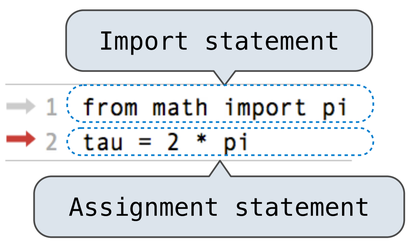
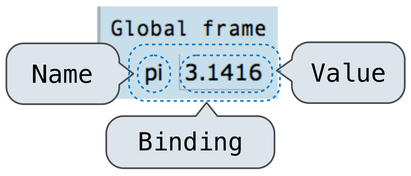




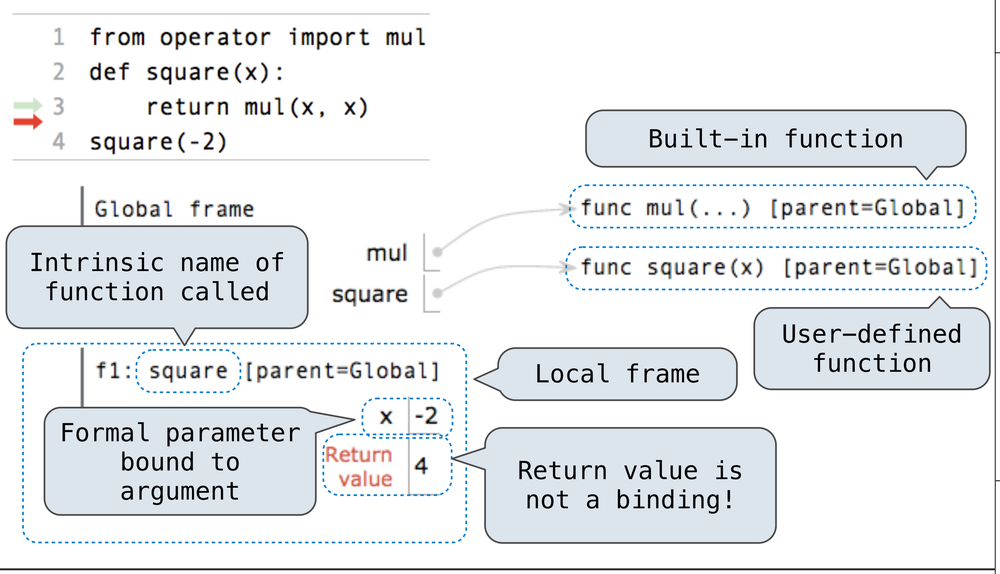
Code (left):

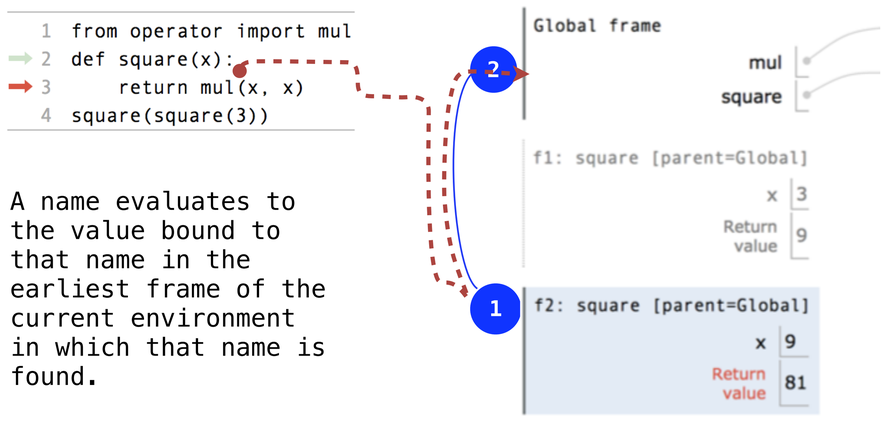
Statements and expressions
Red arrow points to next line.
Gray arrow points to the line just executed

Frames (right):

A name is bound to a value

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





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:

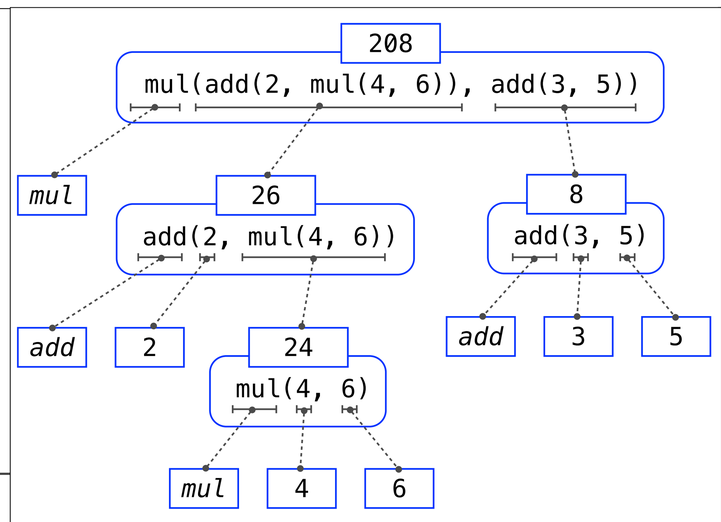
1. Evaluate the subexpression <left>.
2. If the result is a false value v, then the expression evaluates to v.
3. Otherwise, the expression evaluates to the value of the subexpression <right>.

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.



Dictionary Methods

```
>>> food = {"ham":10, "cheese":12}
>>> food["cheese"]
12
>>> "peanuts" in food
False
>>> food["peanuts"] = 7 # adds key-value pair to food dict
>>> "peanuts" in food
True
>>> food["ham"] = food["ham"] + 1
>>> food["ham"]
11
>>> [(key, food[key]) for key in food]
[('ham', 11), ('cheese', 12), ('peanuts', 7)]
```

List comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

A combined expression that evaluates to a list using this evaluation procedure:
1. Add a new frame with the current frame as its parent
2. Create an empty *result list* that is the value of the expression
3. For each element in the iterable value of <iter exp>:
   A. Bind <name> to that element in the new frame from step 1
   B. If <filter exp> evaluates to a true value, then add the value of <map exp> to the result list

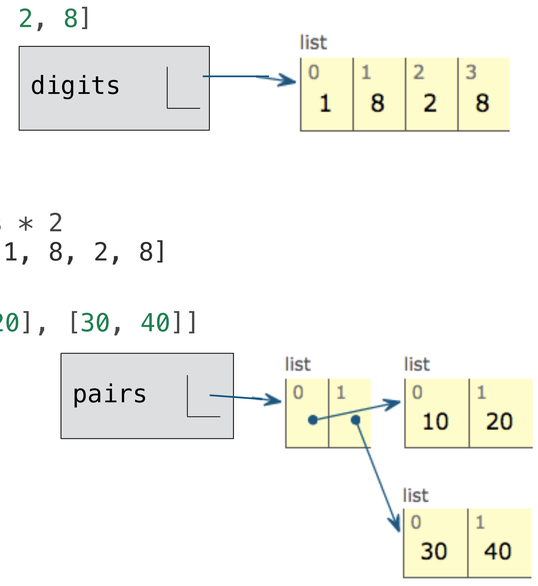
List Methods

```
>>> lst = [8, 61]
>>> lst.append(10)
>>> lst
[8, 61, 10]
>>> lst.extend([2, 3])
>>> lst
[8, 61, 10, 2, 3]
>>> lst.insert(0, 88)
>>> lst
[88, 8, 61, 10, 2, 3]
>>> lst[1:3]
[8, 61]
>>> lst.pop(0)
88
>>> lst
[8, 61, 10, 2, 3]
>>> lst.remove(61)
>>> lst
[8, 10, 2, 3]
>>> lst.pop()
3
>>> lst
[8, 10, 2]
```

List Environment Diagram

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
>>> digits[1:]
[8, 2, 8]
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]

>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```



Lists "Aggregate" Methods

```
>>> lst = [-2, 4, 6]
>>> len(lst)
3
>>> sum(lst)
8
>>> min(lst)
-2
>>> max(lst, key=lambda x: -x)
-2
>>> lst = [(1, 9), (2, 5), (3, 4)]
>>> max(lst, key=lambda y: y[0] * y[1])
(3, 4)
```

Executing a for statement:

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header <expression>, which must yield an iterable value (a list, tuple, iterator, etc.)
2. For each element in that sequence, in order:
   A. Bind <name> to that element in the current frame
   B. Execute the <suite>

... −3, −2, −1, 0, 1, 2, 3, 4, ...

range(−2, 2)

Length: ending value − starting value

Element selection: starting value + index

```
>>> list(range(-2, 2))
[-2, -1, 0, 1]

>>> list(range(4))
[0, 1, 2, 3]
```

List constructor

Range with a 0 starting value

Miscellaneous Operations

```
>>> 5 // 3
1
>>> 5 % 3
2
>>> 2 * 3
6
>>> 2 + 3
5
>>> 6 / 3
2.0
>>> min(2, 1, 4, 3)
1
>>> max(2, 1, 4, 3)
4
>>> abs(-2)
2
>>> pow(2, 3)
8
>>> len('word')
4
>>> print(1, 2)
1 2
```

Functional List Operations

map(func, iterable)
Returns an iterator that applies ***func*** to every item of ***iterable***.

filter(func, iterable)
Returns an iterator from elements of ***iterable*** for which ***func*** returns **True**.

reduce(func, iterable[, initial])
Apply ***func*** of two arguments cumulatively to the items of ***iterable***, from left to right, so as to reduce the iterable to a single value.

If the optional ***initial*** value is present, it is placed before the items of the iterable in the calculation.

```
>>> nums = [1, 2, 3]
>>> list(map(lambda x: x ** 2, nums))
[1, 4, 9]
>>> list(filter(lambda x: x % 2 == 0, nums))
[2]
>>> from functools import reduce
>>> add = lambda x, y: x + y
>>> reduce(add, nums) # 1 + 2 + 3 = 6
6
>>> reduce(add, nums, 10) # 10 + 1 + 2 + 3 = 16
16
```

```
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

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

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.

A function that returns a function

```
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
```

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:

1. Create a **function value**: func <*name*>(<*formal parameters*>)
2. Its parent is the current frame.

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

3. Bind <*name*> to the **function value** in the current frame (which is the first frame of the current environment).

When a function is called:

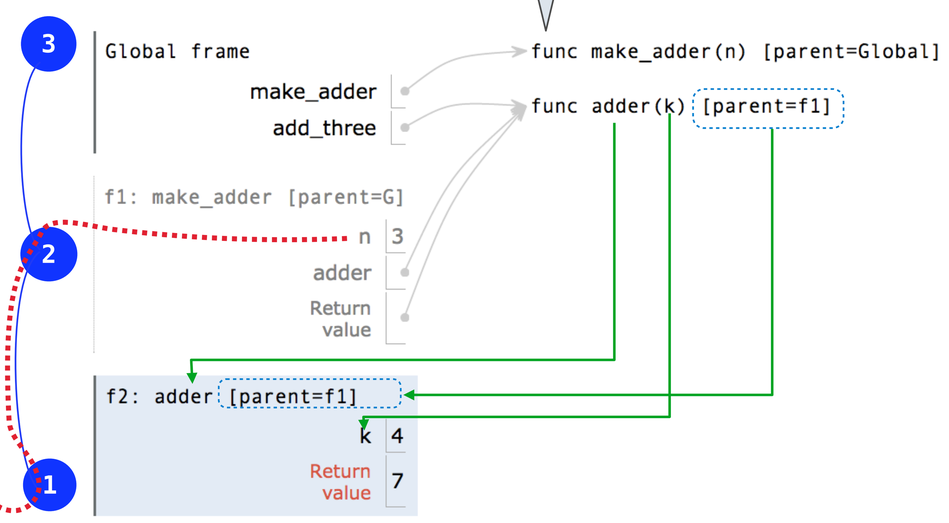
1. Add a **local frame**, titled with the <*name*> of the function being called.
2. Copy the parent of the function to the **local frame**: [parent=<*label*>]
3. Bind the <*formal parameters*> to the arguments in the **local frame**.
4. Execute the body of the function in the environment that starts 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

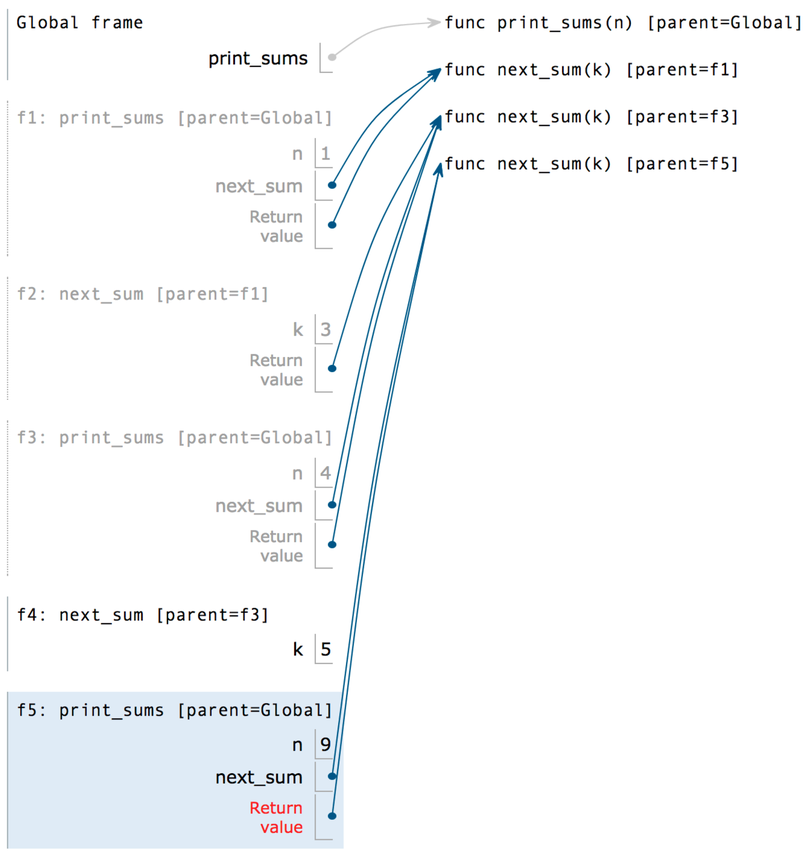
```
1 def make_adder(n):
2     def adder(k):
          return k + n
      return adder
6 add_three = make_adder(3)
7 add_three(4)
```

Nested def




```
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



Python object system:

Idea: All bank accounts have a **balance** and an account **holder**; the **Account** class should add those attributes to each of its instances

A new instance is created by calling a class

```
>>> a = Account('Jim')
>>> a.holder
'Jim'
>>> a.balance
0
```

An account instance

balance: 0
holder: 'Jim'

When a class is called:

1. A new instance of that class is created:
2. The __init__ method of the class is called with the new object as its first argument (named self), along with any additional arguments provided in the call expression.

__init__ is called a constructor

self should always be bound to an instance of the Account class or a subclass of Account

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Falsy values: **0, 0.0, False, None, empty data structure (e.g. string/list/dict/tuple)**
Anything else is truthy.

```
>>> if 0:
...     print('*')
>>> if 1:
...     print('*')
*
>>> if abs:
...     print('*')
*
>>> if 1 and 0:
...     print('*')
>>> if 1 or 0:
...     print('*')
*
>>> if 1 or 1/0:
...     print('*')
*
```

Function call: all arguments within parentheses

```
>>> type(Account.deposit)
<class 'function'>
>>> type(a.deposit)
<class 'method'>
```

```
>>> Account.deposit(a, 5)
10
>>> a.deposit(2)
12
```

Method invocation: One object before the dot and other arguments within parentheses

Dot expression

Call expression

```
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