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

<table>
<thead>
<tr>
<th>In</th>
<th>Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>x: 1</td>
</tr>
<tr>
<td></td>
<td>y: 12</td>
</tr>
<tr>
<td>Environ 1.</td>
<td>x: 2</td>
</tr>
<tr>
<td></td>
<td>y: 12</td>
</tr>
<tr>
<td>Environ 2.</td>
<td>x: 3</td>
</tr>
<tr>
<td></td>
<td>y: 12</td>
</tr>
</tbody>
</table>
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: \( \text{sum\_square}(3,4) \)

- **Global**
  - `square(x)[parent=Global]`
  - `sum\_square(x, y)[parent=Global]`

```
square(x)
```

```
sum\_square(x, y)
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

```
x*x
```

Last modified: Sat Feb 18 22:56:06 2017
def id(x):
    return x
print(id(id)(id(13)))

Execute this
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 \texttt{id}.

• Evaluation proceeds like this:

\[
\begin{align*}
\text{id(id)(id(13))} & \\
\quad & \Rightarrow \text{id( id( id(13))}) \\
\quad & \Rightarrow \text{id( id(13))} \\
\quad & \Rightarrow \text{id}(13) \\
\quad & \Rightarrow 13
\end{align*}
\]

\text{(because first \texttt{id} call returns its argument).}

\text{(because inner \texttt{id} call returns its argument).}

\text{(because call to returned \texttt{id} value returns its argument).}

• \textbf{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
```

`incr(5)(6)`

• *We evaluated the argument to* `print` *by substitution:*

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

===> \lambda x: 5 + x
```

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

• *So how does this work with environments?*
Environments for incr (I)

```python
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)

```python
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 evaluation (execution)*.

- In Python, to evaluate

  \[ \text{TruePart} \text{ if } \text{Condition} \text{ else } \text{FalsePart} \]

  - First evaluate \text{Condition}.
  - If the result is a “*true value,*” evaluate \text{TruePart}; its value is then the value of the whole expression.
  - Otherwise, evaluate \text{FalsePart}; its value is then the value of the whole expression.

- Example:  
  
  If \(x\) is 2:
  
  \[
  \frac{1}{x} \text{ if } x \neq 0 \text{ else } \frac{11}{x} \text{ if } 2 \neq 0 \text{ else } 1
  \]
  
  \[
  \Rightarrow \frac{1}{x} \text{ if } \text{True} \text{ else } 1
  \]
  
  \[
  \Rightarrow \frac{1}{x}
  \]
  
  \[
  \Rightarrow \frac{1}{2}
  \]
  
  \[
  \Rightarrow 0.5
  \]

  If \(x\) is 0:

  \[
  \frac{1}{x} \text{ if } x \neq 0 \text{ else } \frac{11}{x} \text{ if } 0 \neq 0 \text{ else } 1
  \]

  \[
  \Rightarrow \frac{1}{x} \text{ if } \text{False} \text{ else } 1
  \]

  \[
  \Rightarrow 1
  \]
“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: `13 if 0 else 5` and `13 if [] else 5` both evaluate to 5.
Conditional Expressions (II)

- To evaluate \( \text{Left} \) and \( \text{Right} \)
  - Evaluate \( \text{Left} \).
  - If it is a false value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of \( \text{Right} \).

- This is an example of something called "short-circuit evaluation."

- For example,

  \[
  5 \text{ and } "Hello" \implies "Hello". \\
  [] \text{ and } 1 / 0 \implies []. (1/0 \text{ is not evaluated}.)
  \]
Conditional Expressions (III)

• To evaluate $\text{Left or Right}$
  - Evaluate $\text{Left}$.
  - If it is a true value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of $\text{Right}$.

• Another example of “short-circuit evaluation.”

• For example,
  
  \[
  5 \text{ or } "\text{Hello}" \implies 5 .
  \]
  \[
  [] \text{ or } "\text{Hello}" \implies "\text{Hello}" .
  \]
  \[
  [] \text{ or } 1 / 0 \implies ? .
  \]
Conditional Statement

- Finally, this all comes in statement form:

```python
if Condition1:
    Statements1  # Indented blocks are called suites
    ...
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

# Alternative Definition

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