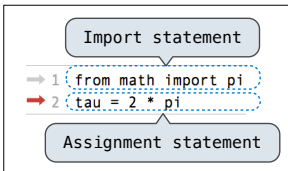
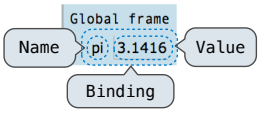




Code (left):

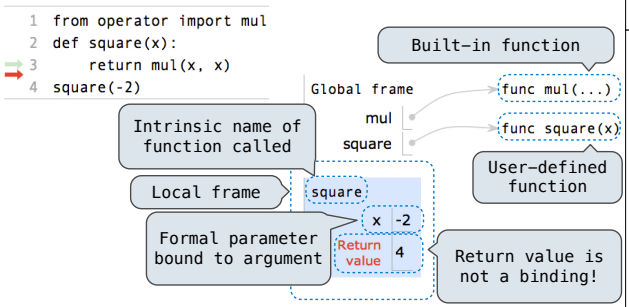
Statements and expressions

Red arrow points to next line.
Gray arrow points to the line just executed

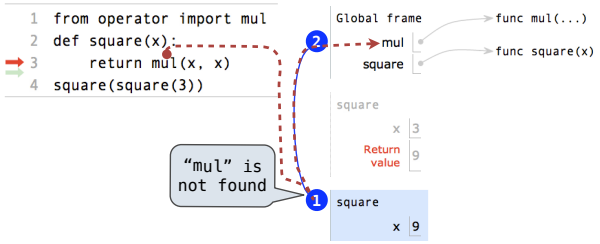
Frames (right):

A name is bound to a value

In a frame, there is at most one binding per name



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



Evaluation rule for call expressions:

1. Evaluate the operator and operand subexpressions.
2. 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:

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

Execution rule for def statements:

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

Execution rule for assignment statements:

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

Execution rule for conditional statements:

Each clause is considered in order.
1. Evaluate the header's expression.
2. If it is a true value, execute the suite, then skip the remaining clauses in the statement.

Evaluation rule for or expressions:

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

Evaluation rule for and expressions:

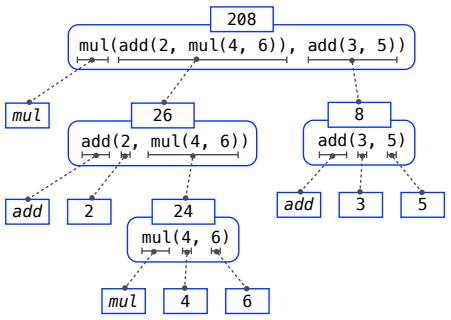
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

Evaluation rule for not expressions:

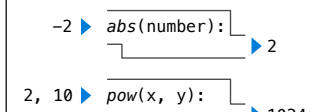
1. Evaluate <exp>; The value is True if the result is a false value, and False otherwise.

Execution rule for while statements:

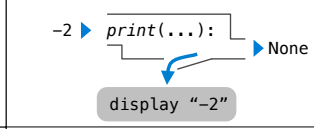
1. Evaluate the header's expression.
2. If it is a true value, execute the (*whole*) suite, then return to step 1.

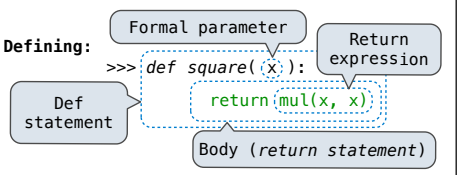


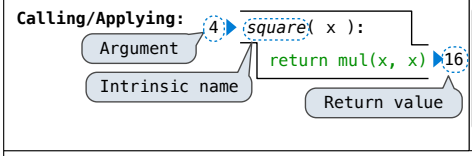
Pure Functions

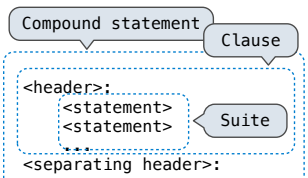


Non-Pure Functions



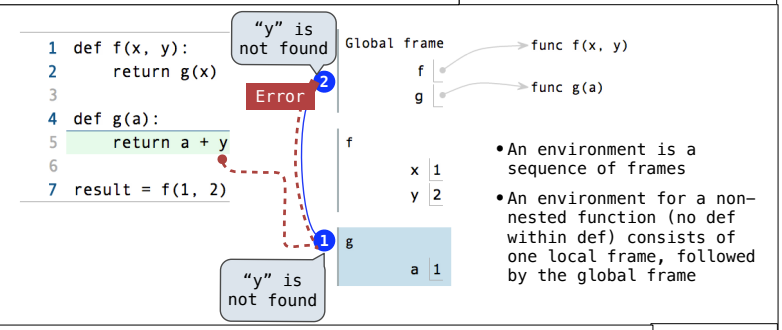




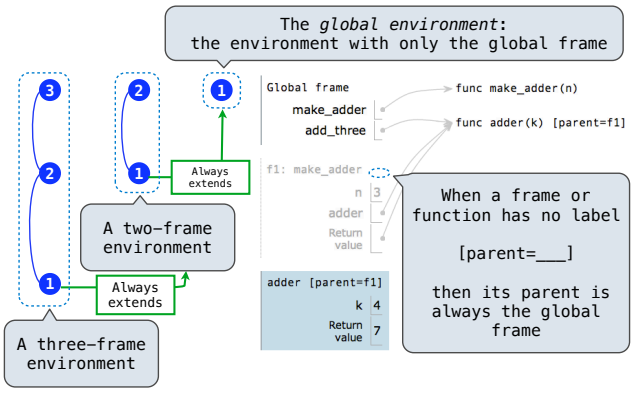


1 statement,
3 clauses,
3 headers,
3 suites,
2 boolean contexts

```
def abs_value(x):
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```



- 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



A frame *extends* the environment that begins with its parent

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

```
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
```

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

$0 + 1^3 + 2^3 + 3^3 + 4^3 + 5^5$

The function bound to term gets called here

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

A function with formal parameters x and y and body "return x * y"

Must be a single expression

```
@trace1
def triple(x):
    return 3 * x
```

is identical to

```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

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

VS

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

- Both create a function with the same arguments & behavior
- Both of those functions are associated with the environment in which they are 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

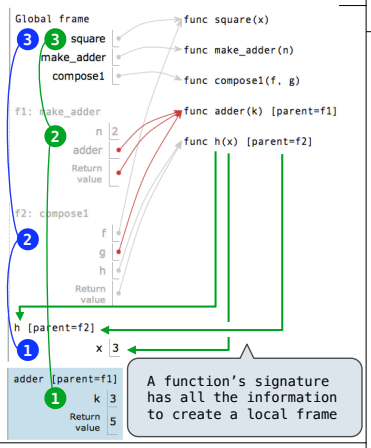
The name add_three is bound to a function

A local def statement

Can refer to names in the enclosing function

```
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)
```

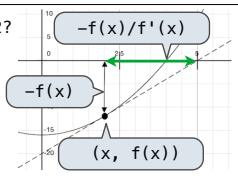
- Every user-defined function has a parent frame
- The parent of a function is the frame in which it was defined
- Every local frame has a parent frame
- The parent of a frame is the parent of the function called



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

How to find the square root of 2?

```
>>> f = lambda x: x**x - 2
>>> find_zero(f, 1)
1.4142135623730951
```

Begin with a function f and an initial guess x

1. Compute the value of f at the guess: f(x)
2. Compute the derivative of f at the guess: f'(x)
3. Update guess to be: $x - \frac{f(x)}{f'(x)}$

```
def iter_improve(update, done, guess=1, max_updates=1000):
    """Iteratively improve guess with update until done returns a true value.

    >>> iter_improve(golden_update, golden_test)
    1.618033988749895
    """
    k = 0
    while not done(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess

def newton_update(f):
    """Return an update function for f using Newton's method."""
    def update(x):
        return x - f(x) / approx_derivative(f, x)
    return update

def approx_derivative(f, x, delta=1e-5):
    """Return an approximation to the derivative of f at x."""
    df = f(x + delta) - f(x)
    return df/delta

def find_root(f, guess=1):
    """Return a guess of a zero of the function f, near guess.

    >>> from math import sin
    >>> find_root(lambda y: sin(y), 3)
    3.141592653589793
    """
    return iter_improve(newton_update(f), lambda x: f(x) == 0, guess)
```

- Compound objects combine objects together
- An *abstract data type* lets us manipulate compound objects as units
- Programs that use data isolate two aspects of programming:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- Data abstraction: A methodology by which functions enforce an abstraction barrier between ***representation*** and ***use***

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

def sum_squares(x, y):
    return square(x)+square(y)
```

What does sum_squares need to know about square?

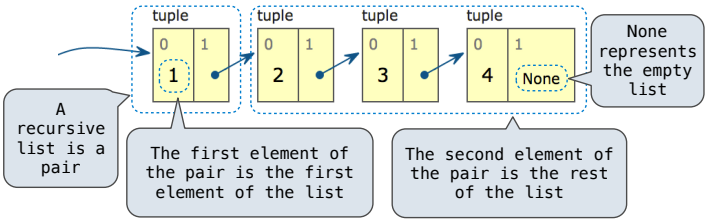
- Square takes one argument. **Yes**
- Square has the intrinsic name square. **No**
- Square computes the square of a number. **Yes**
- Square computes the square by calling mul. **No**

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y), denom(x) * denom(y))

def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)
def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Constructor

Selectors




```
def rational(n, d):
    """Construct a rational number x that represents n/d."""
    return (n, d)

from operator import getitem
def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number x."""
    return getitem(x, 1)
```

```
empty_rlist = None
def rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

If a recursive list s is constructed from a first element f and a recursive list r, then

- first(s) returns f, and
- rest(s) returns r, which is a recursive list.

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

This function represents a pair

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length

def getitem_rlist(s, i):
    """Return the element at index i of rlist s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

```
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

Multiple return values, separated by commas