61A Lecture 3

Wednesday, August 31

Lightning Review: Expressions

Primitive expressions:

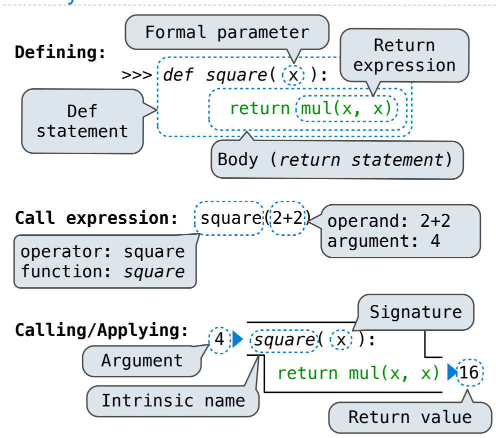
2	add	'hello'
Number	Name	String

Call expressions:

add	(	2	,	3	)
Operator		*Operand 0*		*Operand 1*	

One big nested call expression: `mul(add(2, mul(4, 6)), add(3, 5))`

Life Cycle of a User-Defined Function



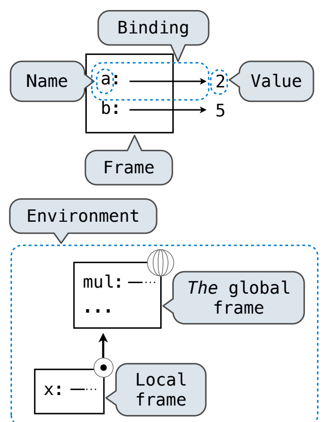
What happens?

Function created
Body stored
Name bound

Op's evaluated
Function called

New frame!
Params bound
Body evaluated

Cast of Characters: Environment Diagrams



Frames:

A name is bound to a value

A frame is a rectangle that contains bindings

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

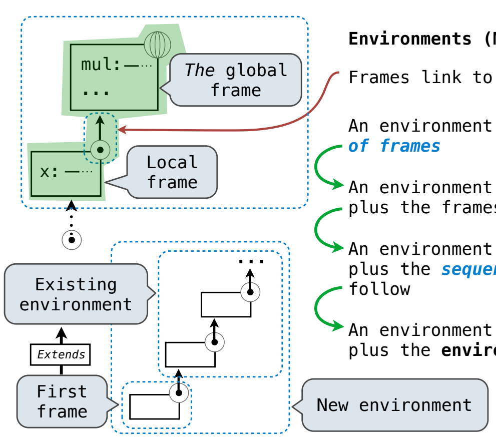
Environments:

An environment is a ***sequence of frames***

So far, environments only have at most two frames

(Friday: longer sequences)

An Environment is a Sequence of Frames



Environments (Memory):

Frames link to each other

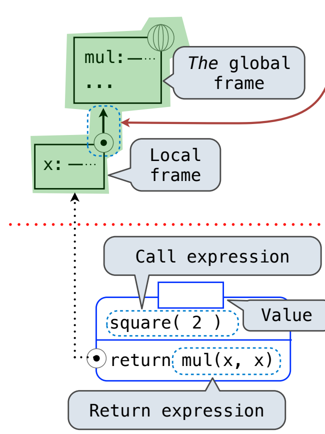
An environment is a ***sequence of frames***

An environment is a first frame, plus the frames that follow

An environment is a first frame, plus the ***sequence of frames*** that follow

An environment is a first frame, plus the **environment** that follows

An Expression is Evaluated in an Environment



Environments (Memory):

Frames link to each other

An environment is a ***sequence*** of frames

An environment is a first frame, plus the **environment** that follows

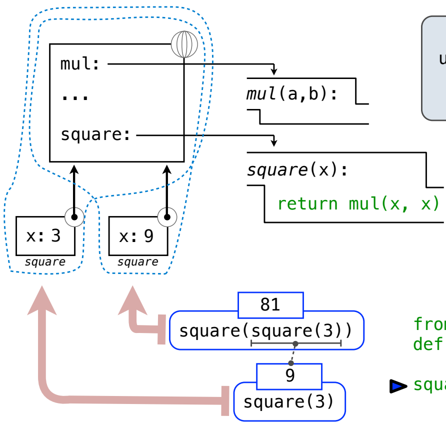
Expressions (Program):

Expressions are Python code

Not part of an environment

They are evaluated in an environment to yield a value

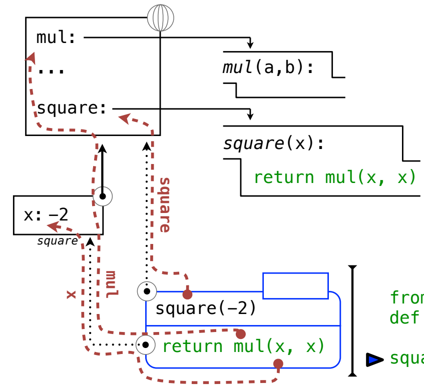
Multiple Environments in One Diagram!



Every call to a user-defined function creates a new local frame

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


Names Have No Meaning Without Environments




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

A name evaluates to the **value bound** to that name

...in the **earliest** frame of the **current environment**

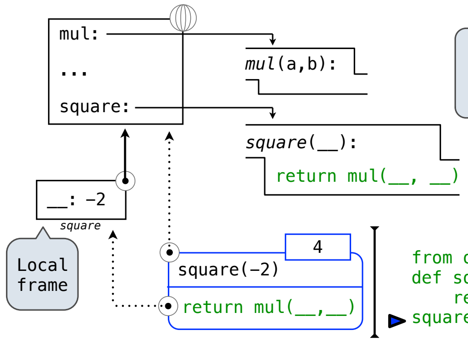
...in which that **name is found**

Formal Parameters

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

vs

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



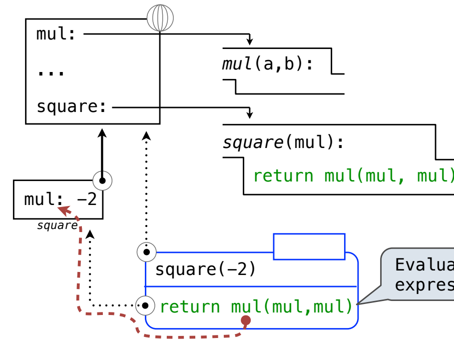
Formal parameter names stay *local* to their frame

```
from operator import mul
def square(__):
    return mul(__, __)
square(-2)
```


Shadowing Names


```
def square(mul):
    return mul(mul, mul)
square(-2)
```

If we use this formal parameter



Evaluating this call expression will fail

Python Feature Demonstration

<Demo>

Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

Statements

</Demo>

Statements

A statement is executed by the interpret to perform an action

Compound statements:

Statement

Clause

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

Suite

The first header determines a statement's type

The header of a clause "controls" the suite that follows

def statements are compound statements

Compound Statements

Compound statements:

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

Suite

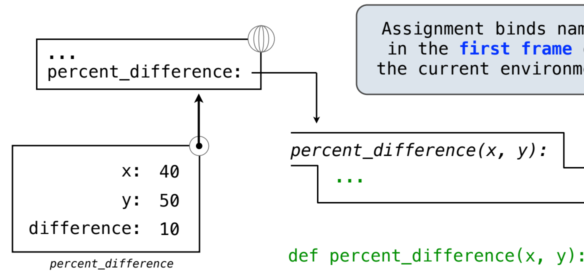
A suite is a sequence of statements

To "execute" a suite means to execute its sequence of statements, in order

Execution Rule for a sequence of statements:

- Execute the first
- Unless directed otherwise, execute the rest

Local Assignment



Assignment binds names in the **first frame** of the current environment

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x
percent_difference(40, 50)
```


Conditional Statements

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

1 statement, 3 clauses, 3 headers, 3 suites

Execution rule for conditional statements:

Each clause is considered in order.

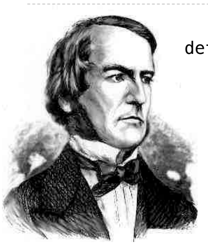
1. Evaluate the header's expression.
2. If it is a true value, execute the suite & skip the rest.

Boolean Contexts

George Boole

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```


Boolean Contexts



George Boole

```
def absolute_value(x):
    """Return the absolute value of x."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x
```

Two boolean contexts

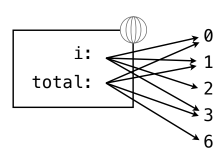
False values in Python: False, 0, '', None *(more to come)*

True values in Python: Anything else (True)

Read Section 1.5.4!

Iteration

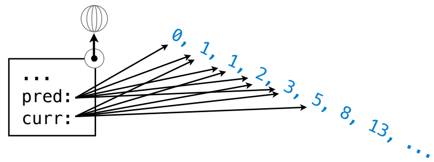
```
i, total = 0, 0
while i < 3:
    i = i + 1
    total = total + i
```



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.

The Fibonacci Sequence




```python
def fib(n):
    """Compute the nth Fibonacci number, for n >= 2."""
    pred, curr = 0, 1   # First two Fibonacci numbers
    k = 2               # Tracks which Fib number is curr
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```


Project 1: Pig

(Demo)