Announcements

- Labs 1 and 2 due Tuesday (at 11:59PM).
- Homework 1 due Thursday.
- Orientations starting: lab orientations are Mondays, discussion orientations Wednesdays. These are recorded.
- Lab party on Monday, homework party on Tuesday. See Piazza @151.
- Conceptual office hours starting this week. See Piazza @174.
- Ask questions on the Piazza thread for today's lecture (@155).

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).
- 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.
Creating the Sample Environment Chain

Executing the following code will result in the environment on the left when execution reaches the comment.

```python
x = 1
y = 12
def g1(x):
def g2(x):
    # Stop here
    print(x)
g2(x + 1)
g1(2)
```

The call to print is executed in this environment. Continuing from the comment, the program would print 3.

Execute in Python tutor

---

Environments: Binding and Evaluation

- **Assigning** to a variable binds a value to it (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.
- This new function value contains a link to this same environment.
- **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.

---

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

```python
def square(x):
    return x*x
def sum_square(x, y):
    return square(x)+square(y)
z = sum_square(3, 4)
```

What Does This Do (And Why)?

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

Execute this
Answer

```python
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 `id`.
- Evaluation proceeds like this:
  ```python
  id(id)(id(13))
  ⇒ [id] ( [id] ( [id] (13) ))
  ⇒ [id] ( [id] (13) )
  (because first `id` call returns its argument).
  ⇒ [id] (13)
  (because inner `id` call returns its argument).
  ⇒ 13
  (because call to returned `id` value 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:
  ```python
def incr(n):
    def f(x):
        return n + x
    return f
ingr(5)(6)
```
- We evaluated the argument to `print` by substitution:
  ```python
  incr(5) ===> def f(x): return 5 + x
             func f(x): 5 + x
  incr(5)(6) ===> func f(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 definition 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
f1:
Returns:
g = incr(5)
print(g(6))
f2:
```

- `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`.)

See in Python Tutor
Recap

- Every expression or statement is evaluated in an environment—a sequence of frames.
- Every assignment to a variable and every `def` binds (or changes the binding) of its variable or defined name in the first frame of this environment.
- 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.

New Topic: 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 evaluation (or execution).
- In Python, to evaluate
  
  ```python
  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:
  
  ```python
  if x is 2:
    1 / x if x != 0 else 11 / x if 2 != 0 else 1
  =⇒ 1 / x if True else 1
  =⇒ 1 / x
  =⇒ 1/2
  ```
  ```python
  if x is 0:
    1 / x if x != 0 else 11 / x if 0 != 0 else 1
  =⇒ 1 / x if False else 1
  =⇒ 1
  ```

"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:
  
  ```python
  13 if 0 else 5 == 13 if [] else 5 == 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" \(\Rightarrow\) "Hello".
  [] and 1 / 0 \(\Rightarrow\) []. (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" \(\Rightarrow\) 5.
  [] or "Hello" \(\Rightarrow\) "Hello".
  [1, 2] or 1 / 0 \(\Rightarrow\) [1, 2].
  [] or 1 / 0 \(\Rightarrow\) ERROR.

Conditional Statement

• Finally, this all comes in statement form:
  
  if Condition_1:
  
  Statements_1

  elif Condition_2:
  
  Statements_2

  ...

  else:
  
  Statements_n

• Execute (only) Statements_1 if Condition_1 evaluates to a true value.
• Otherwise execute Statements_2 if Condition_2 evaluates to a true value (elifs are optional parts).
• ...
• Otherwise execute Statements_n (else is an optional part).

Examples

# Alternative Definitions

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

def max(x, y):
    if x > y:
        return x
    else:
        return y

def min(x, y):
    if x < y:
        return x
    else:
        return y
Side Trip: Suites and Sequences

• The sequence of indented statements after the colon in
  ```python
  if x >= 0:
    print(x)
  y = x
  ```
  is called a **suite**. In effect it is a single statement formed from two.
• Executing the suite itself means executing each of its statements
  in sequence (unless one of them says otherwise).
• Every statement in the suite has the same indentation, and itends
  at the next statement that is indented to a previous level:
  ```python
  x = 0
  if x > 1:
    print(">1")
  if x < 6:
    print("<6")
  print("x =", x)
  # Prints nothing
  # Prints "x = 0"
  ```
• Every language has some way of grouping statements like this.
• Few do it like Python. (Interesting story behind this.)

Iteration

• Suppose you would like to compute $1^2 + 2^2 + \ldots + 100^2$.
• (Yes, I know there is a formula for this. Humor me.)
• You’d probably prefer not to write
  ```python
  print(1 ** 2 + 2 ** 2 + \ldots + 100 ** 2)
  ```
• Actually, we already know enough to do this:
  ```python
  def add_sq(accum, k, n):
    """Return ACCUM + K ** 2 + (K+1)**2 + \ldots + N**2."""
    if k > n:
      return accum
    else:
      return add_sq(accum + k ** 2, k + 1, n)
  ```
• Go ahead: try it in on a small case in the Python Tutor
• This is an example of a **recursive function**. We’ll come back to such
  functions later in the course.

While Statements

• Usually, though, programmers deal with problems like this summation
  using some kind of **looping construct**, which explicitly executes
  statements repeatedly.
• The **while** statement gives us **indefinite repetition**, meaning repetition
  until some condition is met (or as long as some condition is met).
• For our example, (also see a **small case in the Python Tutor**):
  ```python
  accum = 0
  k = 1
  n = 100
  while k <= n:
    accum = accum + k ** 2
    k += 1
  # Another way to write k = k + 1
  print(accum)
  ```
• Meaning of the while loop:
  A. Test the **loop condition** (here, $k <= n$).
  B. If it’s true, execute the suite that follows (the **loop body**), and
    then repeat from step A.
  C. Otherwise, end the loop (and continue to the print call).

Example: Finding Prime Factors

• A **prime number** is an integer greater than 1 whose only factors are
  1 and the number itself (e.g., 3, 5, 7, 11).
• So how do make this function fulfill its comment?
  ```python
  def is_prime(n):
    """Return True iff N is prime."""
    return n > 1 and smallest_factor(n) == n
  ```
  ```python
  def smallest_factor(n):
    """Returns the smallest value k>1 that evenly divides N.""
    # ???
  ```
  ```python
  def print_factors(n):
    """Print the prime factors of N.""
    # ???
  ```
• Try filling these in. (See Demo and also 03.py).