Names, Assignment, and User-Defined Functions

(Demo)
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
- add
- 'hello'

- Number or Numeral
- Name
- 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'

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

**Call expressions:**
- `max`
- ( 2 , 3 )

**Operator**
Types of Expressions

**Primitive expressions:**

- Number or Numeral
- Name
- String

**Call expressions:**

- max
- Operator
- Operand
- ,
- Operand
- 2
- ,
- 3
- )
Types of Expressions

**Primitive expressions:**
2
add
'hello'

Number or Numerical
Name
String

**Call expressions:**
\[
\text{max}(2, 3)\]

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

Primitive expressions:

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

Call expressions:

- Operator: max
- Operand 1: 2
- 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'`

- Number or Numeral
- Name
- String

**Call expressions:**

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

- Operator
- Operand
- Operand

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
```
Discussion Question 1

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

```python
>>> f = min

>>> f = max

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

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
```

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)
```

4
Discussion Question 1

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

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

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

```python
>>> ???
```
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.

Example: http://goo.gl/J2W5NL
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

Example: http://goo.gl/J2W5NL
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

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

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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`  

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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):

Example: http://goo.gl/J2W5NL
Environment diagrams visualize the interpreter’s process.

**Code (left):**
```
1  from math import pi
2  tau = 2 * pi
```

**Frames (right):**
```
Global frame
pi 3.1416
```

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Example: http://goo.gl/J2W5NL

Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

Global frame

pi | 3.1416
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

Just executed

Import statement

1. from math import pi

2. tau = 2 * pi

Global frame

pi 3.1416

Example: http://goo.gl/J2W5NL
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 |

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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

Example: http://goo.gl/J2W5NL
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

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
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

Example: [http://goo.gl/J2W5NL](http://goo.gl/J2W5NL)
Assignment Statements

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)
Assignment Statements

Example: \url{http://goo.gl/Ppn26M}

```
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

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)

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

Global frame

```
  a  1
  b  2
```
Assignment Statements

Example:

```
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>

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)
Assignment Statements

Execution rule for assignment statements:

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)
Assignment Statements

Execution rule for assignment statements:

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

Example: [link](http://goo.gl/Ppn26M)
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.

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)
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.

Example: http://goo.gl/Ppn26M
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.

Example: [http://goo.gl/Ppn26M](http://goo.gl/Ppn26M)
Discussion Question 1 Solution

(Demo)

Example: http://goo.gl/BGLafR
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

(Demo)

```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)
```

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
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)
```

Example: [http://goo.gl/BGLafR](http://goo.gl/BGLafR)
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)
```

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafR](http://goo.gl/BGLafR)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafR](http://goo.gl/BGLafR)
Discussion Question 1 Solution

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

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

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
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)

Example: [http://goo.gl/BGLafR](http://goo.gl/BGLafR)
Discussion Question 1 Solution

(Demo)

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

```
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)
```

Example: http://goo.gl/BGLafR
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

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

Example: [http://goo.gl/BGLafB](http://goo.gl/BGLafB)
Discussion Question 1 Solution

In this solution, we are dealing with a series of function calls involving `min` and `max` functions. The code snippet given is as follows:

```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)
```

Here, `min` and `max` are predefined functions. The code calculates the maximum value among the arguments passed to `max` function.

Example: [Link](http://goo.gl/BGLafR)
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 computational process expressed by a function.
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 \textit{signature} indicates how many arguments a function takes

\begin{verbatim}
>>> def <name>(<formal parameters>):
    return <return expression>
\end{verbatim}

Function \textit{body} defines the computational process expressed by a function

Execution procedure for \texttt{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 \textit{signature} indicates how many arguments a function takes.

\begin{verbatim}
>>> def \texttt{name}(\texttt{formal parameters}): 
    \texttt{return return expression}
\end{verbatim}

Function \textit{body} defines the computational process expressed by a function.

**Execution procedure for def statements:**
1. Create a function with signature \texttt{name}(\texttt{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

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

Function **body** defines the computational process expressed by a function

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

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

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

Example: http://goo.gl/GXYdCP
Calling User-Defined Functions

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

Example: http://goo.gl/GXYdCP
Calling User-Defined Functions

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

1. Add a local frame, forming a new environment

Example: http://goo.gl/GXYdCP
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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
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

Example: http://goo.gl/GXYdCP
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

Example:

```python
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(-2)
```
Procedure for calling/apply 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

Example:

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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
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

Example:

```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

Example:

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

Example: http://goo.gl/GXYdCP
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

Example:

```
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

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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
Calling User-Defined Functions

Procedure for calling/apply 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

Example: http://goo.gl/GXYdCP
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

Example: http://goo.gl/GXYdCP
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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
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

Example:

```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

Example: [http://goo.gl/GXYdCP](http://goo.gl/GXYdCP)
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.)
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)
The Print Function

(Demo)
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 \( \to \) \( \text{abs} \)

**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
- Function: abs
- Return value: 2

**Non-Pure Functions**

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

**Pure Functions**
*just return values*

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

**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*

- Argument
  - `-2` → `abs` → Return value `2`
  - `2, 100` → `pow` → 2 Arguments

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

**Pure Functions**

*just return values*

-2 ➔ \textit{abs} ➔ 2

2, 100 ➔ \textit{pow} ➔ 1267650600228229401496703205376

**Non-Pure Functions**

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

**Pure Functions**
*just return values*

-2 → abs → 2

2, 100 → pow → 126750600228229401496703205376

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

print
Pure Functions & Non-Pure Functions

**Pure Functions**

*just return values*

-2 → abs → 2

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

<table>
<thead>
<tr>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1267650600228229401496703205376</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Arguments</th>
</tr>
</thead>
</table>

**Non-Pure Functions**

*have side effects*

-2 → print
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

-2 ➔ \( \text{abs} \) ➔ 2

Argument

2, 100 ➔ \( \text{pow} \) ➔ 1267650600228229401496703205376

2 Arguments

**Non-Pure Functions**

have side effects

-2 ➔ \( \text{print} \) ➔ None
Pure Functions & Non-Pure Functions

### Pure Functions
*just return values*

- **Argument**: $-2$
- **Function**: `abs`
- **Return value**: 2

- **Argument**: $2, 100$
- **Function**: `pow`
- **Return value**: $1267650600228229401496703205376$

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

- **Argument**: $-2$
- **Function**: `print`
- **Return value**: None

*Python displays the output "-2"*
Pure Functions & Non-Pure Functions

**Pure Functions**

*just return values*

-2 \( \rightarrow \) abs \( \rightarrow \) 2

-2 \( \rightarrow \) pow \( \rightarrow \) 1267650600228229401496703205376

**Non-Pure Functions**

*have side effects*

-2 \( \rightarrow \) print \( \rightarrow \) None

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

**Pure Functions**
*just return values*

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

- **2 Arguments**
  - `[2, 100]` ➔ *pow*
  - `1267650600228229401496703205376` ➔ Return value

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

- **Argument**
  - `-2` ➔ *print*
  - `None` ➔ 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.

- `print` (-2)

- `abs` (-2)

- `pow` (2, 100)

- `pow` (-2)

- `print` (None)
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
  - Returns None!

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

*Python displays the output “-2”*

**Important:** The interactive interpreter (>>>) displays the value of an expression, unless it is 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

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

```
func print(...)

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

```
func print(...):  # display "1"
    1
    print(...):  # display 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))
1
2
None
None
```
Nested Expressions with Print

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

The diagram illustrates the execution of the nested `print` function calls:

1. `print(print(1), print(2))` is the root function call.
2. The first `print` function call is `print(1)`.
   - The `print` function is called with `print(...)`, and the result is `1`.
   - The `print` function is called again with `print(...)`, and the result is `None`.
   - Displaying "1".
3. The second `print` function call is `print(2)`.
   - The `print` function is called with `print(...)`, and the result is `2`.
   - The `print` function is called again with `print(...)`, and the result is `None`.
   - Displaying "2".
Nested Expressions with Print

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

1
```
def print(...):
    display "1"
```

2
```
def print(...):
    display "2"
```
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(...):
   ➔ None

display “None None”

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

1 ➔ print(...):
   ➔ None

display “1”

2 ➔ print(...):
   ➔ None

display “2”
```
Nested Expressions with Print

```python
None, None > print(...):
    None

    display “None None”

None

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

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

func print(...)

None

print(1)

func print(...)

1

print(...):
    None

    display “1”

func print(...)

2

print(...):
    None

    display “2”
```
Nested Expressions with Print

```python
None, None

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

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

does not get displayed
```