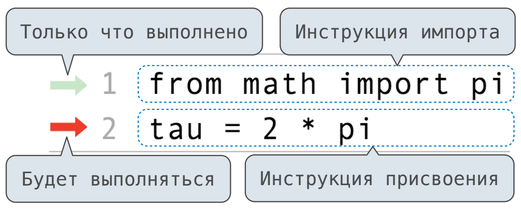
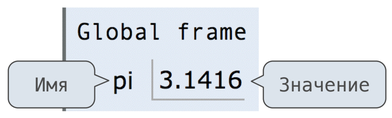
Создание и понимание программ. Шпаргалка №1 — первая страница.





Код (слева):

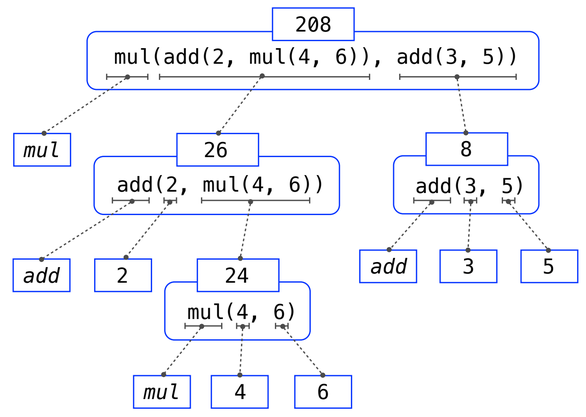
Инструкции и выражения

Стрелки указывают порядок исполнения

Фреймы (справа):

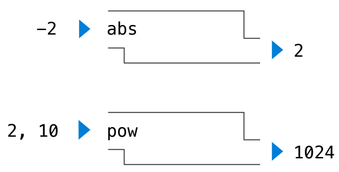
Каждое имя связано со значением

Внутри фрейма имя не может повторяться



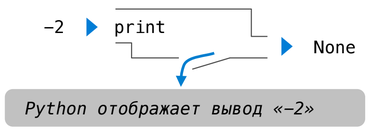
Чистые функции

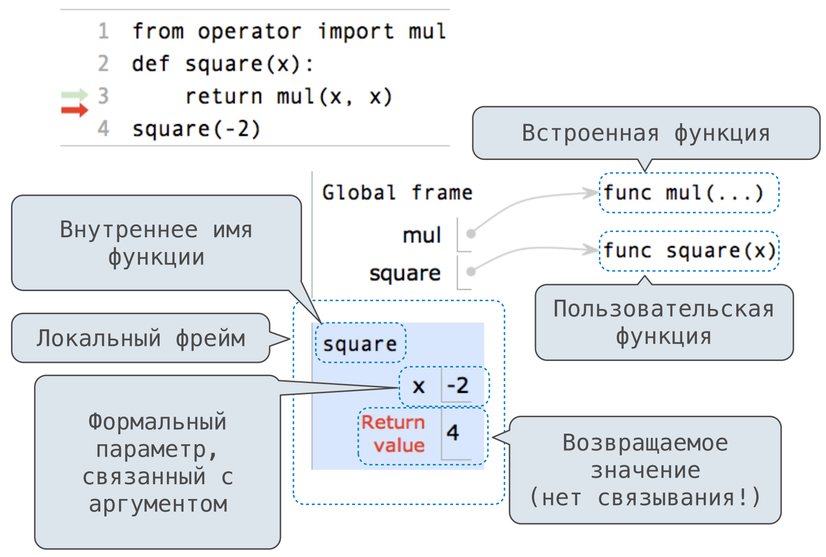
просто возвращают значения



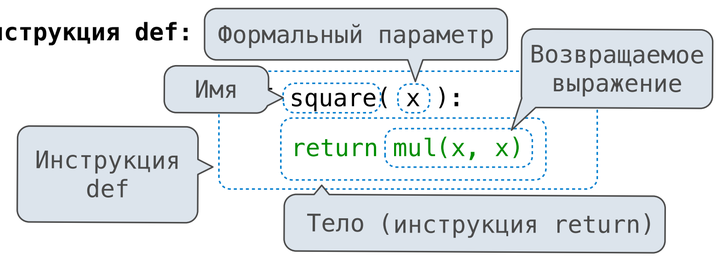
Нечистые функции

имеют побочные эффекты

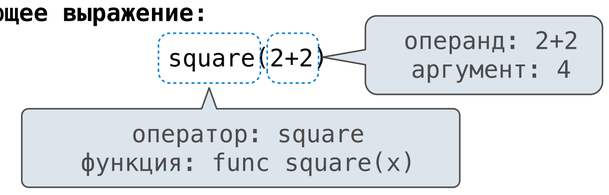




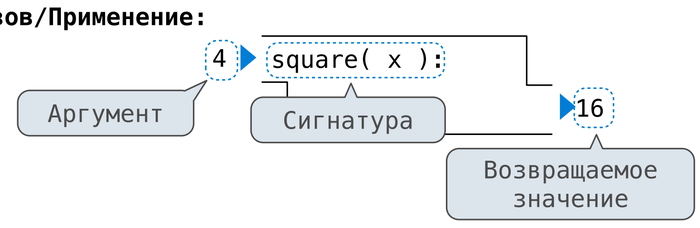
Инструкция def:

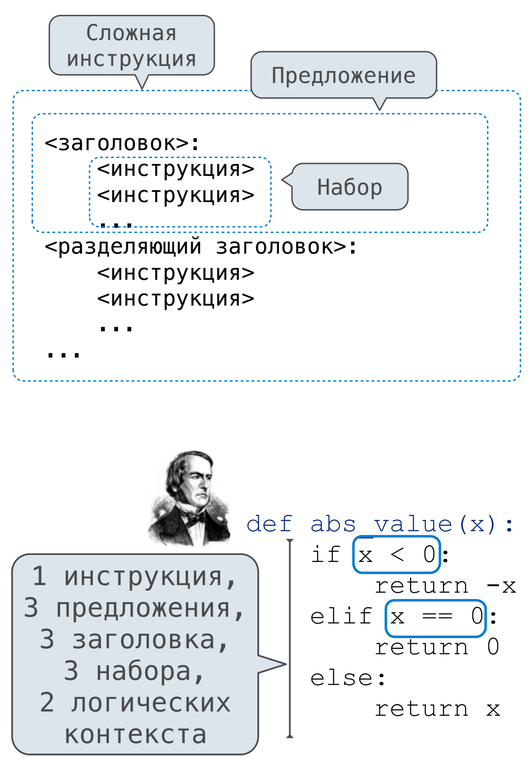


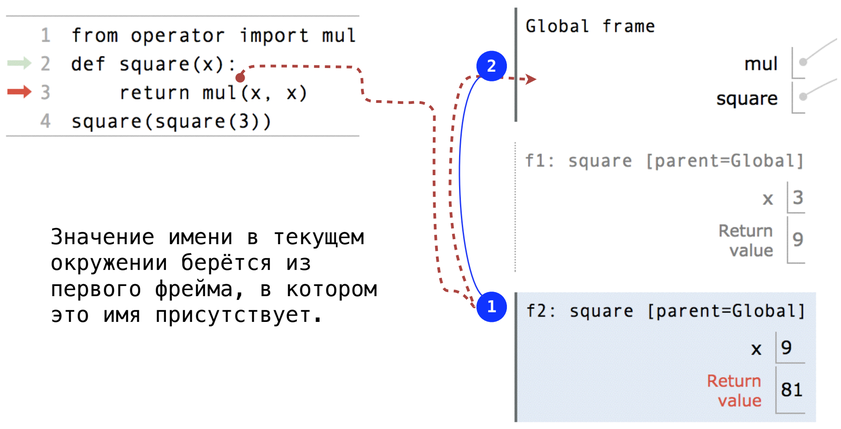
Вызывающее выражение:



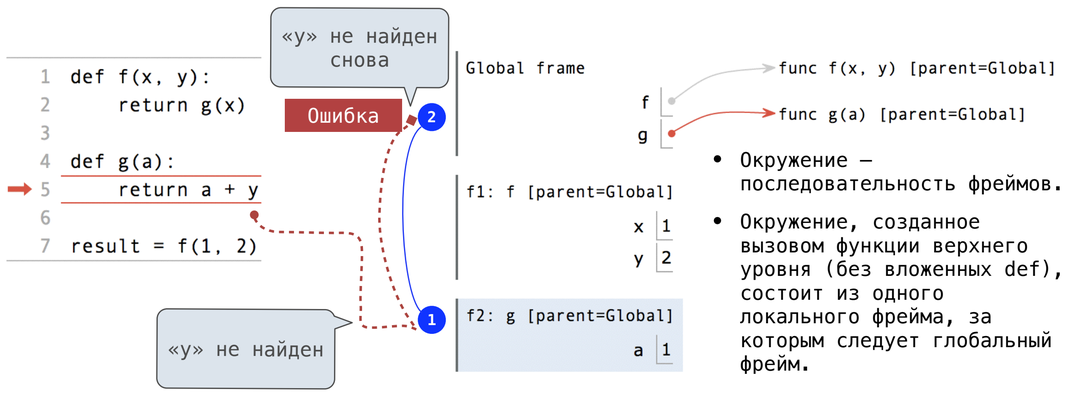
Вызов/Применение:







Значение имени в текущем окружении берётся из первого фрейма, в котором это имя присутствует.



- Окружение — последовательность фреймов.
- Окружение, созданное вызовом функции верхнего уровня (без вложенных def), состоит из одного локального фрейма, за которым следует глобальный фрейм.

Процедура выполнения вызывающего выражения:

- Определить значения подвыражений оператора и операндов.
- Вызвать функцию, являющуюся значением подвыражения оператора, с аргументами равными значениям подвыражений операндов.

Выполнение инструкции def:

- Создать функцию с сигнатурой:
  <имя>(<формальные параметры>)
- Определить тело функции по отступам строк.
- В текущем фрейме связать <имя> с созданной функцией.

Правила выполнения инструкций присвоения:

- Вычислить значения всех выражений справа от = слева направо.
- Связать в текущем фрейме все имена слева от = с полученными значениями.

Процедура вызова/применения пользовательской функции:

- Создать новый локальный фрейм, сформировав новое окружение.
- Связать в этом фрейме формальные параметры функции с аргументами.
- Выполнить тело функции в новом окружении.

Правило выполнения для условных инструкций if:

Каждое предложение рассматривается по порядку.

- Вычислить выражение в заголовке.
- Если оно истинно, выполнить первый набор и пропустить остальные предложения.

Правило выполнения выражения and:

- Выполнить левое подвыражение.
- Если результат является неистинным, то вернуть этот результат.
- В противном случае вернуть результат правого подвыражения.

Правило выполнения выражения or:

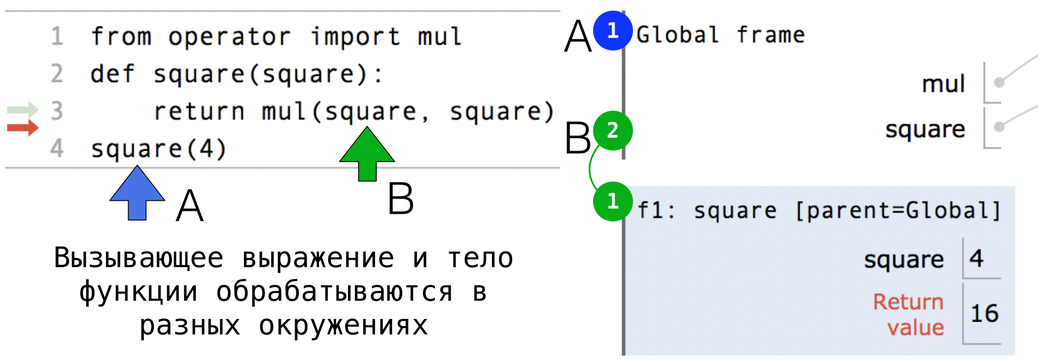
- Выполнить левое подвыражение.
- Если результат является истинным, то вернуть этот результат.
- В противном случае вернуть результат правого подвыражения.

Правило выполнения выражения not:

- Если результат подвыражения истинный, то вернуть неистинный, в противном случае вернуть истинный.

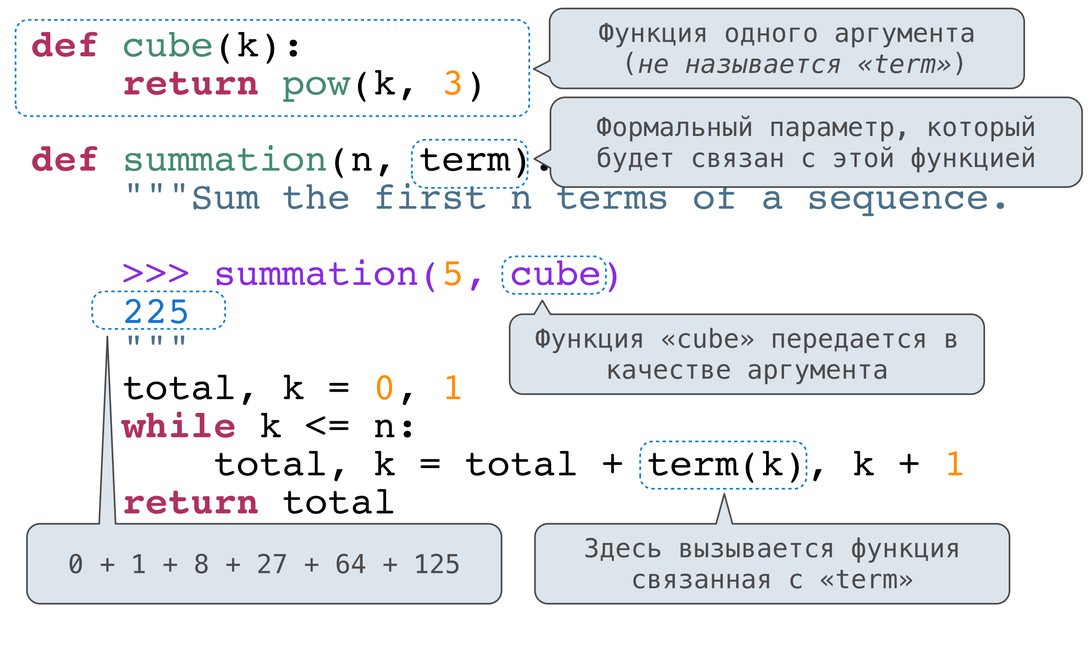
Правило выполнения инструкции while:

- Вычислить выражение в заголовке.
- Если оно истинно, выполнить (весь) набор, затем вернуться к шагу 1.



Вызывающее выражение и тело функции обрабатываются в разных окружениях

```
def fib(n):
    """Вычисляет n-ое число Фибоначчи, для n >= 1."""
    pred, curr = 0, 1   # нулевое и первое числа Фибоначчи
    k = 1               # curr — k-ое число Фибоначчи
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```



Функция высшего порядка — функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

Вложенная инструкция def — функция заданная внутри тела другой функции связывается с именем в локальном фрейме.

При задании функции:

Создается функция: func <имя>(<формальные параметры>) [parent=<метка>]

Её родителем является текущий фрейм.

f1: make_adder func adder(k) [parent=f1]

Происходит связывание <имени> с функцией в текущем фрейме

При вызове функции:

1. Добавляется локальный фрейм, озаглавленный <именем> вызванной функции.
2. Родитель функции копируется в локальный фрейм: [parent=<метка>]
3. <Формальные параметры> связываются с аргументами в локальном фрейме.
4. В окружении, начинающемся с локального фрейма, выполняется тело функции.

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

- Обе создают функцию с одинаковыми поведением, областями определения и значения.
- Обе функции считают родительским фрейм, в котором они определены.
- Обе функции связаны с именем «square».
- Только инструкция «def» назначает функции внутреннее имя.

```
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

Выражение: это приводится к значению

Тоже выражение: приводится к функции

Важно: нет ключевого слова «return»!

Функция с формальным параметром x, которая возвращает значение «x * x»

Должно быть простым выражением

```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.
    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

Функция возвращающая функцию

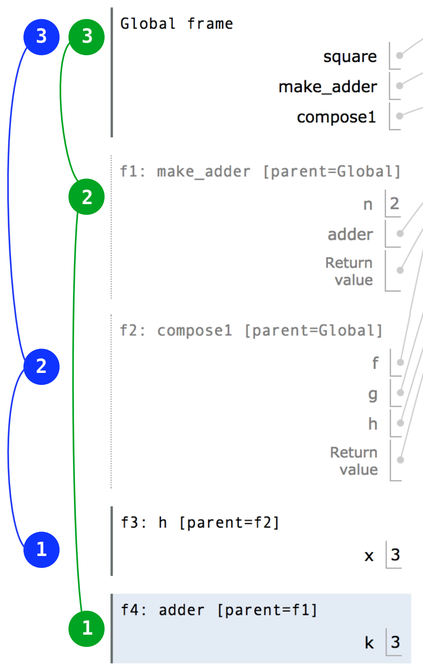
Имя «add_three» связано с функцией

Инструкция def внутри другой инструкции def

Можно обращаться к именам, определенным в содержащей функции

```
1  def square(x):
2      return x * x
3
4  def make_adder(n):
5      def adder(k):
6          return k + n
7      return adder
8
9  def compose1(f, g):
10     def h(x):
11         return f(g(x))
12     return h
13
14 compose1(square, make_adder(2))(3)
```

Возвращаемое значение make_adder — это аргумент для compose1



Структура рекурсивной функции

- Заголовок инструкции def как и в других функциях
- Инструкция ветвления отделяет простые случаи
- Простые случаи выполняются без рекурсивных вызовов
- Рекурсивные случаи выполняются с рекурсивными вызовами

```
def sum_digits(n):
    """Возвращает сумму цифр положительного целого n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

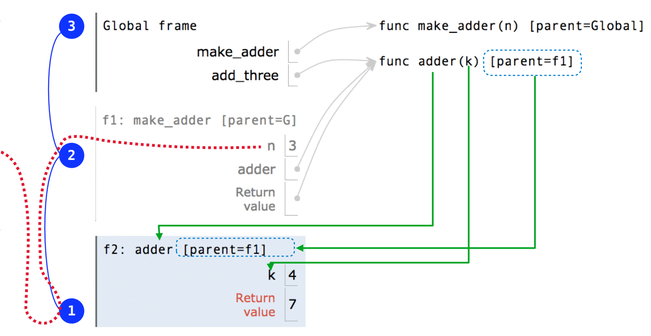
```
def curry2(f):
    """Возвращает функцию g, такую что g(x)(y) возвращает f(x, y).

    >>> from operator import add
    >>> add_three = curry2(add)(3)
    >>> add_three(4)
    7
    """
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

Каррирование:
преобразование функции от многих аргументов в функцию высшего порядка, берущую свои аргументы по одному.

Вложенный def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```

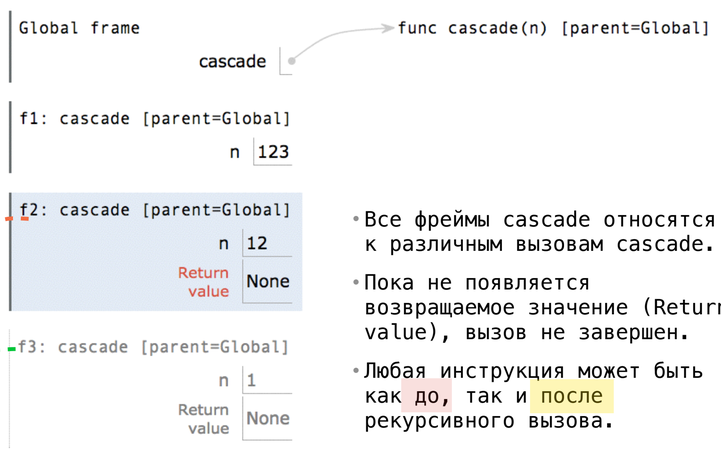


- Каждая пользовательская функция имеет родительский фрейм (часто глобальный)
- Родитель функции — фрейм, в котором задана функция
- Каждый локальный фрейм имеет родительский фрейм (часто глобальный)
- Родитель фрейма — родитель вызванной функции

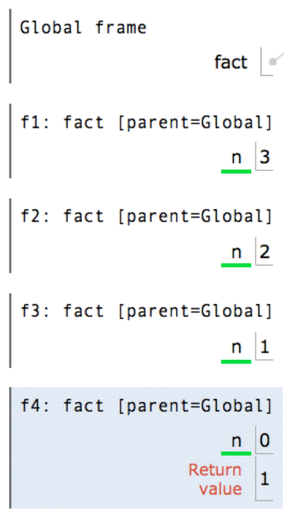
```
1  def cascade(n):
2      if n < 10:
3          print(n)
4      else:
5          print(n)
6          cascade(n//10)
7          print(n)
8
9  cascade(123)
```

Program output:

```
123
12
1
12
```

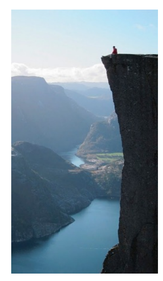


- Все фреймы cascade относятся к различным вызовам cascade.
- Пока не появляется возвращаемое значение (Return value), вызов не завершен.
- Любая инструкция может быть как до, так и после рекурсивного вызова.



Корректна ли функция fact?

1. Проверяем простой случай.
2. Рассматриваем **fact** как функциональную абстракцию!
3. Предполагаем, что fact(n-1) работает правильно.
4. Удостоверяемся, что fact(n) корректен, в предположении, что значение fact(n-1) правильно.



```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

```
1       def inverse_cascade(n):
12          grow(n)
123         print(n)
1234        shrink(n)
123
12      def f_then_g(f, g, n):
1           if n:
                f(n)
                g(n)

grow =   lambda n: f_then_g(grow,  print,  n//10)
shrink = lambda n: f_then_g(print, shrink, n//10)
```

n: 0, 1, 2, 3, 4, 5, 6, 7, 8

fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

- Рекурсивная декомпозиция: поиск более простых случаев.
- Исследуем две возможности:
  - Хотя бы одна часть рана 4
  - Нет ни одной части равной 4
- Решаем две более простые задачи:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Древовидная рекурсия зачастую предполагает исследование различных вариантов.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```