Announcements
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  • All homework is graded on effort; you must make progress on each problem to earn 2/2
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  • All homework is graded on effort; you must make progress on each problem to earn 2/2
  • Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
  ▪ 2 locations in Bechtel Engineering Center (Map: http://goo.gl/dAcHXf)
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
  ▪ 2 locations in Bechtel Engineering Center (Map: http://goo.gl/dAcHXf)
  ▪ 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  - All homework is graded on effort; you must make progress on each problem to earn 2/2
  - Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
  - 2 locations in Bechtel Engineering Center (Map: http://goo.gl/dAcHXf)
  - 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday

• You need to register a class account (Lab 0); that's how we track assignments
Announcements

• Starting next week, submitting labs & attending section will provide a midterm safety net

• Homework 1 is due next Wednesday 1/28
  ▪ All homework is graded on effort; you must make progress on each problem to earn 2/2
  ▪ Homework Party on Tuesday 1/27 5–6:30pm in 2050 VLSB

• Quiz 1 released next Wednesday 1/28 is due next Thursday 1/29 (graded on correctness)

• Ask questions about lab and homework assignments in office hours! (cs61a.org/weekly.html)
  ▪ 2 locations in Bechtel Engineering Center (Map: http://goo.gl/dAcHXf)
  ▪ 11–2 & 3–5 on Monday, 11–6 on Tuesday & Thursday, 11–2 & 3–4 on Wednesday, 11–1 on Friday

• You need to register a class account (Lab 0); that's how we track assignments
  ▪ Please register even if you're on the waitlist or applying for concurrent enrollment
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

add

Number or Numeral

Name
Types of Expressions

Primitive expressions:

- 2
  - Number or Numeral

- add
  - Name

- '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'`

**Number or Numeral**

**Name**

**String**

**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:**

- \(2\)
- \(\text{add}\)
- \('\text{hello}'\)

- **Number or Numeral:** \(2\)
- **Name:** \(\text{add}\)
- **String:** \('\text{hello}'\)

**Call expressions:**

- \(\text{max}\)
- \((2, 3)\)

- **Operator:** \(\text{max}\)
- **Operand 1:** \(2\)
- **Operand 2:** \(3\)
Types of Expressions

**Primitive expressions:**

- 2
- `add`
- 'hello'

- **Number or Numeral**
- **Name**
- **String**

**Call expressions:**

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

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

**Primitive expressions:**

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

**Call expressions:**

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

An operand can also be a call expression.
Types of Expressions

**Primitive expressions:**

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

- **Number or Numeral**
- **Name**
- **String**

**Call expressions:**

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

- **Operator**
- **Operand**
- **Operand**

An operand can also be a call expression.
Discussion Question 1
Discussion Question 1

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

5
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
```
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)
```

```python
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 * pi

Global frame
pi 3.1416

Code (left): Frames (right):

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

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

Global frame

pi 3.1416

Code (left):
Statements and expressions

Frames (right):
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Code (left):

Statements and expressions

Frames (right):

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Code (left):

Statements and expressions

Frames (right):

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**

Statements and expressions

Arrows indicate evaluation order

**Frames (right):**

Global frame

| pi | 3.1416 |

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

<table>
<thead>
<tr>
<th>Just executed</th>
<th>Import statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>from math import pi</code></td>
</tr>
<tr>
<td>2</td>
<td><code>tau = 2 * pi</code></td>
</tr>
</tbody>
</table>

**Code (left):**

Statements and expressions

**Frames (right):**

Global frame

pi | 3.1416

Arrows indicate evaluation order
Environment Diagrams

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 visualize the interpreter’s process.

**Code (left):**

Statements and expressions

Arrows indicate evaluation order

**Frames (right):**

Each name is bound to a value
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.

**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 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)
Assignment Statements

Interactive Diagram
Assignment Statements

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

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

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

Global frame

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

Interactive Diagram
Assignment Statements

Just executed

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

Next to execute

Global frame

| a | 1 |
| b | 2 |
Assignment Statements

Execution rule for assignment statements:

Interactive Diagram
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 those 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 those 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 those 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)
```
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. \( \max(\max(2, g(h(1, 5), 3)), 4) \)

(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. \( g, h = \text{min}, \text{max} \)

   \( \rightarrow 4 \) \( \text{max} = g \)

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

\( \text{func min(...)} \)

\( f(2, g(h(1, 5), 3)) \)

(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
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)

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

Interactive Diagram
Discussion Question 1 Solution

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

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

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

(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)`

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

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

```java
func min(...)
1 f = min
2 f = max
3 g, h = min, max
→ 4 max = g
→ 5 max(f(2, g(h(1, 5), 3)), 4)

func min(...)

func max(...)

(Demo)

Interactive Diagram

Global frame

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

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

Function signature indicates how many arguments a function takes
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

```
>>> 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

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

```
>>> 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>)`
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
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

>>> 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>)
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/applyng 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
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
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)
```

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)
```
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

```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

```
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)
```

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

```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

Interactive Diagram

```
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

```
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
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.

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

>>> does_not_square(4)  # No return

>>> sixteen = 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)

The name `sixteen` is now bound to the value `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

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

>>> does_not_square(4)
None

The name *sixteen* is now bound to the value *None*

>>> 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'
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*

-2 ➤ abs

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

**Pure Functions**

*just return values*

\[-2 \xrightarrow{\text{abs}} 2\]

**Non-Pure Functions**

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

**Pure Functions**
just return values

-2 → abs → 2

**Argument**

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

**Pure Functions**
*just return values*

-2 → \text{abs} → \text{Return value} → 2

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

**Pure Functions**
*just return values*

-2 \(\rightarrow\) abs

**Argument**

-2 \(\rightarrow\) pow

**Return value**

2

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

**Pure Functions**
*just return values*

-2 ➔ `abs`
  Argument ➔ Return value ➔ 2

2, 100 ➔ `pow`

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

**Pure Functions**
*just return values*

-2 $\xrightarrow{\text{abs}}$ 2

2, 100 $\xrightarrow{\text{pow}}$

2 Arguments

**Return value**

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

**Pure Functions**
*just return values*

- **Argument**
  - 2
  - abs
  - 2

- **Argument**
  - 2, 100
  - pow
  - 126765060022829401496703205376

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

**Pure Functions**  
*just return values*

-2 ➔ abs ➔ 2

2, 100 ➔ pow ➔ 1267650600228229401496703205376

2 Arguments

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

print
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

-2 ➔ abs ➔ 2

2, 100 ➔ pow ➔ 1267650600228229401496703205376

2 Arguments

**Non-Pure Functions**
have side effects

-2 ➔ print ➔

Argument

Return value
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ `abs`
  ➔ 2

2 ➔ `pow`
  ➔ 1267650600228229401496703205376

**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` ➔ 1267650600228229401496703205376

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

-2 ➔ `print` ➔ None

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

**Pure Functions**

*just return values*

- `abs(-2) → 2`
- `pow(2, 100) → 1267650600228229401496703205376`

**Non-Pure Functions**

*have side effects*

- `print(-2)`
  - `Returns None!`
- `Python displays the output “−2”`
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 → abs → 2

2, 100 → pow → 1267650600228229401496703205376

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

-2 → print → 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
>>> print(print(1), print(2))
1
2
None None
```

```
func print(…)
```
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
```
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

```
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”

---

None ➔ None

---

print(print(1), print(2)) ➔ None

---

1 ➔ 1 ➔ print(...):

---

display “1”

---

2 ➔ 2 ➔ print(...):

---

display “2”

---

None ➔ None, None ➔ None

---

None ➔ None

---

None ➔ None

---

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

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

None → print(print(1), print(2))
  → None
  → display "None None"

func print(...)

1 → print(1)
  → func print(...)
  → 1
  → display "1"

2 → print(2)
  → func print(...)
  → 2
  → display "2"
Nested Expressions with Print

None, None ▶ print(...):
  ▶ None
  ▶ Does not get displayed
  ▶ display “None None”

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

func print(...)

print(print(1), print(2))

None

func print(...)

print(1)

1

func print(...)

1

print(...):
  ▶ None
  ▶ display “1”

func print(...)

print(2)

2

func print(...)

2

print(...):
  ▶ None
  ▶ display “2”