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).

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 function (environment frame).
```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:

  ```
  id(id)(id(13))
  ⇒ id(id)(13)
  ⇒ id(13)
  ⇒ 13
  ```

• 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:

  ```
  incr(5) ===>
  def f(x):
      return 5 + x
  return f
  ===>
  λx: 5 + x
  incr(5)(6) ===> (λx: 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
g = incr(5)
print(g(6))
```

```
Global incr:
  g:
    func incr(
        n
      )
      [parent=Global]

f1
  n: 5
  f:
    Returns:
    func f(
        n
      )
      [parent=f1]

Evaluate body of incr here:
```

• 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
g = incr(5)
print(g(6))
```

```
Global incr:
  g:
    func incr(
        n
      )
      [parent=Global]

f1
  n: 5
  f:
    Returns:
    func f(
        n
      )
      [parent=f1]

f2
  x: 6
  Returns: 11
```

• `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.

---

### 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, 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 control statement is a statement that, like a control expression, evaluates some or all of its operands, etc.

• 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, produces no value, but is used solely for its side effects.

• The two concepts are essentially the same, apart from the question of a value.

---

### Answer

```
from what you've seen so far:

Important: There is nothing new on this slide! Everything follows:

(new call to returned value returns its argument)  P
(because call returns its argument)  P
(new call to returned value returns its argument)  P
(because call returns its argument)  P
(new call to returned value returns its argument)  P
(because call returns its argument)  P
```

• By the shorthand `...` we'll denote the user-defined function value created by `def id()`:

  ```python
  def id()
  ```
Conditional Expressions (I)

The most common kind of control is **conditional evaluation** (evaluation).

In Python, to evaluate

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

- 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:

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

\[ 1 / x \text{ if } 2 \neq 0 \text{ else } 1 \]

\[ \Rightarrow 1 / x \text{ if True else } 1 \]

\[ \Rightarrow 1 / x \]

\[ \Rightarrow \frac{1}{2} \]

If `x` is 0:

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

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

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

\[ \Rightarrow 1 \]

**True Values**

Conditions in conditional constructs can have any value, not just True or False.

- `False`
- `None`
- `0`
- Empty strings, lists, tuples, and dictionaries.

All else is a "true value" by default.

- Empty strings, lists, tuples, and dictionaries.
- `None`
- `True`
- `False`
- `13`
- `1 / 1`
- `0 / 0`
- `[]`
- `if True:
  print("True")
else:
  print("False")`

**Example:**

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

**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,

\[ 5 \text{ and } "Hello" \]

\[ \Rightarrow "Hello" \]

\[ \text{[] and } 1 / 0 \]

\[ \Rightarrow \text{[]} \] (\(1/0\) 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."

- Otherwise the value of the expression is that of `Right`
- If `1 / 0` is a true value, that becomes the value of the whole expression.
- Evaluate `Left`
- Left or Right

To evaluate

**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**

```python
# Alternative Definition

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

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

**Conditional Expressions (I)**

The most common kind of control is conditional evaluation (execute-

\[ \text{if } x \neq 0 \text{ then } 1/0 \text{ else } 1 \]

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

\[ \Rightarrow 1 \]

\[ \Rightarrow \frac{1}{0} \]

\[ \Rightarrow \text{?} \]