Names, Assignment, and User-Defined Functions

(Demo)
Types of Expressions
Types of Expressions

Primitive expressions:
Types of Expressions

**Primitive expressions:**

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

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

**Call expressions:**

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

**Primitive expressions:**

- 2 (Number or Numeral)
- add (Name)
- 'hello' (String)

**Call expressions:**

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

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

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

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

**Primitive expressions:**

- 2
- add
- 'hello'

- Number or Numeral
- Name
- String

**Call expressions:**

- \( \text{max} \) (2, 3)
- \( \text{max}(\text{min}(\text{pow}(3, 5), -4), \text{min}(1, -2)) \)

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

**Primitive expressions:**

- 2
- add
- 'hello'

- Number or Numeral
- Name
- String

**Call expressions:**

- max
- (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
```
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
```

```python
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
>>> 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 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. \( \text{tau} = 2 \times pi \)

Global frame

| pi | 3.1416 |

Code (left):          Frames (right):
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):

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):

Global frame

pi 3.1416

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**

Statements and expressions

**Frames (right):**

Global frame

| pi  | 3.1416 |

Interactive Diagram
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):

Just executed

Import statement

Global frame
pi 3.1416

1 from math import pi

2 tau = 2 * pi

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.

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

Global frame
- `pi` 3.1416

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

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

Interactive Diagram
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Just executed

Import statement

1 from math import pi

2 tau = 2 * pi

Next to execute

Assignment statement

Global frame

Name

pi 3.1416

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
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
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

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

Interactive Diagram
Assignment Statements

1. \( a = 1 \)
2. \( b = 2 \)
3. \( b, a = a + b, b \)

Interactive Diagram: Global frame
\[
\begin{array}{c|c}
  a & 1 \\
  b & 2 \\
\end{array}
\]
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

Just executed

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

Next to execute

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:
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)

(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)
Discussion Question 1 Solution

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

(Demo)

Global frame

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

Interactive Diagram
Discussion Question 1 Solution

\begin{align*}
1 & \quad f = \text{min} \\
2 & \quad f = \text{max} \\
3 & \quad g, h = \text{min}, \text{max} \\
4 & \quad \text{max} = g \\
5 & \quad \text{max}(f(2, g(h(1, 5), 3)), 4)
\end{align*}

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

def f(x):
    if x < 5:
        return min(x, 5)
    else:
        return max(x, 5)

g, h = min, max
max = g

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

(hero)

**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)

Global frame

- func max(…)
- func min(…)

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]
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 \textit{signature} indicates how many arguments a function takes.

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

Function \textit{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.

>>> 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

```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>)`
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/applying user-defined functions (version 1):

1. Add a local frame, forming a new environment

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
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
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)
```

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
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)
```
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
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)
```

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

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

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
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

Interactive Diagram
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`.

*Cautious:* `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)
```

```text
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 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
...     return None
>>> 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  
>>>     ...

>>> does_not_square(4)
```

No return

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

```python
>>> sixteen = does_not_square(4)
>>> sixteen
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
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 \( \rightarrow \) \( \text{abs} \)

**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*

-2 → **abs** → 2

-2 → **pow**

**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`

- **2 Arguments**
  - `2, 100`  
  - `pow`

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

Pure Functions
*just return values*

-2 ➔ abs ➔ 2

- 2 Arguments

Non-Pure Functions
*have side effects*

2, 100 ➔ pow ➔ 1267650600228229401496703205376

- 2 Arguments
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*

- `print`
Pure Functions & Non-Pure Functions

**Pure Functions**
just return values

-2 ➔ abs ➔ 2

2, 100 ➔ pow ➔ 126765060022829401496703205376

2 Arguments

**Non-Pure Functions**
have side effects

-2 ➔ print

Argument
Pure Functions & Non-Pure Functions

**Pure Functions**
*just return values*

-2 ➔ abs ➔ 2

**Argument**

2, 100 ➔ pow ➔ 12676506022829401496703205376

**Return value**

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

-2 ➔ print ➔ None

**2 Arguments**
Pure Functions & Non-Pure Functions

**Pure Functions**  
just return values

<table>
<thead>
<tr>
<th>Argument</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Non-Pure Functions**  
have side effects

<table>
<thead>
<tr>
<th>Argument</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 100</td>
<td>1267650600228229401496703205376</td>
</tr>
</tbody>
</table>

-2 ▶ print ▶ None

Python displays the output “–2”
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 ➔ Returns None!

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

**Pure Functions**
*just return values*

- **Argument**
  - `-2` → `abs`
  - `2`

- **Argument**
  - `2, 100` → `pow`
  - `1267650600228229401496703205376`

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

- **Argument**
  - `-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
```

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

```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
```

Diagram:

```
print(print(1), print(2))
  |   |
  v   v
func print(...) None
  |   |
  v   v
print(1)  print(2)
    |     |
    v     v
func print(...) 1 2
  |   |   |   |
  v   v   v   v
print(...) 1 2
  |   |   |
  v   v   v
1 2
  |
  v
None None
```

display “1”
display “2”
Nested Expressions with Print

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

1
2
None None

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

None, None ➔ print(...):

None ➔ display “None None”

call print(print(1), print(2)) ➔ None

None ➔ print(print(1), print(2)) ➔ None

None ➔ print(print(1)) ➔ print(1) ➔ func print(...) ➔ None ➔ display “1”

None ➔ print(print(2)) ➔ print(2) ➔ func print(...) ➔ None ➔ display “2”

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

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

display “None None”
Nested Expressions with Print

None, None

print(...):

None

Does not get displayed

display “None None”

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