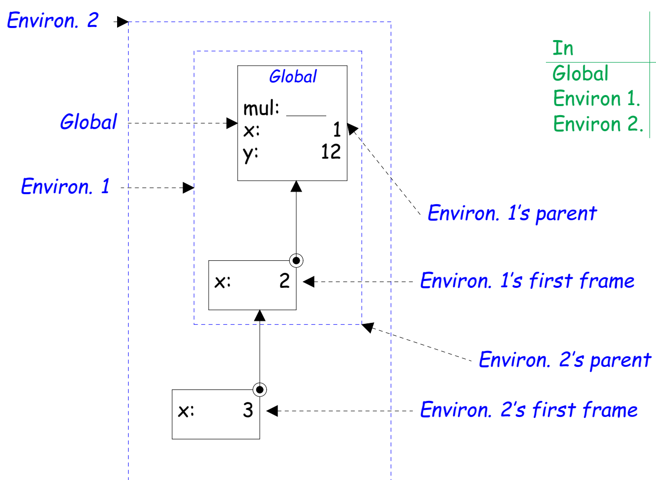
Lecture #3: Recap of Function Evaluation; Control

Summary: Environments

- *Environments* map names to values.
- They consist of chains of *environment frames*.
- An environment is either a *global frame* or a first (local) frame chained to a *parent environment* (which is itself either a global frame or . . .).
- We say that a name is *bound to* a value in a frame.
- The *value (or meaning) of a name* in an environment is the value it is bound to in the first frame, if there is one, . . .
- . . . or if not, the meaning of the name in the parent environment (recursively).

A Sample Environment Chain

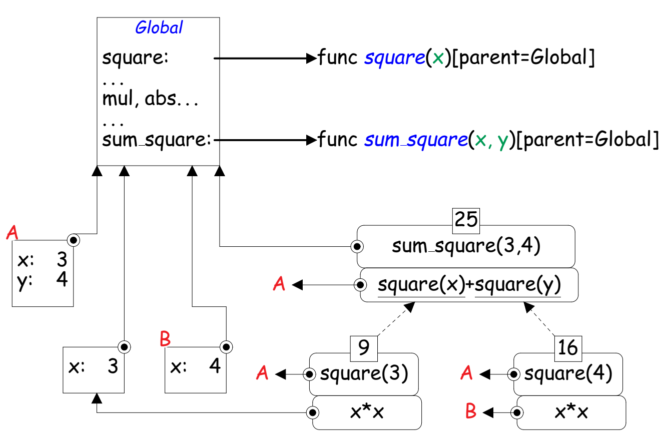


	Value of	
In	x	y
Global	1	12
Environ 1.	2	12
Environ 2.	3	12

Environments: Binding and Evaluation

- Every expression and statement is evaluated (executed) in an environment, which determines the meaning of its names.
- Expressions and subexpressions (pieces of an expression) are evaluated in the same environment as the statement or expression containing them.
- *Assigning* to a variable binds a value to it in (for now) the first frame of the environment in which the assignment is executed.
- *Def statements* bind a name to a function value in the first frame of the environment in which the def statement is executed.
- *Calling* a user-defined function creates a new local environment frame that binds the function's *formal parameters* to the operand values (*actual parameters*) in the call.
- This new local frame is attached to an existing (parent) frame that is taken from the function value that is called, forming a new local environment in which the function's body is evaluated.
- So far, the only parent frames we've seen have been global frames, but we'll see that it can get more complicated.

Example: Evaluation of a Call: sum_square(3,4)



What Does This Do (And Why)?

```
def id(x):
    return x
print(id(id)(id(13)))
```

Execute this

Answer

```
def id(x):
    return x
print(id(id)(id(13)))
```

- We'll denote the user-defined function value created by def id():... by the shorthand $\boxed{\text{id}}$.
- Evaluation proceeds like this:

  id(id)(id(13))

  $\Longrightarrow$ $\boxed{\text{id}}$ (id)($\boxed{\text{id}}$ (id))($\boxed{\text{id}}$ (id)($\boxed{13}$))

  $\Longrightarrow$ $\boxed{\text{id}}$ ($\boxed{13}$)

  (because $\boxed{\text{id}}$ returns its argument).

  $\Longrightarrow$ $\boxed{13}$

  (again because $\boxed{\text{id}}$ returns its argument).

- *Important:* There is nothing new on this slide! Everything follows from what you've seen so far.

Nested Functions

- In lecture #2, I had this example:

```
def incr(n):
    def f(x):
        return n + x
    return f

incr(5)(6)
```

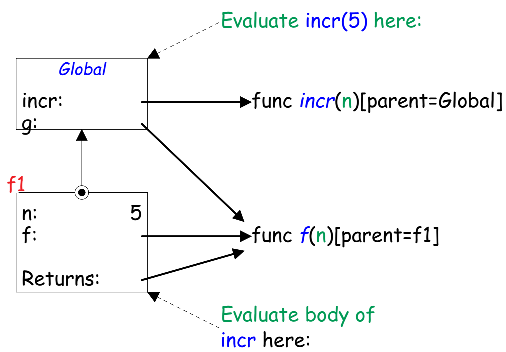
- We evaluated the argument to print by substitution:

```
incr(5)  ===>  | def f(x):  return 5 + x |  ===> λ x: 5 + x
               | return f                 |

incr(5)(6) ===> (λ x: 5 + x)(6) ===> 5 + 6 ===> 11
```

- So how does this work with environments?

Environments for incr (I)

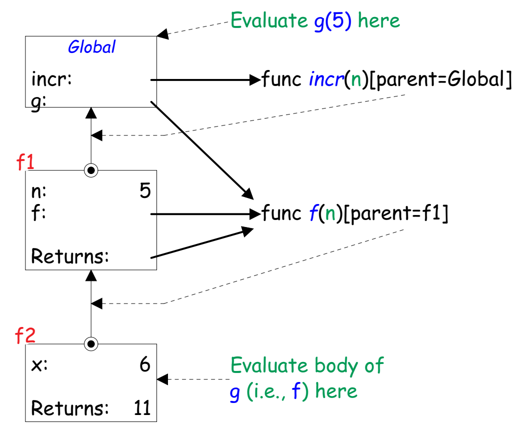
```
def incr(n):
    def f(x):
        return n + x
    return f

# Break incr(5)(6)
# into two steps:
g = incr(5)
print(g(6))
```



- The parent points of incr is Global because the defintion of incr was evaluated in the global environment.
- The parent pointer for the value of g (returned by incr(5)) is f1, not Global, because the definition of f was evaluated in f1.

Environments for incr (II)

```
def incr(n):
    def f(x):
        return n + x
    return f

g = incr(5)
print(g(6))
```



- f2 gets its parent pointer from g's value, since it is the local frame for evaluating a call to g. (Same rule for f1.)

Recap

- Every expression or statement is evaluated in an environment—a sequence of frames.
- Every frame (except the global frame) is linked to a parent frame.
- Every function *value* is linked to the environment in which its **def** is evaluated.
- Every function *call* creates a new local frame that is linked to the same frame as the function value being called.
- The total effect is the same as for the substitution model, but we can also handle changes in the values of variables.
- Looking ahead, there are still two constructs—**global** and **nonlocal**—that will require additions.
- But what we have here basically covers how names work in most of Python.

Control

- The expressions we've seen evaluate all of their operands in the order written.
- While there are very clever ways to do everything with just this [challenge!], it's generally clearer to introduce constructs that *control* the order in which their components execute.
- A *control expression* evaluates some or all of its operands in an order depending on the kind of expression, and typically on the values of those operands.
- A *statement* is a construct that produces no value, but is used solely for its side effects.
- A *control statement* is a statement that, like a control expression, evaluates some or all of its operands, etc.
- We typically speak of statements being *executed* rather than evaluated, but the two concepts are essentially the same, apart from the question of a value.

Conditional Expressions (I)

- The most common kind of control is *conditional evalutation (execution)*.
- In Python, to evaluate

  TruePart `if` *Condition* `else` *FalsePart*

  - First evaluate *Condition*.
  - If the result is a "*true value*," evaluate *TruePart*; its value is then the value of the whole expression.
  - Otherwise, evaluate *FalsePart*; its value is then the value of the whole expression.
- Example:

If x is 2:	If x is 0:
1 / x if x != 0 else 1	1 / x if x != 0 else 1
1 / x if $\boxed{2}$!= 0 else 1	1 / x if 0 != 0 else 1
$\Longrightarrow$ 1 / x if $\boxed{\text{True}}$ else 1	$\Longrightarrow$ 1 / x if $\boxed{\text{False}}$ else 1
$\Longrightarrow$ 1 / x	$\Longrightarrow$ 1
$\Longrightarrow$ $\boxed{1}$ / $\boxed{2}$	$\Longrightarrow$ $\boxed{1}$
$\Longrightarrow$ $\boxed{0.5}$	

"True Values"

- Conditions in conditional constructs can have any value, not just True or False.
- For convenience, Python treats a number of values as indicating "false":
  - False
  - None
  - 0
  - Empty strings, sets, lists, tuples, and dictionaries.
- All else is a "true value" by default.
- For example: 13 if 0 else 5 and 13 if [] else 5 both evaluate to 5.

Conditional Expressions (II)

- To evaluate

  Left `and` *Right*

  - Evaluate *Left*.
  - If it is a false value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of *Right*.
- This is an example of something called "*short-circuit evaluation*."
- For example,

  5 and "Hello" $\Longrightarrow$ $\boxed{\text{"Hello"}}$.

  [] and 1 / 0 $\Longrightarrow$ $\boxed{\text{[]}}$. (1/0 is not evaluated.)

Conditional Expressions (III)

- To evaluate

  Left `or` *Right*

  - Evaluate *Left*.
  - If it is a true value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of *Right*.
- Another example of "*short-circuit evaluation*."
- For example,

  5 or "Hello" $\Longrightarrow$ $\boxed{5}$.

  [] or "Hello" $\Longrightarrow$ $\boxed{\text{"Hello"}}$.

  [] or 1 / 0 $\Longrightarrow$ $\boxed{?}$.

Conditional Statement

- Finally, this all comes in statement form:

```
if Condition1:
    Statements1        # Indented blocks are called suites
    ...                # They group statements
elif Condition2:
    Statements2
    ...
...
else:
    Statementsn
    ...
```

- Execute (only) *Statements1* if *Condition1* evaluates to a true value.
- Otherwise execute *Statements2* if *Condition2* evaluates to a true value (optional part).
- ...
- Otherwise execute *Statementsn* (optional part).

Example

```
def signum(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1
```

```
# Alternative Definition
def signum(x):
    return 1 if x > 0 else 0 if x == 0 else -1
```

Indefinite Repetition

- With conditionals and function calls, we can conduct computations of any length.
- For example, to sum the squares of all numbers from 1 to N (a parameter):

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
```

- This will repeatedly call `sum_squares` with decreasing values (down to 1), adding in squares: Execute here

```
sum_squares(3) => 3**2 + sum_squares(2)
               => 3**2 + 2**2 + sum_squares(1)
               => 3**2 + 2**2 + 1**2 + sum_squares(0)
               => 3**2 + 2**2 + 1**2 + 0 => 14
```


Explicit Repetition

- But in the Python, C, Java, and Fortran communities, it is more usual to be explicit about the repetition.
- The simplest form is **while**:

```
while Condition:
    Statements
```

means "If condition evaluates to a true value, execute statements and repeat the entire process. Otherwise, do nothing."

- The effect is (nearly) identical to

```
def loop():
    if Condition:
        Statements
        loop()

loop()   # Start things off
```

- ...except that (for most Python implementations) the latter eventually runs out of memory; and we'll have to do something about assignments to variables (more on that later).

Sum_squares Iteratively?

- Our original sum_squares was

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
```

- How do we do the same thing with a **while** loop?

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
```


Sum_squares Iteratively (II)

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    k = 1
    while k <= N:
        result += k**2
        k += 1
    return result
```

Execute this

Another Way

- Alternatively, I can make this a little shorter by adding the other way:

```
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    while N >= 1:
        result += N**2     # Or result = result + N**2
        N -= 1             # Or N = N-1
    return result
```

Execute here