The Elements of Programming
• Primitive Expressions and Statements
  ▪ *The simplest building blocks of a language*
The Elements of Programming

- Primitive Expressions and Statements
  - *The simplest building blocks of a language*

- Means of Combination
  - *Compound elements are built from simpler ones*
The Elements of Programming

• Primitive Expressions and Statements
  ▪ The simplest building blocks of a language

• Means of Combination
  ▪ Compound elements are built from simpler ones

• Means of Abstraction
  ▪ Compound elements can be named and manipulated as units
The Elements of Programming

• Primitive Expressions and Statements
  ▪ *The simplest building blocks of a language*

• Means of Combination
  ▪ *Compound elements are built from simpler ones*

• Means of Abstraction
  ▪ *Compound elements can be named and manipulated as units*

Programming languages allow us to communicate, too
Data: Stuff we want to manipulate
Functions and Data

Data: Stuff we want to manipulate

Functions: Rules for manipulating data
Functions and Data

**Data:** Stuff we want to manipulate

**Functions:** Rules for manipulating data
Functions and Data

**Data:** Stuff we want to manipulate

“The Art of Computer Programming”

2

**Functions:** Rules for manipulating data
Functions and Data

**Data:** Stuff we want to manipulate

“The Art of Computer Programming”

*Donald Knuth*

**Functions:** Rules for manipulating data
Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

Functions: Rules for manipulating data
Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

2

Donald Knuth

This slide

Functions: Rules for manipulating data

Add numbers
Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”
2
Donald Knuth

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

This slide
Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Pronounce someone’s name

Donald Knuth

This slide
Functions and Data

**Data:** Stuff we want to manipulate

“The Art of Computer Programming”

2

Donald Knuth
(Ka-NOOTH)

**Functions:** Rules for manipulating data

*Count the words in a line of text*

*Add numbers*

*Pronounce someone’s name*
Functions and Data

Data: Stuff we want to manipulate

“The Art of Computer Programming”

Donald Knuth
(Ka-NOOTH)

Functions: Rules for manipulating data

Count the words in a line of text

Add numbers

Load the next slide

Pronounce someone’s name
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

\[18 + 69 = 6 \quad \text{and} \quad 23\]
Types of expressions

An expression describes a computation and evaluates to a value.

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

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

\[ \sum_{i=1}^{100} i \quad | - 1869| \]
Types of expressions

An expression describes a computation and evaluates to a value

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

An expression describes a computation and evaluates to a value

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

An expression describes a computation and evaluates to a value

\[ 18 + 69 \]

\[ \frac{6}{23} \]

\[ f(x) \]

\[ \sin \pi \]

\[ \sqrt{3493161} \]

\[ \left| -1869 \right| \]

\[ \sum_{i=1}^{100} i \]

\[ \binom{69}{18} \]
All expressions can use function call notation (Demo)
Anatomy of a Call Expression
Anatomy of a Call Expression

\[ \text{add} \ ( \ 2 \ , \ 3 \ ) \]
Anatomy of a Call Expression

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

*Operator*
Anatomy of a Call Expression

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

Operator  \hspace{2cm} \text{Operand 0}  \hspace{2cm} \text{Operand 1}
Anatomy of a Call Expression

```
add ( 2 , 3 )

Operator  Operand 0  Operand 1
```

Operators and operands are expressions

Monday, August 29, 2011
Anatomy of a Call Expression

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

Operator \quad \text{Operand } 0 \quad \text{Operand } 1

Operators and operands are expressions

So they evaluate to values
Anatomy of a Call Expression

**Evaluation procedure for call expressions:**

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

*Operator* \quad *Operand 0* \quad *Operand 1*

Operators and operands are expressions

So they evaluate to values
Anatomy of a Call Expression

Evaluation procedure for call expressions:

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

Evaluation procedure for call expressions:

1. Evaluate the operator and operand subexpressions

2. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpression
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[
\text{mul} (\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul} (\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[
\text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]

\[ \text{mul}(26, 24) \]

\[ \text{add}(2, \text{mul}(4, 6)) \]

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

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

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

\[
mul(add(2, mul(4, 6)), add(3, 5))
\]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]
Evaluating Nested Expressions

\[ \text{mul}(\text{add}(2, \text{mul}(4, 6)), \text{add}(3, 5)) \]

Diagram:

```
       208
      /     /
mul(add(2, mul(4, 6)), add(3, 5))
     /   /
    26   8
   /   /  /
mul(add(2, mul(4, 6)), add(3, 5))
  /   /  /
 mul(2, 4) mul(3, 5)
 /   /  /
add(2, mul(4, 6)) add(3, 5)
 /   /
 mul(4, 6) 7
 /   /
add(4, 6) 2
 /   /
 4   6
```
The Print Function

(Demo)
Pure Functions & Non-Pure Functions

Pure Functions

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

\[ \text{abs(number)}: \_
\]

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

Function signature: how many parameters

abs(number): __

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

-2 \( \rightarrow \) \text{abs(number)}: \_

Function signature: how many parameters

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

Function signature: how many parameters

-2 → abs(number):

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

\[-2 \xrightarrow{\text{abs(number)}} 2\]

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

\[-2 \xrightarrow{\text{abs(number):}} 2\]

\[\text{pow(x, y):}\]

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

-2 ➔ \text{abs(number)}: \text{2}

2, 100 ➔ \text{pow(x, y)}: \text{ }
Pure Functions & Non-Pure Functions

Pure Functions

-2 ▶ abs(number): ▶ 2

2, 100 ▶ pow(x, y): ▶ 1267650600228229401496703205376

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

-2 \[\text{abs(number)}: \]
\[\downarrow\]
\[2\]

2, 100 \[\text{pow(x, y)}: \]
\[\downarrow\]
\[1267650600228229401496703205376\]

Non-Pure Functions
Pure Functions & Non-Pure Functions

Pure Functions

-2 ▶ abs(number): ▶ 2

2, 100 ▶ pow(x, y): ▶ 1267650600228229401496703205376

Non-Pure Functions

print(...):
Pure Functions

-2 $\Rightarrow$ abs(number): $\Rightarrow$ 2

Only produces return values

2, 100 $\Rightarrow$ pow(x, y): $\Rightarrow$ 1267650600228229401496703205376

Non-Pure Functions

-2 $\Rightarrow$ print(...):
Pure Functions & Non-Pure Functions

Pure Functions

-2 ▶️ abs(number):
  ▼ 2

2, 100 ▶️ pow(x, y):
  ▼ 1267650600228229401496703205376

Non-Pure Functions

-2 ▶️ print(...):
  ▼ None
Pure Functions & Non-Pure Functions

Pure Functions

-2 ➤ $\text{abs(number)}$: ➤ 2

2, 100 ➤ $\text{pow(x, y)}$: ➤ $1267650600228229401496703205376$

Non-Pure Functions

-2 ➤ $\text{print(...)}$: ➤ None

display “–2”

Only produces return values
Pure Functions & Non-Pure Functions

Pure Functions

-2 ➔ abs(number): ➔ 2

Only produces return values

2, 100 ➔ pow(x, y): ➔ 1267650600228229401496703205376

Non-Pure Functions

-2 ➔ print(...): ➔ None

May create side effects

display “–2”
Nested Expressions with Print

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

\[\text{print(print(1), print(2))}\]
Nested Expressions with Print

\[
\text{print(print(1), print(2))}
\]
Nested Expressions with Print

\[
\text{print(\text{print(1)}, \text{print(2)})}
\]

\[
\text{print(1)}
\]
Nested Expressions with Print

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

1. `print(print(1), print(2))`:
   - `print(1)`
   - `print(2)`

2. `print(...):`
   - `None`

3. `display "1"`
Nested Expressions with Print

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

```
print(1)
```

```
1

print(...):
```

```
display "1"
```
Nested Expressions with Print

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

print

None
print(1)

print(2)

1 ➔ print(...):

None

display "1"
Nested Expressions with Print

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

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

```
print
None
print(1)
```

```
print
None
print(2)
```

```
1. print(...):
   None
   display "1"
```

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

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

```
1 ➤ `print(...):`
   ➤ None
   ➤ display “1”

2 ➤ `print(...):`
   ➤ None
   ➤ display “2”
```
Nested Expressions with Print

None, None ➔ \texttt{print(...):} ➔ None

\texttt{display “None None”}

\texttt{print(print(1), print(2))}

\texttt{print} ➔ None

\texttt{print(1)} ➔ None

\texttt{print(2)} ➔ None

1 ➔ \texttt{print(...):} ➔ None

\texttt{display “1”}

2 ➔ \texttt{print(...):} ➔ None

\texttt{display “2”}

Monday, August 29, 2011
Nested Expressions with Print

None, None ➔ print(...):

None ➔ None

display “None None”

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

print(print(1))

None ➔ None

display “1”

print(print(2))

None ➔ None

display “2”
Names and Assignment

(Demo)
Environments

\[
\begin{align*}
\text{abs:} & \\
\text{...} & \\
\text{max:} & \\
\text{...} & \\
\text{abs}(x): & \\
\text{max}(a, b, c, \ldots): & \\
\end{align*}
\]
Environments

Built-in function

abs:
...
max:
...

abs(x):

max(a, b, c, ...):
from math import pi

Environments

Built-in function

abs: ...

max: ...

abs(x):

max(a, b, c, ...):
Environments

Built-in function

abs: ...
max: ...
pi: 3.14...

abs(x):
max(a, b, c, ...):

from math import pi
Environments

```
from math import pi

pi: 3.14...

abs: ...

max: ...

abs(x):

max(a, b, c, ...):

from math import pi
```
from math import pi

tau = 2 * pi
Environments

- **Built-in function**
  - `abs(x)`: ...
  - `max(a, b, c, ...)`: ...

- **Imported value**
  - `pi`: 3.14...
  - `tau`: 6.28...

```python
from math import pi
tau = 2 * pi
```
Environments

- **Built-in function**
  - `abs`: 
  - `max`: 
    - `max(a, b, c, ...)`: 

- **Imported value**
  - `pi`: 3.14...
  - `tau`: 6.28...

- **Assigned value**
  - `from math import pi`
  - `tau = 2 * pi`

---

Monday, August 29, 2011
The environment does not track where names came from.
Environments

The environment does not track where names came from

```python
from math import pi
tau = 2 * pi
```

```
Environments

abs:  
...
max:  
...
pi:   3.14...
tau:   6.28...
```

```
from math import pi
```
Environments

The environment does not track where names came from

from math import pi

tau = 2 * pi
The environment does not track where names came from
Environments

A frame holds name bindings

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>3.14</td>
</tr>
<tr>
<td>tau</td>
<td>6.28</td>
</tr>
</tbody>
</table>

Binding to a numeric value

from math import pi

tau = 2 * pi

The environment does not track where names came from
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def <name>(<formal parameters>):
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def <name>(<formal parameters>):
    return <return expression>
```
User-Defined Functions

Named values are a simple means of abstraction

Named expressions are a more powerful means of abstraction

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

def expressions:
  • Create a new function
  • Bind a name to it
User-Defined Functions

Named values are a simple means of abstraction

Named expressions are a more powerful means of abstraction

def expressions:
  • Create a new function
  • Bind a name to it
User-Defined Functions

Named values are a simple means of abstraction

Named expressions are a more powerful means of abstraction

```python
>>> def expressions:
    • Create a new function
    • Bind a name to it
```

>>> def
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def expressions:
    # Create a new function
    # Bind a name to it
```

```python
>>> def square(x):
```

Monday, August 29, 2011
Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def expressions:
    • Create a new function
    • Bind a name to it

>>> def square(x):
    return mul(x, x)
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
>>> def square(x):
    return mul(x, x)
```

```python
def expressions:
    • Create a new function
    • Bind a name to it
```
User-Defined Functions

Named values are a simple means of abstraction

Named \textit{expressions} are a more powerful means of abstraction

\begin{tcolorbox}
def expressions:
  \begin{itemize}
    \item Create a new function
    \item Bind a name to it
  \end{itemize}
\end{tcolorbox}

\begin{verbatim}
>>> def square(x):
    return mul(x, x)
\end{verbatim}
User-Defined Functions

Named values are a simple means of abstraction

Named expressions are a more powerful means of abstraction

```python
>>> def expressions:
    • Create a new function
    • Bind a name to it

>>> def square(x):
    return mul(x, x)
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
def expressions:
    • Create a new function
    • Bind a name to it

square(x):
    return mul(x, x)
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
def expressions:
    • Create a new function
    • Bind a name to it

    mul:
    ...

    mul(a, b):

    square(x):
        return mul(x, x)
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
def expressions:
    • Create a new function
    • Bind a name to it

mul:

...  # Definition of the mul function

square:

square(x):
    return mul(x, x)
```
User-Defined Functions

Named values are a simple means of abstraction

Named *expressions* are a more powerful means of abstraction

```python
def expressions:
    # Create a new function
    # Bind a name to it

mul: ...

mul(a, b):

square:

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

(Demo)
Calling User-Defined Functions

```python
from operator import mul

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

```mermaid
graph U {
    mul[a] -- mul(a,b): [
        square[x]: [
            return mul(x, x)
        ]
    ]
}
```
from operator import mul

def square(x):
    return mul(x, x)
Calling User-Defined Functions

```python
from operator import mul

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

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

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

- Bind formal parameters
- Eval return expression
Calling User-Defined Functions

```python
from operator import mul

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

square(-2)
```

- Bind formal parameters
- Eval return expression

Environments & Values

Expressions
Calling User-Defined Functions

```
from operator import mul

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

square(-2)
```

- Bind formal parameters
- Eval return expression

Environments & Values
Expressions

```
from operator import mul
def square(x):
    return mul(x, x)
```

Monday, August 29, 2011
Calling User-Defined Functions

```python
from operator import mul

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

square(-2)
```

- Bind formal parameters
- Eval return expression

Environments & Values
Expressions
Calling User-Defined Functions

- Bind formal parameters
- Eval return expression

```python
from operator import mul
def square(x):
    return mul(x, x)

square(-2)
```

Monday, August 29, 2011
Calling User-Defined Functions

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

- Bind formal parameters
- Eval return expression

Environments & Values

Expressions

Monday, August 29, 2011
Calling User-Defined Functions

```
from operator import mul

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

square(-2)
```

- Bind formal parameters
- Eval return expression
Calling User-Defined Functions

```
from operator import mul

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

square(-2)
```

- Bind formal parameters
- Eval return expression

Environments & Values

Expressions

Monday, August 29, 2011
Calling User-Defined Functions

from operator import mul

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

square(-2)
Calling User-Defined Functions

```python
from operator import mul

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

def mul(a, b):
    return mul(a, b)

x = -2

print(square(x))  # Output: 4
```

Monday, August 29, 2011
Evaluating a Name in an Environment

\[ \text{mul}(x, x) \]

\[ \text{square}(x) : \]
\[ \text{return mul}(x, x) \]
Evaluating a Name in an Environment

```
mul: ...
square: 

mul(a, b):

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

x: -2

Points to an environment, which starts with a frame

return mul(x, x)
```
Evaluating a Name in an Environment

A name evaluates to the **value bound** to that name.
Evaluating a Name in an Environment

A name evaluates to the value bound to that name

...in the earliest frame of the current environment

A name evaluates to the value bound to that name

Points to an environment, which starts with a frame

return mul(x, x)
Evaluating a Name in an Environment

A name evaluates to the value bound to that name

...in the earliest frame of the current environment

...in which that name is found
A name evaluates to the value bound to that name...

...in the earliest frame of the current environment...

...in which that name is found.
Intrinsic Function Names Don’t Play a Role

\[
\begin{align*}
mul(a, b) : & \quad \text{return } mul(x, x) \\
\text{square}(x) : & \quad \text{return } mul(x, x) \\
x : & \quad -2
\end{align*}
\]
Intrinsic Function Names Don’t Play a Role

```
mul: ...

square:

return mul(x, x)
```

```
mul(a,b): ...

square(x):

return mul(x, x)
```

```
x: -2

square:

return mul(x, x)
```

Does not get inspected
Intrinsic Function Names Don’t Play a Role

**mul**: 

... 

**square**: 

**x**: -2 

**square**: 

**mul(a,b)**: 

**square(x)**: 

return mul(x, x) 

return mul(x, x)

Witness Protection Program

Does not get inspected
Intrinsic Function Names Don’t Play a Role

Witness Protection Program

Functions aren’t accessed by their intrinsic names!

(Demo)
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

Monday, August 29, 2011
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

```
sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

def sum_squares(5, 12):
    return add(square(5), square(12))
```

Monday, August 29, 2011
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

def sum_squares(x, y):
    return add(square(x), square(y))
```

```
sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

Monday, August 29, 2011
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

Monday, August 29, 2011
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```
Example: Function Application

sum_squares(5,12)

add(square(x), square(y))

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

def sum_squares(x, y):
    return add(square(x), square(y))
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

```
A

x: 5
y: 12

sum_squares

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

sum_squares(x, y):
return add(square(x), square(y))
```

```
square:
sum_squares:
add, mul, ...

sum_squares(5, 12)
A

add(square(x), square(y))
A

25

square(x)
square(y)

mul(x, x)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

Example:
```
sum_squares(5, 12)
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

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

def sum_squares(x, y):
    return add(square(x), square(y))
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))
```

Example:
```
sum_squares(5, 12)
```
**Example: Function Application**

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

```
Example: Function Application
square:
sum_squares:
add, mul, ...

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

sum_squares(x, y):
return add(square(x), square(y))

x: 5
y: 12
def square(x):
    return mul(x, x)
def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

```
Monday, August 29, 2011
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```
Example: Function Application

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

def sum_squares(x, y):
    return add(square(x), square(y))

def main(x, y):
    return sum_squares(x, y)

square(5): 25
square(12): 144
sum_squares(5, 12): 169
```
Example: Function Application

```
sum_squares(5, 12)
```

```
def sum_squares(x, y):
    return add(square(x), square(y))
```

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

```
x: 5
y: 12
```

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

```
square(y)
```

```
square(5):
    mul(x, x)
```

```
square(12):
    mul(x, x)
```

```
sum_squares(5, 12):
    add(square(5), square(12))
```

```
add(25, 144)
```

```
x: 5
mul(x, x)
```

```
x: 12
mul(x, x)
```

```
sum_squares:
    add, mul, ...
```

```
square:
```

Monday, August 29, 2011
Example: Function Application

```python
def square(x):
    return mul(x, x)
def sum_squares(x, y):
    return add(square(x), square(y))

sum_squares(5, 12)
```

```
Example: Function Application

- square(x):
  return mul(x, x)
- sum_squares(x, y):
  return add(square(x), square(y))

sum_squares(5, 12)
```