Functions
Announcements
Expressions
Types of expressions
Types of expressions

An expression describes a computation and evaluates to a value
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 = 87 \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sqrt{3493161} \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \sqrt{3493161} \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \sqrt{3493161} \]

| – 1869|
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \quad \frac{6}{23} \quad \sin \pi \quad \sqrt{3493161} \quad \sum_{i=1}^{100} i \quad | - 1869 | \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \sqrt{3493161} \]
\[ \sum_{i=1}^{100} i \]
\[ | - 1869| \]
\[ (69) \]
\[ (18) \]
Types of expressions

An expression describes a computation and evaluates to a value

\[
18 + 69 \quad 6 \quad \sin \pi \\
\frac{6}{23} \\
\sum_{i=1}^{100} i \\
\sqrt{3493161} \\
\text{max}(-1869) \\
(69) \\
(18)
\]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ 2^{100} \]
\[ f(x) \]
\[ \sum_{i=1}^{100} i \]
\[ | -1869| \]
\[ \sqrt{3493161} \]
\[ (69) \]
\[ (18) \]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \log_2 1024 \]
\[ 2^{100} \]
\[ f(x) \]
\[ \sqrt{3493161} \]
\[ \sum_{i=1}^{100} i \]
\[ | - 1869 | \]

\[ \binom{69}{18} \]
Types of expressions

An expression describes a computation and evaluates to a value

\[
\begin{align*}
18 + 69 &= 6 \quad \frac{23}{6} \quad \sin \pi \\
2^{100} &= \sum_{i=1}^{100} i \\
f(x) &= \sqrt{3493161} \\
7 \mod 2 &= | -1869 |
\end{align*}
\]
Types of expressions

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \log_2 1024 \]
\[ 2^{100} \]
\[ f(x) \]
\[ 7 \mod 2 \]
\[ \sum_{i=1}^{100} i \]
\[ | - 1869| \]
\[ \sqrt{3493161} \]
\[ \lim_{x \to \infty} \frac{1}{x} \]
Types of expressions

An expression describes a computation and evaluates to a value.

\[ 18 + 69 \]
\[ \frac{6}{23} \]
\[ \sin \pi \]
\[ \log_2 1024 \]
\[ 2^{100} \]
\[ f(x) \]
\[ 7 \mod 2 \]
\[ | - 1869| \]
\[ \sum_{i=1}^{100} i \]
\[ \sqrt{3493161} \]
\[ \lim_{x \to \infty} \frac{1}{x} \]
\[ \binom{69}{18} \]
All expressions can use function call notation

(Demo)
Anatomy of a Call Expression
Anatomy of a Call Expression

```
add ( 2 , 3 )
```
Anatomy of a Call Expression

```
add ( 2 , 3 )
```
Anatomy of a Call Expression

add ( 2 , 3 )

Operator
Anatomy of a Call Expression

\[
\text{add} \quad ( \quad 2, \quad 3 \quad )
\]

Operator \quad Operand \quad Operand
Anatomy of a Call Expression

Operators and operands are also expressions
Anatomy of a Call Expression

Operators and operands are also expressions

So they evaluate to values
Anatomy of a Call Expression

Evaluation procedure for call expressions:

Operators and operands are also expressions

So they evaluate to values

Evaluation procedure for call expressions:
Anatomy of a Call Expression

Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions.
Anatomy of a Call Expression

Operators and operands are also expressions

So they evaluate to values

Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

mul(add(4, mul(4, 6)), add(3, 5))
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

mul(add(4, mul(4, 6)), add(3, 5))

mul(add(4, mul(4, 6)), add(3, 5))
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]

Diagram:
- `mul`
- `add`
Evaluating Nested Expressions

```
mul(add(4, mul(4, 6)), add(3, 5))
```

```
mul
```
```
add(4, mul(4, 6))
```
```
add
```
```
4
```
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[
mul(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul} \left( \text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5) \right) \]

\[ \text{mul} \]

\[ \text{add}(4, \text{mul}(4, 6)) \]

\[ \text{add} \quad 4 \quad 24 \]

\[ \text{mul}(4, 6) \]

\[ \text{mul} \quad 4 \quad 6 \]
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

```
mul(add(4, mul(4, 6)), add(3, 5))
```

```
mul
mul
add
mul
add
mul
mul
```
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]

\[ \text{mul}(28, 8) \]

\[ \text{add}(4, 24) \]

\[ \text{mul}(4, 6) \]

\[ \text{mul}(4, 6) \]

\[ \text{mul}(4, 6) \]
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[
mul(add(4, mul(4, 6)), add(3, 5))
\]

224

mul

28

add

4

mul(4, 6)

4

6

mul

24

add

3

5

add

8

add

3

5
Evaluating Nested Expressions

\[ \text{mul}\left(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)\right) \]
Evaluating Nested Expressions

Expression tree

\[ 224 = \text{mul}(\text{add}(4, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

Expression tree

Operand subexpression

224
mul(add(4, mul(4, 6)), add(3, 5))

mul

28
add(4, mul(4, 6))

add

4
24
mul(4, 6)

mul

4
6

8
add(3, 5)

add

3
5

Expression tree
Evaluating Nested Expressions

Expression tree

Operand subexpression

Value of subexpression

mul(add(4, mul(4, 6)), add(3, 5))

224

mul

add(4, mul(4, 6))

28

add

4

24

mul

mul(4, 6)

mul

4

6

add

3

5

8

Expression tree
Evaluating Nested Expressions
Evaluating Nested Expressions

Expression tree

Operand subexpression

Value of subexpression

1st argument to mul

Value of the whole expression

mul(add(4, mul(4, 6)), add(3, 5))

mul(add(4, mul(4, 6)), add(3, 5))

mul

add(4, mul(4, 6))

add(4, mul(4, 6))

add

4

24

mul(4, 6)

mul(4, 6)

mul

4

6

add

3

5

add

8

224
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
- add

Number or Numeral

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

**Call expressions:**
- \(\text{max} \ (\ 2 \ , \ 3 \ )\)
Types of Expressions

**Primitive expressions:**

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

**Call expressions:**

- **max**
  - Operator
  - (2, 3)

Types of Expressions

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

- Number or Numeral
- Name
- String

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

- Operator
- Operand
- Operand
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:**

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

**Call expressions:**

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

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
```

11
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
```

11
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)

11
```
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
Environment diagrams visualize the interpreter’s process.

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

**Code (left):**

**Frames (right):**

Global frame

- `pi`: 3.1416
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

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

**Code (left):**

Statements and expressions

**Frames (right):**

Global frame

<table>
<thead>
<tr>
<th>pi</th>
<th>3.1416</th>
</tr>
</thead>
</table>

[Source](http://pythontutor.com/composingprograms.html#code=from%20math%20import%20pi%0Atau%20%3D%202%20*%20pi&cumulative=false&curInstr=1&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D)
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Statements and expressions

Code (left):

Statements and expressions

Frames (right):
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**

Statements and expressions

**Frames (right):**

Import statement

1. `from math import pi`

Assignment statement

2. `tau = 2 * pi`

Global frame

- `pi`: 3.1416
Environment diagrams visualize the interpreter’s process.

Environment Diagrams

Statements and expressions

Arrows indicate evaluation order
Environment diagrams visualize the interpreter’s process.

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

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

**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.

Just executed

1. \texttt{from math import pi}

2. \texttt{tau = 2 * pi}

Import statement

Global frame

\texttt{pi} | 3.1416

Next to execute

Assignment statement

http://pythontutor.com/composingprograms.html?code=from\%20math\%20import\%20pi\n\tau\%20%3D\%202\%20*\%20pi&cumulative=false&curInstr=1&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Just executed

1. \texttt{from math import pi}

2. \texttt{tau = 2 * pi}

Global frame

\begin{array}{c}
\text{pi} \\
3.1416
\end{array}

Next to execute

Import statement

Assignment statement

Frames (right):

Each name is bound to a value

Code (left):

Statements and expressions

Arrows indicate evaluation order
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

Just executed

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

Import statement

Next to execute

Assignment statement

Global frame

pi | 3.1416

Name

Code (left):

Statements and expressions

Arrows indicate evaluation order

Frames (right):

Each name is bound to a value
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 illustrate the interpreter’s process. The interpreter is a program that executes code by binding names to values and following specific rules.

**Import statement:**

Each name is bound to a value

**Statements and expressions:**

Arrows indicate the evaluation order

**Frames (right):**

Global frame

Name  Value

pi   3.1416

**Just executed:**

1  from math import pi

2  tau = 2 * pi

**Next to execute:**

Assignment statement

Import statement
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 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

http://pythontutor.com/composingprograms.html?i=a%20=%201%0Ab%20=%202%0Ab,%20a%20=%20a%20%2B%20b,%20b
Assignment Statements

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

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

Global frame

- \( a \): 1
- \( b \): 2
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

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

Global frame

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

Execution rule for assignment statements:

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

Global frame

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

Execution rule for assignment statements:

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

Execution rule for assignment statements:

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

2. Bind all names to the left of = to 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)
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)
```
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. \( f = \min \)
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. \( g, h = \text{min}, \text{max} \)
   \[ \text{max} = g \]
4. \( \text{max}(f(2, g(h(1, 5), 3)), 4) \)

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

(Demo)
Discussion Question 1 Solution

```python
func min(
    f = min
    f = max
    g, h = min, max
    max = g
    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)

```
Global frame
f
h
min
max
f(2, g(h(1, 5), 3))
2
g(h(1, 5), 3)
```
Discussion Question 1 Solution

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

(Demo)
Discussion Question 1 Solution

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

(Demo)

Global frame
  func max(...)
    f
  h
  g
  max

func min(...)
  f(2, g(h(1, 5), 3))

func max(...)
  g(h(1, 5), 3)

func min(...)
  h(1, 5)

func max(...)
  1 5
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

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

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

\( f \)

\( h \)

\( g \)

\( \text{max} \)
Discussion Question 1 Solution

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

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

```
http://pythontutor.com/composingprograms.html?code=f%20%3D%20min%0Af%20%3D%20max%0Ag,%20h%20%3D%20min,%20max%0Amax%20%3D%20...%29,%203%29%29,%204%29&cumulative=false&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D
```
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

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

**Diagram:**

- **Global frame**
  - `func min(...)`
  - `func max(...)`
  - `g`
  - `h`
  - `max`

- **Function Calls**
  - `f(2, g(h(1, 5), 3))`
  - `g(h(1, 5), 3)`
  - `h(1, 5)`

**Evaluation Steps**

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)
Discussion Question 1 Solution

```python
func min(...) 3
f(2, g(h(1, 5), 3))

func max(...) 2
f(2, g(h(1, 5), 3))

func min(...) 5
h(1, 5)

func max(...) 1
5
```
function 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)

function min(...)

Demo

(Demo)

Global frame

f

h

g

max

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.

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

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

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

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

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

Function signature indicates how many arguments a function takes

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

```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):

```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
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)
```

![Diagram showing the process of calling a function, including the flow of variables and the function call graph.]
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

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
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
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
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
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
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

```
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

```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

![Diagram]
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.
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)