Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade

• Homework 1 is due next Wednesday 9/10 at 11:59pm
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade

• Homework 1 is due next Wednesday 9/10 at 11:59pm

• Office hours are a great place to ask questions about lab and homework assignments (demo)
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade

• Homework 1 is due next Wednesday 9/10 at 11:59pm

• Office hours are a great place to ask questions about lab and homework assignments (demo)

• You can switch to sections with open space. http://goo.gl/nWfv7Z
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade

• Homework 1 is due next Wednesday 9/10 at 11:59pm

• Office hours are a great place to ask questions about lab and homework assignments (demo)

• You can switch to sections with open space. http://goo.gl/nWfv7Z

• Michelle Hwang's sections (15, 18) are for students with little prior CS experience
Announcements

• Lab 1 is due Wednesday 9/3 at 11:59pm

• Submitting labs and attending section may help your grade

• Homework 1 is due next Wednesday 9/10 at 11:59pm

• Office hours are a great place to ask questions about lab and homework assignments (demo)

• You can switch to sections with open space. http://goo.gl/nWfv7Z

• Michelle Hwang's sections (15, 18) are for students with little prior CS experience

• Videos are a mix of Fall 2013 and new content
Names, Assignment, and User-Defined Functions
Types of Expressions
Types of Expressions

Primitive expressions:
Types of Expressions

Primitive expressions:

2

Number or Numeral
Types of Expressions

Primitive expressions:

2
(Number or Numeral)

add
(Name)
Types of Expressions

Primitive expressions:

2
Number or Numeral

add
Name

'h hello'
String
Types of Expressions

**Primitive expressions:**

- 2
- `add`
- `'hello'`

  - Number or Numeral
  - Name
  - String

**Call expressions:**
Types of Expressions

Primitive expressions:

- 2
- add
- 'hello'

Call expressions:

- max (2, 3)
Types of Expressions

**Primitive expressions:**

- 2
- `add`
- `'hello'`

**Call expressions:**

- `max ( 2 , 3 )`

Operator:

- Number or Numeral
- Name
- String
Types of Expressions

**Primitive expressions:**

- **Number or Numeral:** 2
- **Name:** 'hello'
- **Operator:** add

**Call expressions:**

- **Operator:** max
- **Operand:** (2, 3)
Types of Expressions

**Primitive expressions:**

- Number or Numeral: 2
- Name: add
- String: 'hello'

**Call expressions:**

```
max
  Operator

2
  Operand

,  

3
  Operand
```

```
max(min(pow(3, 5), -4), min(1, -2))
```
Types of Expressions

**Primitive expressions:**
- Number or Numeral: 2
- Name: add
- String: 'hello'

**Call expressions:**
- Operator: max
- Operand: (2, 3)

An operand can also be a call expression:
- max(min(pow(3, 5), -4), min(1, -2))
Types of Expressions

**Primitive expressions:**

- 2
- `add`
- 'hello'

**Call expressions:**

- `max`
- `( 2 , 3 )`

An operand can also be a call expression:

- `max(min(pow(3, 5), -4), min(1, -2))`
Discussion Question 1
Discussion Question 1

What is the value of the final expression in this sequence?
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min
```
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min

>>> f = max
```

5
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min
>>> f = max
>>> g, h = min, max
```

5
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min

>>> f = max

>>> g, h = min, max

>>> max = g
```

5
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min
>>> f = max
>>> g, h = min, max
>>> max = g
>>> max(f(2, g(h(1, 5), 3)), 4)
```

5
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min

>>> f = max

>>> g, h = min, max

>>> max = g

>>> max(f(2, g(h(1, 5), 3)), 4)

???
```
Discussion Question 1

What is the value of the final expression in this sequence?

```python
>>> f = min
>>> f = max
>>> g, h = min, max
>>> max = g
>>> max(f(2, g(h(1, 5), 3)), 4)
```

???
Environment Diagrams
Environment Diagrams

Environment diagrams visualize the interpreter’s process.
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. from math import pi
2. tau = 2 * pi
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. `from math import pi`
2. `tau = 2 * pi`

Global frame

| pi | 3.1416 |
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. from math import pi
2. \( \tau = 2 \times \pi \)

Global frame

\( \pi \) 3.1416

Code (left): Frames (right):
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

<table>
<thead>
<tr>
<th></th>
<th>Code (left):</th>
<th>Frames (right):</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>from math import pi</td>
<td>Global frame</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>pi</td>
</tr>
<tr>
<td>2</td>
<td>tau = 2 * pi</td>
<td></td>
</tr>
</tbody>
</table>

Statements and expressions
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. `from math import pi`
2. `tau = 2 * pi`

**Code (left):**

Statements and expressions

**Frames (right):**

Global frame

| pi | 3.1416 |
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. from math import pi
2. tau = 2 * pi

Code (left):

Statements and expressions

Frames (right):

Global frame

pi | 3.1416

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. `from math import pi`  
2. `tau = 2 * pi`

**Code (left):**
Statements and expressions

**Frames (right):**

Arrows indicate evaluation order

**Interactive Diagram**
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. Import statement
   ```python
   from math import pi
   ```

2. Assignment statement
   ```python
   tau = 2 * pi
   ```

   **Global frame**
   ```
   pi | 3.1416
   ```

   Code (left):
   Statements and expressions
   Arrows indicate evaluation order

   Interactive Diagram
Environment diagrams visualize the interpreter’s process.

Code (left):
Statements and expressions

Arrows indicate evaluation order

Frames (right):

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Code (left):
Statements and expressions
Arrows indicate evaluation order

Frames (right):
Each name is bound to a value

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

1. `from math import pi`
   - Just executed

2. `tau = 2 * pi`
   - Next to execute

**Code (left):**
Statements and expressions

**Frames (right):**
Each name is bound to a value

Arrows indicate evaluation order
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**
Statements and expressions

**Frames (right):**
Each name is bound to a value

Arrows indicate evaluation order

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**
- Statements and expressions
- Arrows indicate evaluation order

**Frames (right):**
- Each name is bound to a value
- Within a frame, a name cannot be repeated

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**
- Statements and expressions
- Arrows indicate evaluation order

**Frames (right):**
- Each name is bound to a value
- Within a frame, a name cannot be repeated

(Demo)

**Interactive Diagram**
Assignment Statements

Interactive Diagram
Assignment Statements

1  a = 1
2  b = 2
3  b, a = a + b, b
### Assignment Statements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a = 1</td>
</tr>
<tr>
<td>2</td>
<td>b = 2</td>
</tr>
<tr>
<td>3</td>
<td>b, a = a + b, b</td>
</tr>
</tbody>
</table>

**Interactive Diagram**

```
Global frame

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Assignment Statements

1. a = 1
2. b = 2
3. b, a = a + b, b

Interactive Diagram

Global frame

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
Assignment Statements

Just executed

1. a = 1
2. b = 2
3. b, a = a + b, b

Next to execute

Interactive Diagram
Assignment Statements

Execution rule for assignment statements:

```
1 a = 1
2 b = 2
3 b, a = a + b, b
```

Global frame

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
</tbody>
</table>
Assignment Statements

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
**Assignment Statements**

**Execution rule for assignment statements:**

1. Evaluate all expressions to the right of `=` from left to right.
2. Bind all names to the left of `=` to the resulting values in the current frame.
Assignment Statements

Execution rule for assignment statements:

1. Evaluate all expressions to the right of $=$ from left to right.

2. Bind all names to the left of $=$ to the resulting values in the current frame.
Assignment Statements

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to the resulting values in the current frame.
Discussion Question 1 Solution

(Demo)

Interactive Diagram
Discussion Question 1 Solution

1  f = min
2  f = max
3  g, h = min, max
4  max = g
5  max(f(2, g(h(1, 5), 3)), 4)

(Demo)
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

(Demo)

Interactive Diagram
Discussion Question 1 Solution

```
1  f = min
2  f = max
3  g, h = min, max
4  max = g
5  max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

Interactive Diagram
Discussion Question 1 Solution

```plaintext
1  f = min
2  f = max
3  g, h = min, max
4  \textcolor{red}{\textbf{max}} = g
5  \textcolor{blue}{\textbf{max}}(f(2, g(h(1, 5), 3)), 4)

func min(...)
func max(...)
```

(Demo)
Discussion Question 1 Solution

```plaintext
1  f = min
2  f = max
3  g, h = min, max
4  max = g
5  max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

Interactive Diagram

(Demo)

Global frame

- \( f \)
- \( h \)
- \( g \)
- \( \text{max} \)

- \( \text{func max}(\ldots) \)
- \( \text{func min}(\ldots) \)

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \max = g \)
5. \( \max(f(2, g(h(1, 5), 3)), 4) \)

(Demo)

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

(Demo)

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

(Demo)

Interactive Diagram
Discussion Question 1 Solution

```
1 f = min
2 f = max
3 g, h = min, max
   4 max = g
   5 max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

```
Global frame
   f
   h
   g
   max
```

Interactive Diagram
Discussion Question 1 Solution

1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

Interactive Diagram
Discussion Question 1 Solution

```python
1 f = min
2 f = max
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

Interactive Diagram
Discussion Question 1 Solution

```
1 f = min
2 f = max
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

Interactive Diagram
1. \( f = \text{min} \)
2. \( f = \text{max} \)
3. \( g, h = \text{min}, \text{max} \)
4. \( \text{max} = g \)
5. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

Interactive Diagram

(Demo)
Discussion Question 1 Solution

```
1 f = min
2 f = max
3 g, h = min, max
4 max = g
5 max(f(2, g(h(1, 5), 3)), 4)
```

(Demo)

```
Global frame
```

Interactive Diagram
Discussion Question 1 Solution

1. \(f = \text{min}\)
2. \(f = \text{max}\)
3. \(g, h = \text{min}, \text{max}\)
4. \(\text{max} = g\)
5. \(\text{max}(f(2, g(h(1, 5), 3)), 4)\)

The diagram illustrates the process of evaluating expressions involving \(f\), \(g\), and \(h\) with \(\text{min}\) and \(\text{max}\) functions. The interactive diagram shows how the values are calculated step by step.
Defining Functions
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions
Defining Functions

Assignment is a simple means of abstraction: binds names to values.

Function definition is a more powerful means of abstraction: binds names to expressions.

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```
Defining Functions

Assignment is a simple means of abstraction: binds names to values.

Function definition is a more powerful means of abstraction: binds names to expressions.

Function *signature* indicates how many arguments a function takes.

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions

Function **signature** indicates how many arguments a function takes

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```

Function **body** defines the computation performed when the function is applied
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions

Function *signature* indicates how many arguments a function takes

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```

Function *body* defines the computation performed when the function is applied

Execution procedure for def statements:
Defining Functions

Assignment is a simple means of abstraction: binds names to values.

Function definition is a more powerful means of abstraction: binds names to expressions.

Function signature indicates how many arguments a function takes.

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```

Function body defines the computation performed when the function is applied.

Execution procedure for def statements:

1. Create a function with signature `<name>(<formal parameters>)`
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions

Function signature indicates how many arguments a function takes

Function body defines the computation performed when the function is applied

Execution procedure for def statements:

1. Create a function with signature \(<name>(<formal\ parameters>)\):

   \[
   \text{def } <name>(<formal\ parameters>):
   \]

   \[
   \text{return } <return\ expression>
   \]

2. Set the body of that function to be everything indented after the first line
Defining Functions

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to expressions

>>> def <name>(<formal parameters>):
    return <return expression>

Function *signature* indicates how many arguments a function takes

Function *body* defines the computation performed when the function is applied

Execution procedure for def statements:

1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame
Calling User-Defined Functions

Interactive Diagram
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```python
from operator import mul

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

square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```python
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```python
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```
### Calling User-Defined Functions

**Procedure for calling/applying user-defined functions (version 1):**

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```
Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```python
1 from operator import mul
2 def square(x):
    return mul(x, x)
3 square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):
1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4     square(-2)
```
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
1 from operator import mul
2 def square(x):
3   return mul(x, x)
4 square(-2)
```
Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):
1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```
from operator import mul

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

square(-2)
```

A function’s signature has all the information needed to create a local frame

Interactive Diagram
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

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

A function’s signature has all the information needed to create a local frame
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

```python
from operator import mul

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

A function’s signature has all the information needed to create a local frame

Interactive Diagram
Looking Up Names In Environments
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

• The global frame alone, or
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
• The global frame alone, or
• A local frame, followed by the global frame.
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
• The global frame alone, or
• A local frame, followed by the global frame.

_Most important two things I’ll say all day:_
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
- The global frame alone, or
- A local frame, followed by the global frame.

Most important two things I’ll say all day:

An environment is a sequence of frames.
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
• The global frame alone, or
• A local frame, followed by the global frame.

**Most important two things I’ll say all day:**

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
• The global frame alone, or
• A local frame, followed by the global frame.

Most important two things I’ll say all day:
An environment is a sequence of frames.
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the square function:
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
- The global frame alone, or
- A local frame, followed by the global frame.

*Most important two things I’ll say all day:*

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the square function:
- Look for that name in the local frame.
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
- The global frame alone, or
- A local frame, followed by the global frame.

**Most important two things I’ll say all day:**

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the square function:
- Look for that name in the local frame.
- If not found, look for it in the global frame.
  (Built-in names like “max” are in the global frame too, but we don’t draw them in environment diagrams.)
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:
- The global frame alone, or
- A local frame, followed by the global frame.

Most important two things I’ll say all day:

An environment is a sequence of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the square function:
- Look for that name in the local frame.
- If not found, look for it in the global frame.
  (Built-in names like “max” are in the global frame too, but we don’t draw them in environment diagrams.)

(Demo)
Print and None

(Demo)
None Indicates that Nothing is Returned
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful: `None` is not displayed* by the interpreter as the value of an expression.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python

A function that does not explicitly return a value will return `None`

*Careful: None is not displayed* by the interpreter as the value of an expression

```python
>>> def does_not_square(x):
...    x * x
...```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...    x * x
...    # No return
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful*: `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
... 
>>> does_not_square(4)
```

No return.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful: None is not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
...     # No return

>>> does_not_square(4)
None value is not displayed
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python. A function that does not explicitly return a value will return `None`.

*Careful*: `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
...     # No return

>>> does_not_square(4)
None value is not displayed

>>> sixteen = does_not_square(4)
```
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful: `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
...     # No return

>>> does_not_square(4)

>>> sixteen = does_not_square(4)
```

The name `sixteen` is now bound to the value `None`.

`None` value is not displayed.
None Indicates that Nothing is Returned

The special value `None` represents nothing in Python.

A function that does not explicitly return a value will return `None`.

*Careful:* `None` is *not displayed* by the interpreter as the value of an expression.

```python
>>> def does_not_square(x):
...     x * x
...     # No return

>>> does_not_square(4)
None value is not displayed

>>> sixteen = does_not_square(4)

>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

The name `sixteen` is now bound to the value `None`. 
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

abs
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ \( \text{abs} \)

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

-2 ➔ abs ➔ 2

**Non-Pure Functions**
have side effects
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

**Non-Pure Functions**
have side effects
Pure Functions & Non-Pure Functions

**Pure Functions**  
*just return values*

**Non-Pure Functions**  
*have side effects*
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

- Argument: `-2`
  - `abs`
  - Return value: `2`

- Argument: `2, 100`
  - `pow`

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ `abs` ➔ 2

2, 100 ➔ `pow` ➔ 2

**Non-Pure Functions**
*have side effects*
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Argument
-2
abs
Return value
2

pow
2, 100

126765060022829401496703205376

Non-Pure Functions
have side effects
Pure Functions & Non-Pure Functions

Pure Functions
just return values

Non-Pure Functions
have side effects

Argument

Return value

-2

abs

2

2, 100

pow

126765060022829401496703205376

print
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ▶ abs
  ▶ 2

2, 100 ▶ pow
  ▶ 126765060022829401496703205376

**Non-Pure Functions**
*have side effects*

-2 ▶ print
Pure Functions & Non-Pure Functions

Pure Functions
just return values

-2 ➤ abs ➤ 2

Argument

2, 100 ➤ pow ➤ 1267650600228229401496703205376

2 Arguments

Non-Pure Functions
have side effects

-2 ➤ print ➤ None
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ 2

2, 100 ➔ pow ➔ 126765060022829401496703205376

**Non-Pure Functions**
*have side effects*

-2 ➔ print ➔ None

*Python displays the output “–2”*
Pure Functions & Non-Pure Functions

**Pure Functions**

*just return values*

-2 → abs → 2

<table>
<thead>
<tr>
<th>Argument</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 100</td>
<td>126765060022829401496703205376</td>
</tr>
</tbody>
</table>

| 2 Arguments |

**Non-Pure Functions**

*have side effects*

-2 → print → None

Python displays the output “-2”
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ **abs** ➔ Return value ➔ 2

**Argument**

2, 100 ➔ **pow** ➔ 1267650600228229401496703205376

**2 Arguments**

**Non-Pure Functions**
*have side effects*

-2 ➔ **print** ➔ Returns None!

**Python displays the output “–2”**

A side effect isn't a value; it's anything that happens as a consequence of calling a function
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None

print(print(1), print(2))
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```

print(print(1), print(2))
Nested Expressions with Print

```python
func print(...)
print(print(1), print(2))

>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```python
>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

>>> print(print(1), print(2))
1
2
None None
Nested Expressions with Print

```python
print(print(1), print(2))
```

```
1
2
None None
```
Nested Expressions with Print

```python
print(print(1), print(2))

>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

```
None, None → print(...):
    None
    display "None None"

>>> print(print(1), print(2))
1
2
None None
```
Nested Expressions with Print

None, None ➔ print(...):

display “None None”

Does not get displayed

>>> print(print(1), print(2))
1
2
None None

1 ➔ print(...):

display “1”

2 ➔ print(...):

display “2”