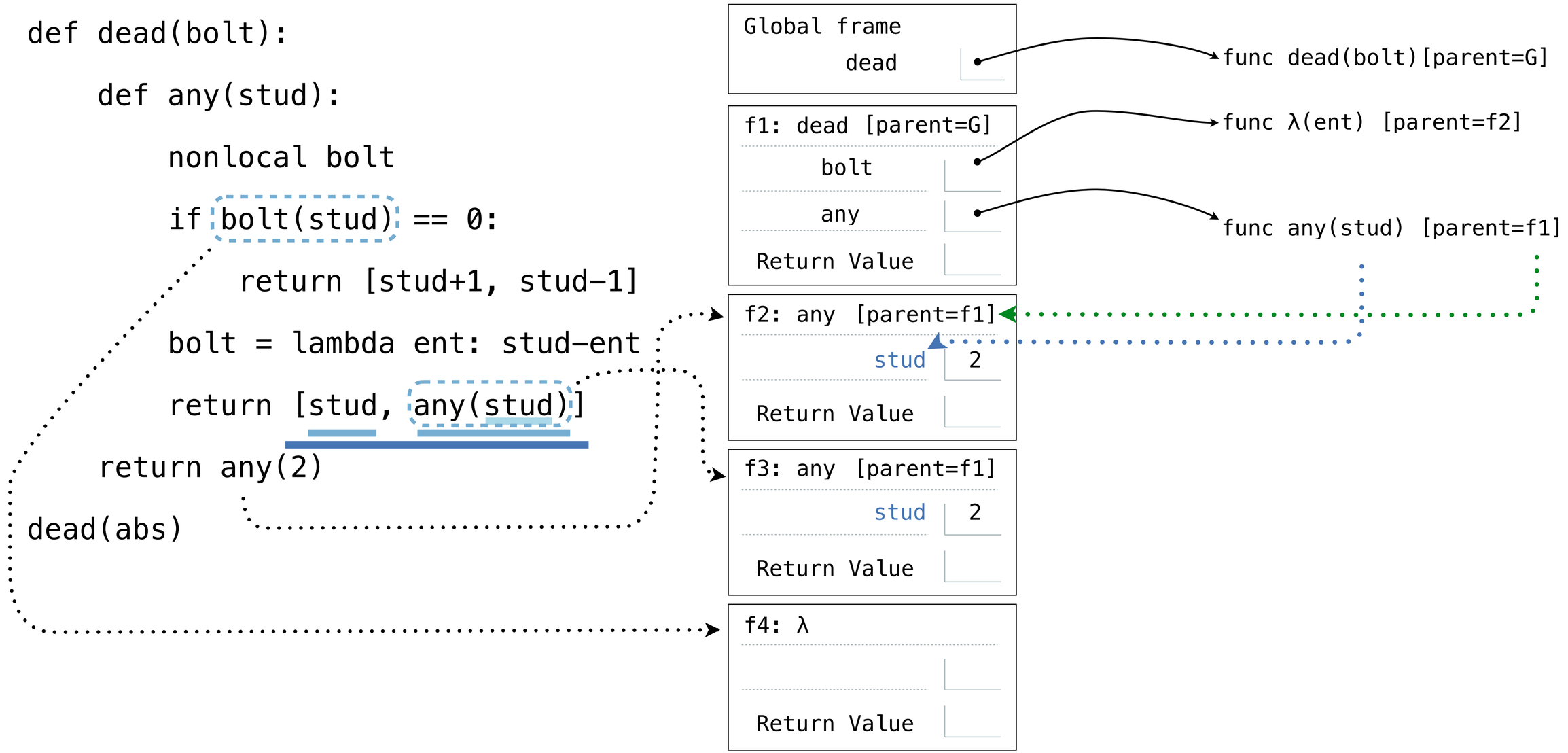
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



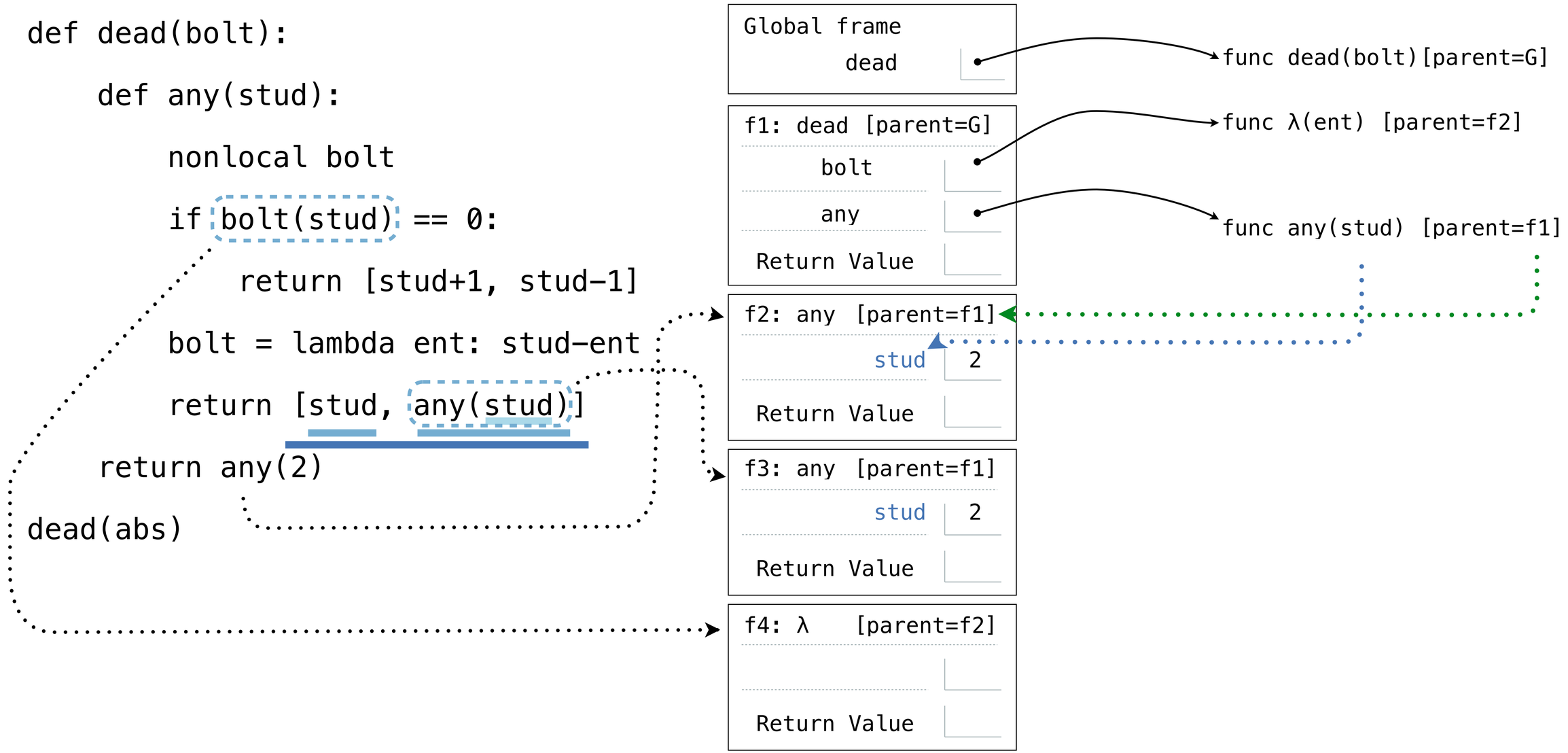
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



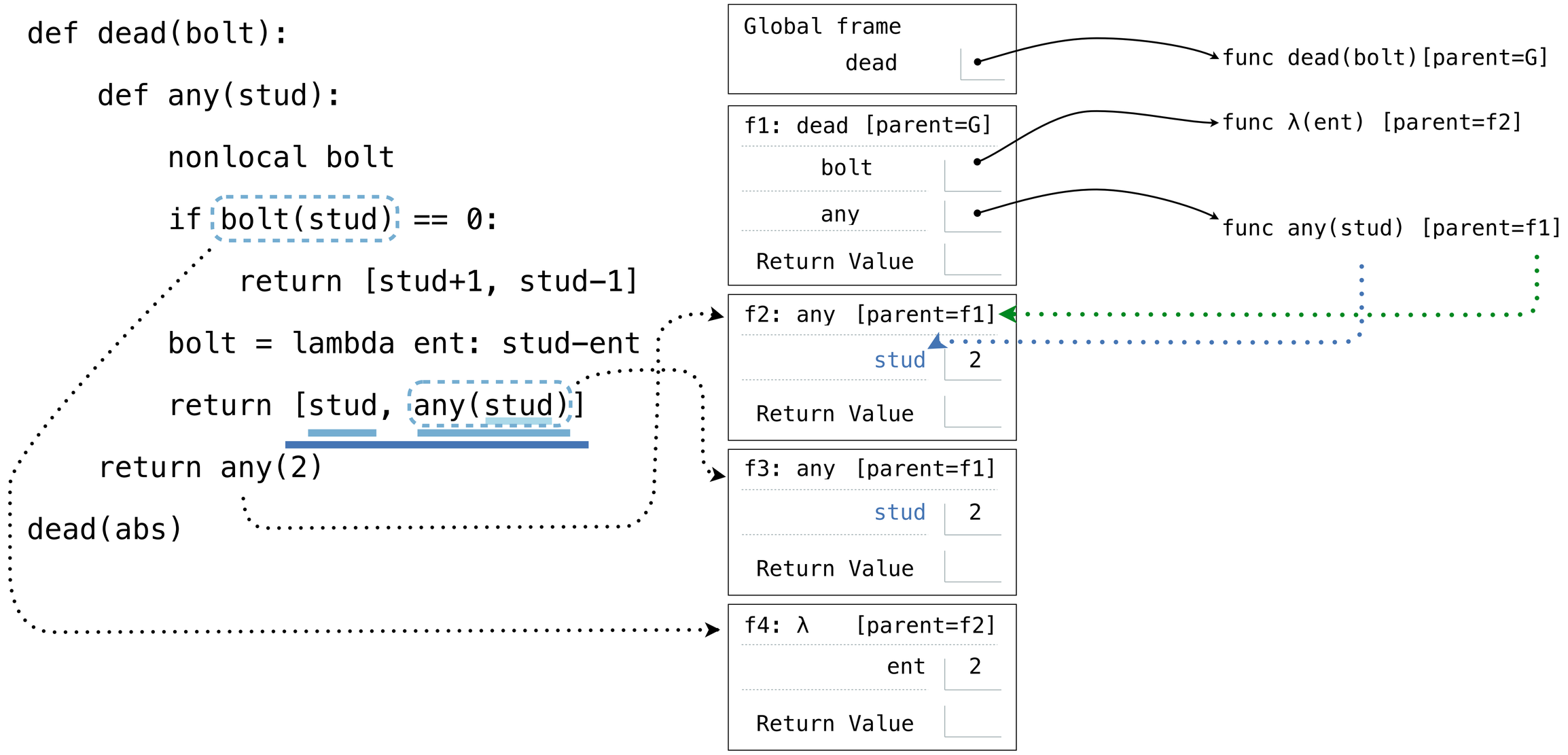
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



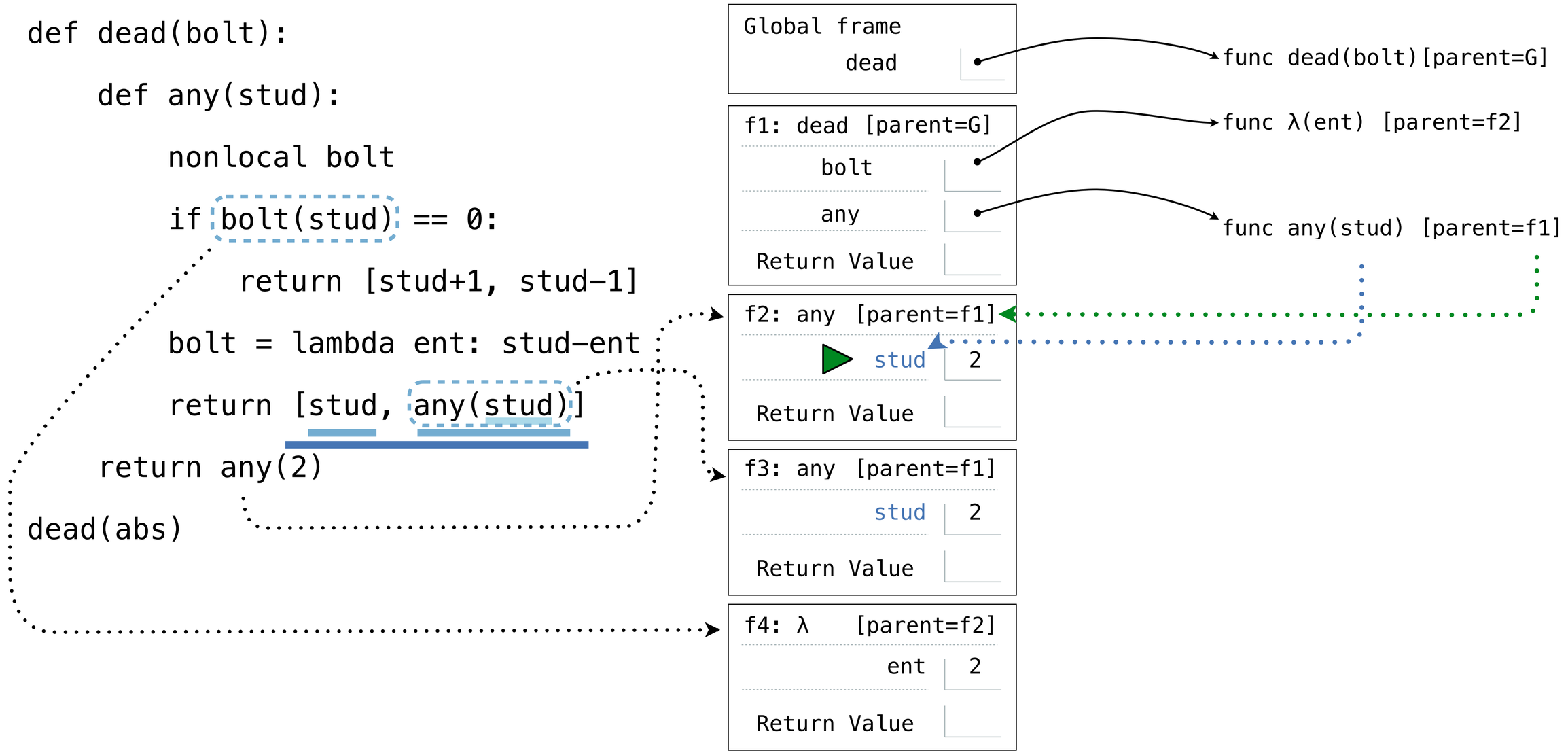
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



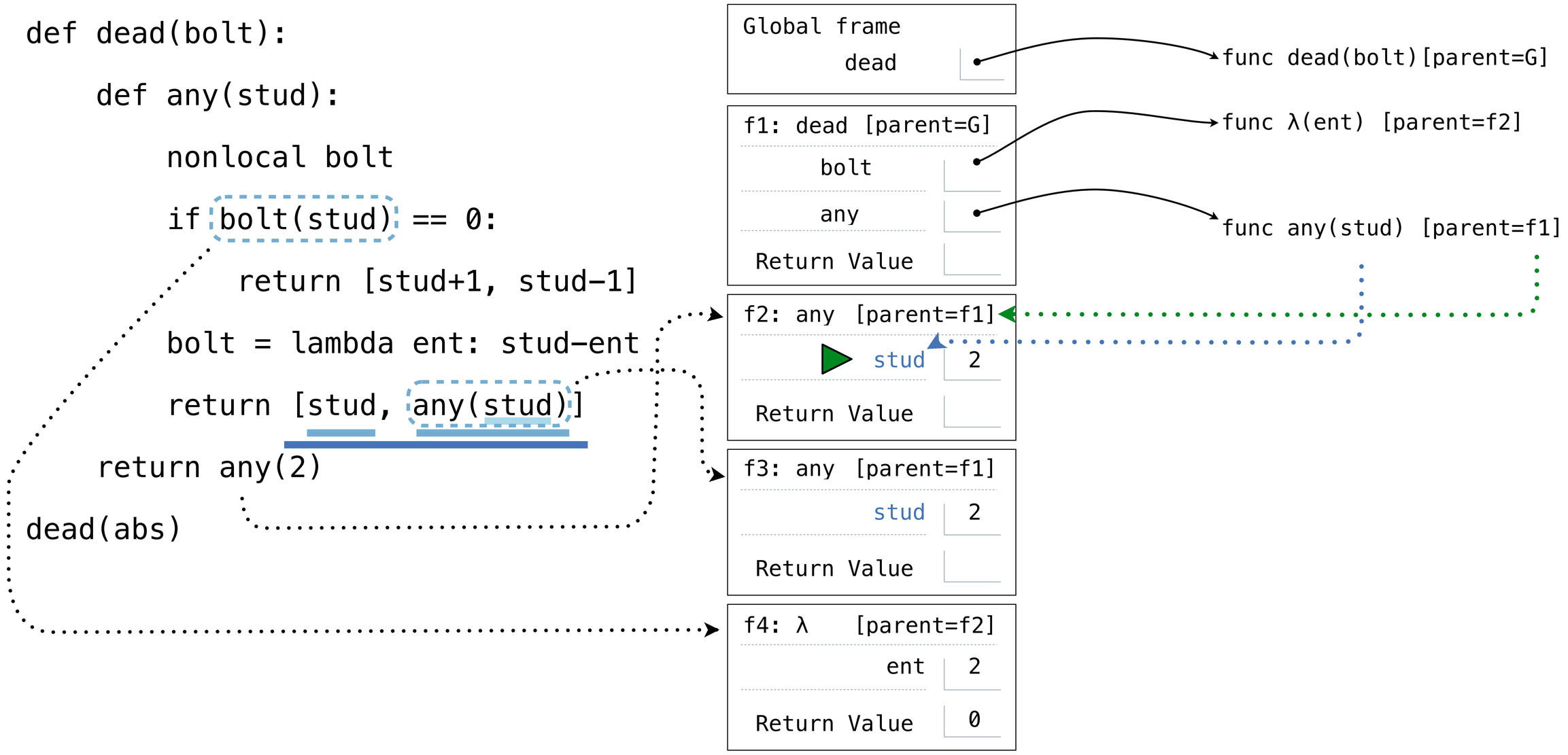
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



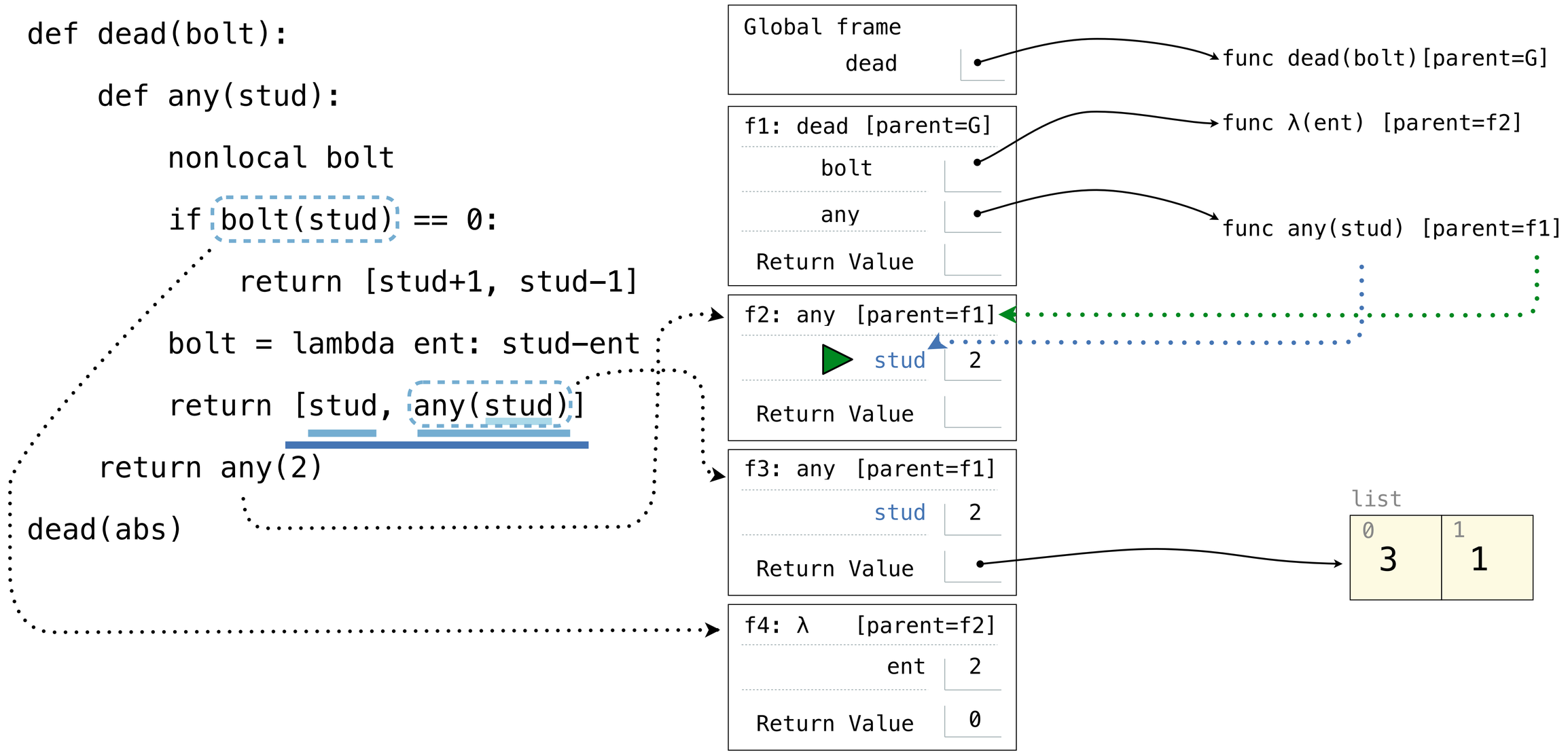
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



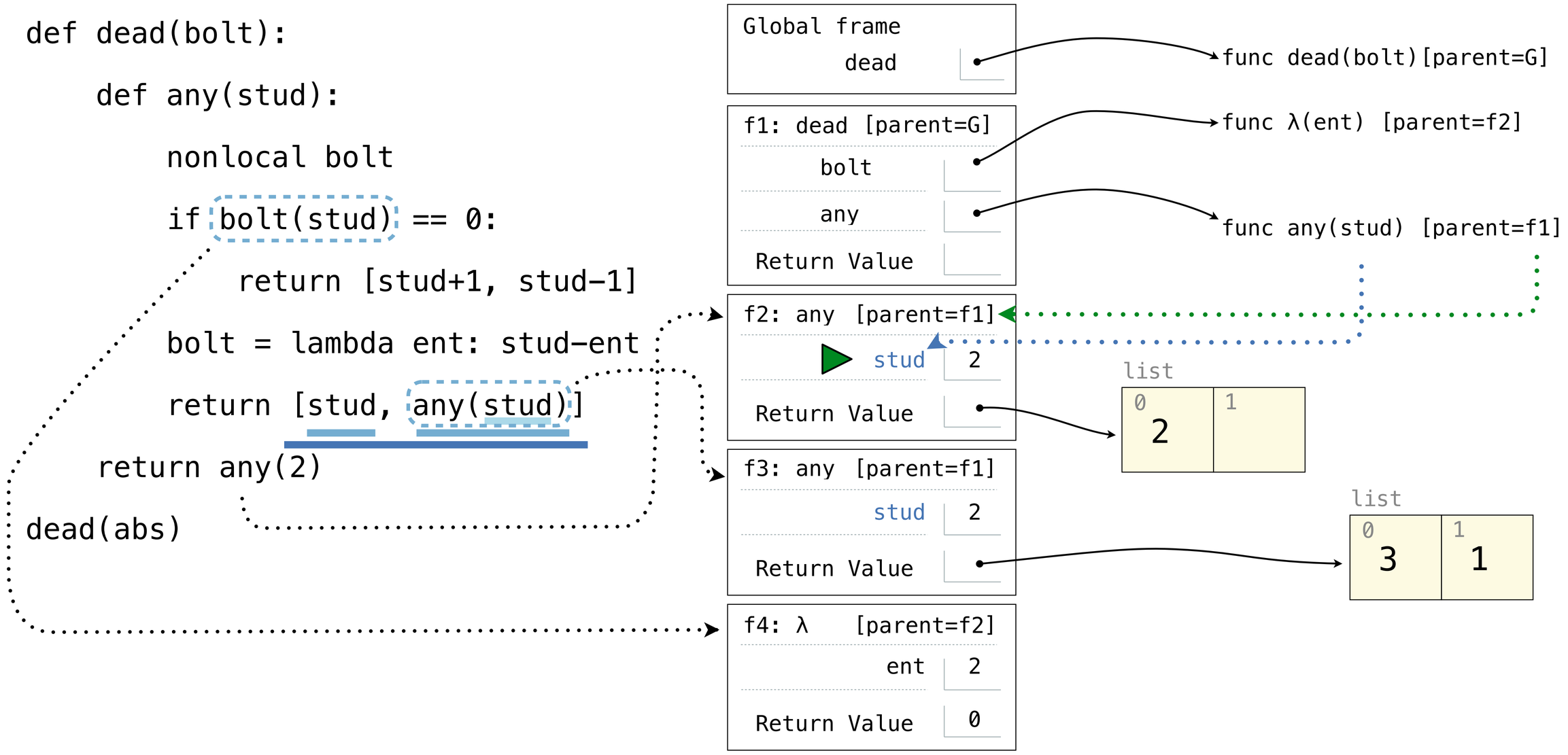
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



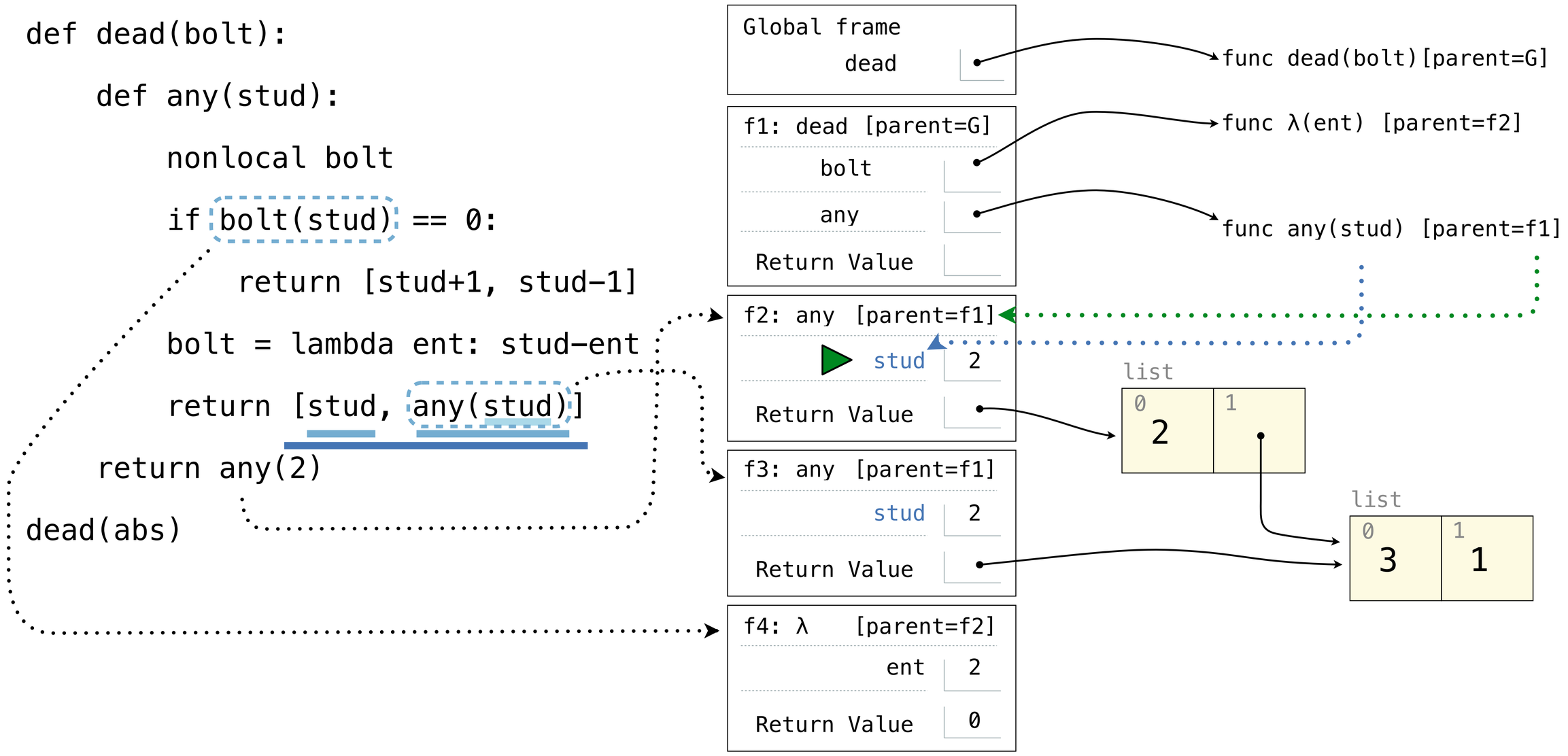
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



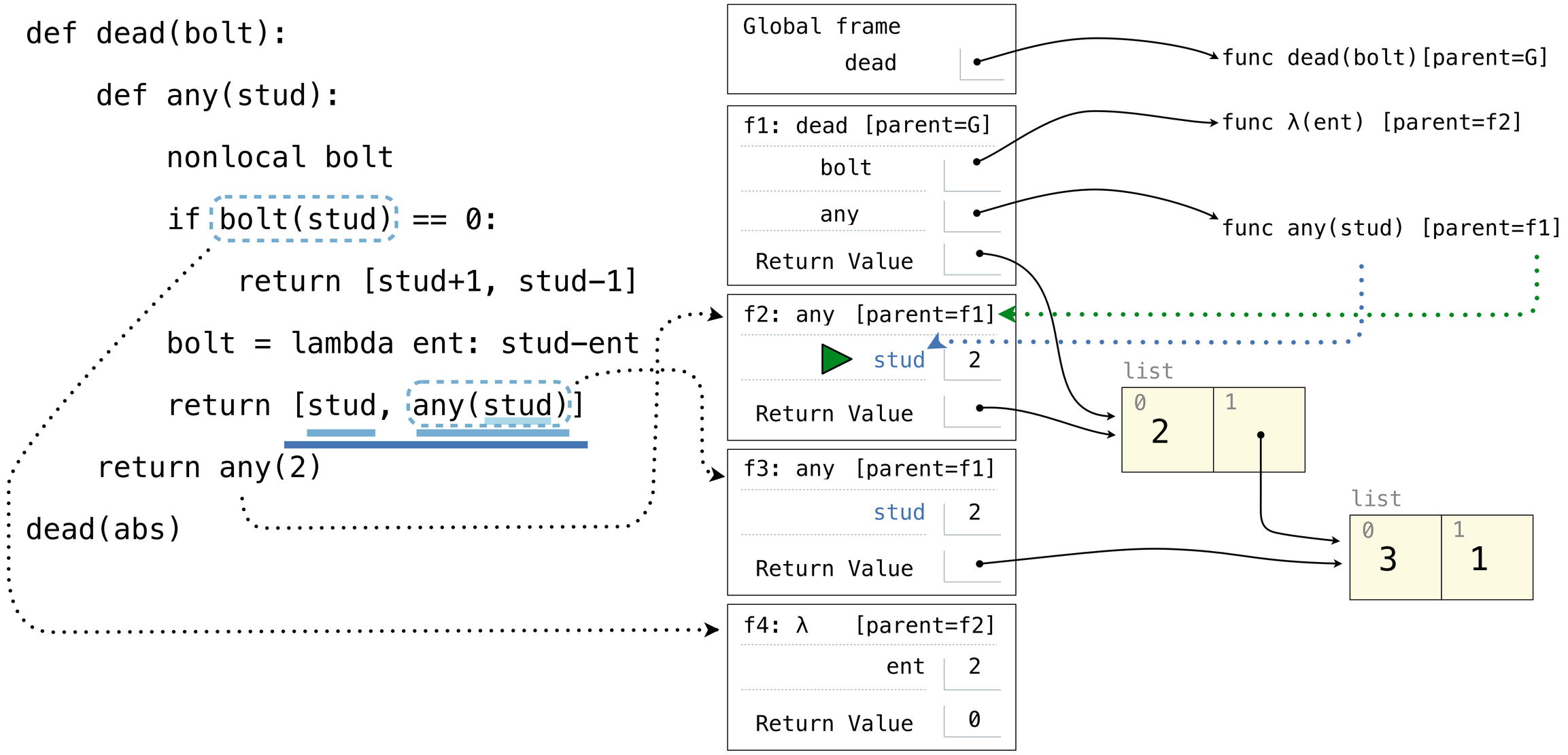
Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```



Вперед!

```
def dead(bolt):
    def any(stud):
        nonlocal bolt
        if bolt(stud) == 0:
            return [stud+1, stud-1]
        bolt = lambda ent: stud-ent
        return [stud, any(stud)]
    return any(2)
dead(abs)
```