

**Import statement**

```
1 from math import pi
2 tau = 2 * pi
```

**Assignment statement**

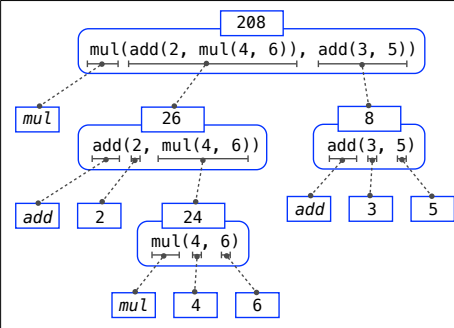
**Global frame**

Name	Value
pi	3.1416

**Binding**

**Code (left):** Statements and expressions  
Red arrow points to next line. Gray arrow points to the line just executed

**Frames (right):** A name is bound to a value  
In a frame, there is at most one binding per name



**Pure Functions**

```
-2 > abs(number): 2
2, 10 > pow(x, y): 1024
```

**Non-Pure Functions**

```
-2 > print(...): None
```

display "-2"

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```

**Global frame**

**Intrinsic name of function called**

mul

square

**Local frame**

f1: square [parent=Global]

**Formal parameter bound to argument**

x -2

**Return value**

4

**Return value is not a binding!**

**Built-in function**

func mul(...) [parent=Global]

func square(x) [parent=Global]

**User-defined function**

**Defining:**

```
>>> def square(x):
    return mul(x, x)
```

**Def statement**

**Formal parameter**

**Return expression**

**Body (return statement)**

**Call expression:** square(2+2)

**operator:** square

**function:** func square(x)

**operand:** 2+2

**argument:** 4

**Compound statement**

**Clause**

```
<header>:
<statement>
<statement>
```

**Suite**

```
<separating header>:
<statement>
<statement>
...
```

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

**Global frame**

mul

square

**Local frame**

f1: square [parent=Global]

x 3

**Return value**

9

f2: square [parent=Global]

x 9

**Return value**

81

**Calling/Applying:**

```
4 > square(x):
```

**Argument**

**Intrinsic name**

**Return value**

16

```
def abs_value(x):
    1 statement,
    3 clauses,
    3 headers,
    3 suites,
    2 boolean contexts
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

**Evaluation rule for call expressions:**

- Evaluate the operator and operand subexpressions.
- Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

**Applying user-defined functions:**

- Create a new local frame with the same parent as the function that was applied.
- Bind the arguments to the function's formal parameter names in that frame.
- Execute the body of the function in the environment beginning at that frame.

```
1 def f(x, y):
2     return g(x)
3
4 def g(a):
5     return a + y
6
7 result = f(1, 2)
```

**Global frame**

func f(x, y) [parent=Global]

func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

**Error**

"y" is not found

- An environment is a sequence of frames
- An environment for a non-nested function (no def within def) consists of one local frame, followed by the global frame

**Execution rule for def statements:**

- Create a new function value with the specified name, formal parameters, and function body.
- Its parent is the first frame of the current environment.
- Bind the name of the function to the function value in the first frame of the current environment.

**Execution rule for assignment statements:**

- Evaluate the expression(s) on the right of the equal sign.
- Simultaneously bind the names on the left to those values, in the first frame of the current environment.

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(4)
```

**Global frame**

mul

square

**Local frame**

f1: square [parent=Global]

square 4

**Return value**

16

A call expression and the body of the function being called are evaluated in different environments

**Higher-order function:** A function that takes a function as an argument value or returns a function as a return value

**Nested def statements:** Functions defined within other function bodies are bound to names in the local frame

**Execution rule for conditional statements:**

Each clause is considered in order.

- Evaluate the header's expression.
- If it is a true value, execute the suite, then skip the remaining clauses in the statement.

**Evaluation rule for or expressions:**

- Evaluate the subexpression <left>.
- If the result is a true value v, then the expression evaluates to v.
- Otherwise, the expression evaluates to the value of the subexpression <right>.

**Evaluation rule for and expressions:**

- Evaluate the subexpression <left>.
- If the result is a false value v, then the expression evaluates to v.
- Otherwise, the expression evaluates to the value of the subexpression <right>.

**Evaluation rule for not expressions:**

- Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

**Execution rule for while statements:**

- Evaluate the header's expression.
- If it is a true value, execute the (whole) suite, then return to step 1.

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1 # Zeroth and first Fibonacci numbers
    k = 1 # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

**def cube(k):**

**return pow(k, 3)**

**def summation(n, term):**

"""Sum the first n terms of a sequence.

```
>>> summation(5, cube)
225
```

total, k = 0, 1

```
while k <= n:
    total, k = total + term(k), k + 1
return total
```

0 + 1<sup>3</sup> + 2<sup>3</sup> + 3<sup>3</sup> + 4<sup>3</sup> + 5<sup>3</sup>

Function of a single argument (not called term)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

The function bound to term gets called here

```
square = lambda x,y: x * y
```

Evaluates to a function. No "return" keyword!

A function with formal parameters x and y that returns the value of "x \* y"

Must be a single expression

VS

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

- Both create a function with the same domain, range, and behavior.
- Both functions have as their parent the environment in which they were defined.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name.

```
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
```

A function that returns a function

The name add\_three is bound to a function

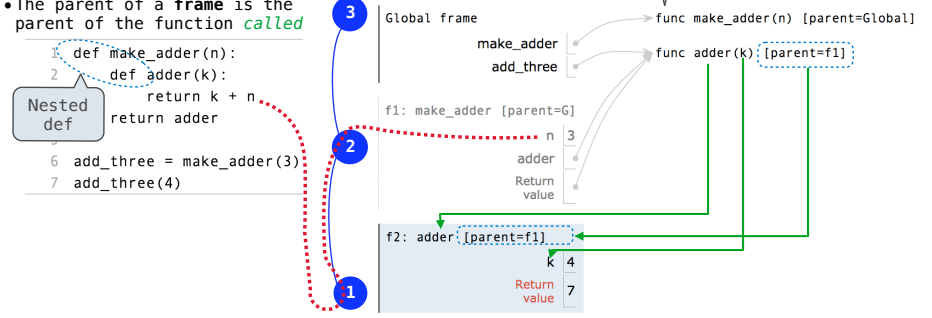
A local def statement

Can refer to names in the enclosing function

- When a function is defined:
- Create a **function value**: func <name>(<formal parameters>)
  - Its parent is the current frame.
- ```
f1: make_adder      func adder(k) [parent=f1]
```
- Bind <name> to the **function value** in the current frame (which is the first frame of the current environment).
- When a function is called:
- Add a **local frame**, titled with the <name> of the function being called.
  - Copy the parent of the function to the **local frame**: [parent=<label>]
  - Bind the <formal parameters> to the arguments in the **local frame**.
  - Execute the body of the function in the environment that starts with the **local frame**.

- Every user-defined function has a **parent frame** (often global)
- The parent of a function is the frame in which it was **defined**
- Every local frame has a **parent frame** (often global)
- The parent of a frame is the parent of the function **called**

A function's signature has all the information to create a local frame



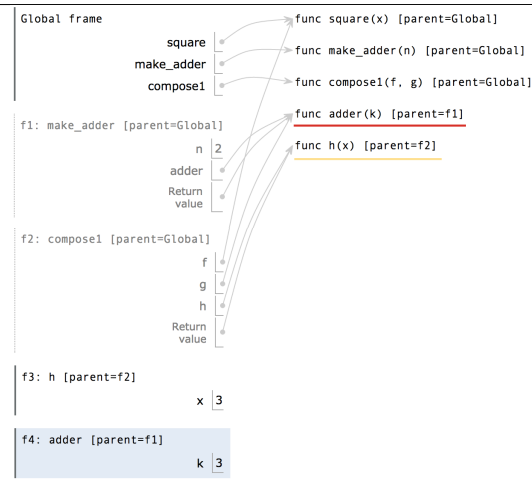
```
1 def make_adder(n):
2   def adder(k):
3     return k + n
4   return adder
5
6 add_three = make_adder(3)
7 add_three(4)
```

Nested def

```
>>> min(2, 1, 4, 3) >>> 2 + 3
1                    5
>>> max(2, 1, 4, 3) >>> 2 * 3
4                    6
>>> abs(-2)         >>> 2 ** 3
2                    8
>>> pow(2, 3)       >>> 5 / 3
8                    1.6666666666666667
>>> len('word')     >>> 5 // 3
4                    1
>>> round(1.75)    >>> 5 % 3
2                    2
>>> print(1, 2)    >>> print(print(1))
1 2                  1
None
```

```
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)
```

Return value of make\_adder is an argument to compose1



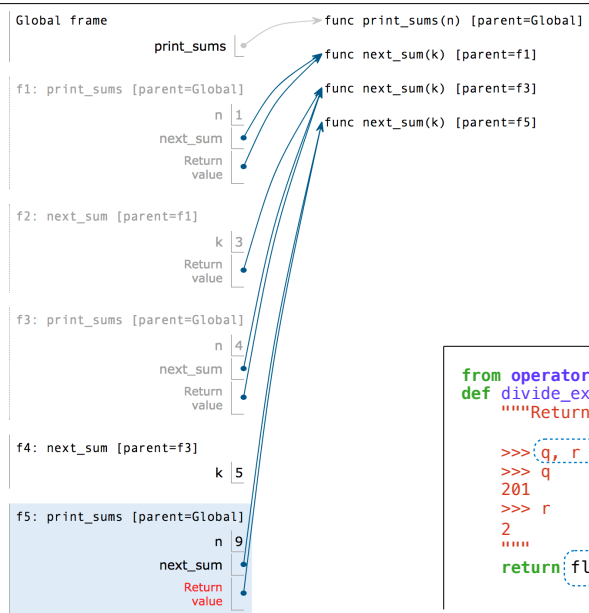
```
def search(f):
    """Return the smallest non-negative integer x for which f(x) is a true value.
    """
    x = 0
    while True:
        if f(x):
            return x
        x += 1

def is_three(x):
    """Return whether x is three.
    """
    >>> search(is_three)
    3
    """
    return x == 3

def inverse(f):
    """Return a function g(y) that returns x such that f(x) == y.
    """
    >>> sqrt = inverse(lambda x: x * x)
    >>> sqrt(16)
    4
    """
    return lambda y: search(lambda x: f(x)==y)
```

```
1 def print_sums(n):
2   print(n)
3   def next_sum(k):
4     return print_sums(n+k)
5   return next_sum
6
7 print_sums(1)(3)(5)
```

Printed output:  
1  
4  
9



```
from operator import add, mul

def curry2(f):
    """Curry a two-argument function.
    """
    >>> m = curry2(add)
    >>> add_three = m(3)
    >>> add_three(4)
    7
    >>> m(2)(1)
    3
    """
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g
```

```
from operator import floordiv, mod
def divide_exact(n, d):
    """Return the quotient and remainder of dividing N by D.
    """
    >>> q, r = divide_exact(2012, 10)
    >>> q
    201
    >>> r
    2
    """
    return floordiv(n, d), mod(n, d)
```

Multiple assignment to two names

Two return values, separated by commas