Lecture 2: Functions

Brian Hou
June 21, 2016
Announcements
Announcements

• Set up your computer and all accounts (Lab 0) by today
Announcements

• Set up your computer and all accounts (Lab 0) by today
  • Piazza, Instructional (cs61a-??), OK
Announcements

• Set up your computer and all accounts (Lab 0) by today
  • Piazza, Instructional (cs61a-??), OK
• Discussion sections begin today!
Announcements

- Set up your computer and all accounts (Lab 0) by today
  - Piazza, Instructional (cs61a-??), OK
- Discussion sections begin today!
- Office hours begin today!
Announcements

- Set up your computer and all accounts (Lab 0) by today
  - Piazza, Instructional (cs61a-??), OK
- Discussion sections begin today!
- Office hours begin today!
- Homework 0 is due tomorrow (Wednesday) at 11:59pm
Announcements

• Set up your computer and all accounts (Lab 0) by today
  • Piazza, Instructional (cs61a-??), OK
• Discussion sections begin today!
• Office hours begin today!
• Homework 0 is due tomorrow (Wednesday) at 11:59pm
• Quiz 1 will be on Thursday at the beginning of lecture
Expressions

Primitive expressions, names, and environments
Primitive expressions
Primitive expressions

- *Expressions* in programs evaluate to values
Primative expressions

- Expressions in programs evaluate to values
- Primitive expressions evaluate directly to values with minimal work needed
Primitive expressions

• *Expressions* in programs evaluate to values
• *Primitive expressions* evaluate directly to values with minimal work needed
  • *Numbers* (e.g. 42, 3.14, 0)
Primitive expressions

- *Expressions* in programs evaluate to values
- *Primitive expressions* evaluate directly to values with minimal work needed
  - *Numbers* (e.g. 42, 3.14, 0)
  - *Names* (e.g. pi, add)
Primitive expressions

- *Expressions* in programs evaluate to values
- *Primitive expressions* evaluate directly to values with minimal work needed
  - *Numbers* (e.g. 42, 3.14, 0)
  - *Names* (e.g. pi, add)
  - *Functions* (later today!)
Primitive expressions

- Expressions in programs evaluate to values
- Primitive expressions evaluate directly to values with minimal work needed
  - Numbers (e.g. 42, 3.14, 0)
  - Names (e.g. pi, add)
  - Functions (later today!)
- Some non-primitive expressions: 1 * 2, add(3, 4)
Names
Names

• Giving names to values makes programming easier!
Names

- Giving names to values makes programming easier!
- An *assignment statement* is one way to bind a name to a value (e.g. \( x = 1 \))
Names

• Giving names to values makes programming easier!
• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))
• Each name can only be bound to one value
Names

• Giving names to values makes programming easier!
• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))
• Each name can only be bound to one value
  • Environments keep track of names and their values
Names

• Giving names to values makes programming easier!

• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))

• Each name can only be bound to one value
  • Environments keep track of names and their values

Execution Rule for Assignment Statements:
Names

• Giving names to values makes programming easier!
• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))
• Each name can only be bound to one value
  • Environments keep track of names and their values

Execution Rule for Assignment Statements:
1. Evaluate all expressions to the right of = from left to right.
Names

• Giving names to values makes programming easier!
• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))
• Each name can only be bound to one value
  • Environments keep track of names and their values

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 environment frame.
Names

• Giving names to values makes programming easier!

• An assignment statement is one way to bind a name to a value (e.g. \( x = 1 \))

• Each name can only be bound to one value
  • *Environments* keep track of names and their values

**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 environment frame.
Environment diagrams

- Environment diagrams visualize the interpreter's progress
Environment diagrams

- Environment diagrams visualize the interpreter's progress

1. \( x = 1 \)
2. \( y = x \)

Global frame

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>1</td>
</tr>
</tbody>
</table>
Environment diagrams

- Environment diagrams visualize the interpreter's progress

1. \( x = 1 \)
2. \( y = x \)

Global frame:

\[ x \mid 1 \]
Environment diagrams

- Environment diagrams visualize the interpreter's progress

1. \( x = 1 \)
2. \( y = x \)

Global frame

Code (left)          Frames (right)
Environment diagrams

- Environment diagrams visualize the interpreter's progress

```
1 \textcolor{green!50!black}{x} = 1

2 \textcolor{red!50!black}{y} = x
```

Code (left) 

```
Global frame
\begin{array}{l}
\text{x} \mid 1
\end{array}
```

Frames (right)
Environment diagrams

- Environment diagrams visualize the interpreter's progress.

```
1  x = 1  
2  y = x 
```

Just executed

Next to execute

Code (left)  Frames (right)
Environment diagrams

- Environment diagrams visualize the interpreter's progress

Just executed

1 \[ x = 1 \]

2 \[ y = x \]

Global frame

x \[ \boxed{1} \]

Code (left)
Frames (right)

Statements and expressions
Environment diagrams

- Environment diagrams visualize the interpreter's progress

Statements and expressions

Code (left)          Frames (right)

Statements and expressions
Environment diagrams

- Environment diagrams visualize the interpreter's progress

```
Just executed

1 \( x = 1 \)

2 \( y = x \)

Next to execute

Global frame

\( x \mid 1 \)
```

**Code (left)**
- Statements and expressions

**Frames (right)**
- Each name is bound to a value
Environment diagrams

- Environment diagrams visualize the interpreter's progress

1. Just executed
   - x = 1

2. Next to execute
   - y = x

3. Assignment statement

   Code (left)
   Statements and expressions

   Frames (right)
   Each name is bound to a value
Environment diagrams

- Environment diagrams visualize the interpreter's progress

![Diagram showing code and frames]

**Code (left)**
Statements and expressions

**Frames (right)**
Each name is bound to a value
Environment diagrams

- Environment diagrams visualize the interpreter's progress

Code (left)
- Statements and expressions

Frames (right)
- Each name is bound to a value
- A name cannot be repeated in a frame
Environment diagrams visualize the interpreter's progress.

- Each name is bound to a value.
- A name cannot be repeated in a frame.
Functions

Call expressions, functions, and def statements
Call expressions
Call expressions

add ( 2 , 3 )
Call expressions

```
add ( 2 , 3 )
```

operator
Call expressions

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

operator \quad \quad \quad operands
Call expressions

- Call expressions use functions to compute a value

```
    add ( 2, 3 )
```

- operator
- operands
Call expressions

- Call expressions use functions to compute a value
- The operator and operands themselves are expressions
Call expressions

- Call expressions use functions to compute a value
- The operator and operands themselves are expressions
- To evaluate this call expression:
Call expressions

- Call expressions use functions to compute a value
- The operator and operands themselves are expressions
- To evaluate this call expression:
  1. *Evaluate* the operator to get a function value

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

operator _______ _______ operands
Call expressions

• Call expressions use functions to compute a value
• The operator and operands themselves are expressions
• To evaluate this call expression:
  1. Evaluate the operator to get a function value
  2. Evaluate the operands to get its values

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

operator \_\_\_\_\_

 operands \_\_\_\_\_


Call expressions

- *Call expressions* use functions to compute a value
- The operator and operands themselves are expressions
- To evaluate this call expression:
  1. *Evaluate* the operator to get a function value
  2. *Evaluate* the operands to get its values
  3. *Apply* the function to the values of the operands to get the final value

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

• Functions have inputs and outputs
Defining functions

• Functions have inputs and outputs

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

- Functions have inputs and outputs

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

- Functions have **inputs** and **outputs**

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

- Functions have inputs and outputs

Function **signature** indicates name and number of arguments

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

- Functions have **inputs** and **outputs**

  Function **signature** indicates name and number of arguments

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

  Function **body** defines computation performed when function is applied
Defining functions

- Functions have **inputs** and **outputs**

Function *signature* indicates name and number of arguments

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

Function *body* defines computation performed when function is applied

```python
def square(x):
    return x * x

y = square(-2)
```
Defining functions

- Functions have **inputs** and **outputs**

Function **signature** indicates name and number of arguments

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

Function **body** defines computation performed when function is applied

```
def square(x):
    return x * x
```

```
y = square(-2)
```

Execution Rule for **def** Statements:
Defining functions

- Functions have **inputs** and **outputs**

Function **signature** indicates name and number of arguments

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

Function **body** defines computation performed when function is applied

```
def square(x):
    return x * x
```

```
y = square(-2)
```

**Execution Rule for def Statements:**

1. Create a function with signature `<name>(<parameters>)`
Defining functions

- Functions have **inputs** and **outputs**

Function **signature** indicates name and number of arguments

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

Function **body** defines computation performed when function is applied

```
def square(x):
    return x * x
```

```
y = square(-2)
```

**Execution Rule for** `def` **Statements:**

1. Create a function with signature `<name>(<parameters>)`
2. Set the body of that function to be everything indented after the first line
Defining functions

- Functions have **inputs** and **outputs**

**Function signature** indicates name and number of arguments

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

**Function body** defines computation performed when function is applied

```
def square(x):
    return x * x
```

```
y = square(-2)
```

**Execution Rule for def Statements:**

1. Create a function with signature `<name>(<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
Calling user-defined functions

1 def square(x):
2     return x * x
3 y = square(-2)
Calling user–defined functions

Rules for calling user–defined functions (version 1):
1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1 def square(x):
2     return x * x
3 y = square(-2)
```
Calling user–defined functions

Rules for calling user–defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1 def square(x):
2   return x * x
3 y = square(-2)
```
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
def square(x):
    return x * x
3 y = square(-2)
```
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
def square(x):
    return x * x

y = square(-2)
```
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1 def square(x):
2     return x * x
3 y = square(-2)
```
Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
def square(x):
    return x * x
```

Frames

Global frame

```
square
```

f1: square

```
x
```
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```
    def square(x):
        return x * x
    y = square(-2)
```
Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1 def square(x):
2     return x * x
3 y = square(-2)
```
Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1  def square(x):
2     return x * x
3  y = square(-2)
```
Calling user-defined functions

Rules for calling user-defined functions (version 1):

1. Create a new environment frame
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

```python
1  def square(x):
2     return x * x
3  y = square(-2)
```

Return value (not a binding!)
Break!
Environments
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
- An environment is a sequence of frames
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
Looking up names in environments

• Every expression is evaluated in the context of an environment
• An environment is a sequence of frames
• So far, there have been two possible environments:
  • The global frame
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1):
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1):

1. Look it up in the local frame
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1):
1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
Looking up names in environments

- Every expression is evaluated in the context of an environment
- An environment is a sequence of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1):

1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, NameError
Looking up names in environments

Rules for looking up names in user-defined functions (version 1):
1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, NameError
Multiple environments
Multiple environments

```python
>>> def square(x):
    ...
    return x * x

>>> y = square(square(-2))
```
Multiple environments  

```python
>>> def square(x):
    ...
    return x * x

>>> y = square(square(-2))
```
Multiple environments

```python
>>> def square(x):
...     return x * x

>>> y = square(square(-2))
```

(Demo)
Multiple environments

def square(x):
    return x * x

>>> y = square(square(-2))
Multiple environments

```python
>>> def square(x):
...    return x * x

>>> y = square(square(-2))
```

<table>
<thead>
<tr>
<th>Global frame</th>
<th>func square(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>square</td>
<td>y 16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1: square</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>Return value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f2: square</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>Return value</td>
</tr>
</tbody>
</table>
Multiple environments

```python
>>> def square(x):
...     return x * x

>>> y = square(square(-2))
```

```
1
```
Multiple environments  

>>> def square(x):
...     return x * x

>>> y = square(square(-2))
Multiple environments (demo)

```python
>>> def square(x):
...     return x * x

>>> y = square(square(-2))
```

![Diagram showing variable scope and function calls](image)
Multiple environments

```python
>>> def square(x):
...     return x * x

>>> y = square(square(-2))
```

(demo)
None and Print
None means that nothing is returned
None means that nothing is returned

• The special value None represents nothing in Python
None means 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 means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- *Note:* None is *not displayed* by the interpreter as the value of an expression
None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- *Note:* None is *not displayed* by the interpreter as the value of an expression

```python
global def does_not_square(x):
    ... x * x
```
None means that nothing is returned

- The special value `None` represents nothing in Python.
- A function that does not explicitly return a value will return `None`.
- *Note:* `None` is *not displayed* by the interpreter as the value of an expression.

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

No `return`
None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- *Note:* None is *not displayed* by the interpreter as the value of an expression

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

>>> does_not_square(-2)
```

No *return*
None means that nothing is returned

- The special value None represents nothing in Python.
- A function that does not explicitly return a value will return None.
- *Note:* None is *not displayed* by the interpreter as the value of an expression.

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

>>> does_not_square(-2)
None
```

No return

None value is not displayed
None means that nothing is returned

- The special value None represents nothing in Python.
- A function that does not explicitly return a value will return None.
- Note: None is *not displayed* by the interpreter as the value of an expression.

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

>>> does_not_square(-2)

>>> not_four = does_not_square(-2)
```

No return

None value is not displayed
None means that nothing is returned

- The special value `None` represents nothing in Python
- A function that does not explicitly return a value will return `None`
- *Note:* `None` is *not displayed* by the interpreter as the value of an expression

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

>>> does_not_square(-2)

>>> not_four = does_not_square(-2)

No return

The name `not_four` is now bound to the value `None`

None value is not displayed
None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- **Note:** None is *not displayed* by the interpreter as the value of an expression

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

>>> does_not_square(-2)
None

>>> not_four = does_not_square(-2)

>>> not_four + 4

No return

The name `not_four` is now bound to the value `None`

None value is not displayed
None means that nothing is returned

- The special value None represents nothing in Python
- A function that does not explicitly return a value will return None
- *Note:* None is *not displayed* by the interpreter as the value of an expression

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

>>> does_not_square(-2)

>>> not_four = does_not_square(-2)

>>> not_four + 4

TypeError: unsupported operand type(s) for +:
'NoneType' and 'int'
```

No return

The name `not_four` is now bound to the value `None`
Pure and non-pure functions
Pure and non–pure functions

*Pure functions*
just return values
Pure and non-pure functions

*Pure functions* just return values

2, 3 \(\text{max}\) 3
Pure and non-pure functions

*Pure functions* just return values

2, 3 → \text{max} → 3

2 Arguments
Pure and non-pure functions

*Pure functions* just return values

- 2, 3
- max
- 3
- 2 Arguments
- Return value
Pure and non–pure functions

**Pure functions**
just return values

2, 3 → max

2 Arguments

Return value

3

**Non–Pure functions**
have side effects
Pure and non-pure functions

**Pure functions**
just return values

-2, 3 \(\rightarrow\) max
2 Arguments \(\rightarrow\) Return value

**Non-Pure functions**
have side effects

-2 \(\rightarrow\) print
Pure and non-pure functions

**Pure functions**
just return values

- 2, 3

**Non-Pure functions**
have side effects

- -2

Python displays the output “-2”
Pure and non–pure functions

Pure functions just return values

Non–Pure functions have side effects

2 Arguments

2, 3

max

3

-2

print

None

Returns None!

Python displays the output “–2”
Pure and non-pure functions

Pure functions just return values

2, 3 \rightarrow \text{max}

2 Arguments

Return value

3

Non-Pure functions have side effects

-2 \rightarrow \text{print}

Python displays the output “-2”

Returns None!

None

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

>>> print(print(1), print(2))
Nested expressions with print

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

Nested expressions with print

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

`print` function calls are nested, with each subsequent call wrapping the output of the previous call.
Nested expressions with print

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

1
  print
    ```
    >>> print(print(1), print(2))
    ```
Nested expressions with print

1

| print |

display “1”

```python
>>> print(print(1), print(2))
1
```
Nested expressions with print

```python
>>> print(print(1), print(2))
1
None
display "1"
```
Nested expressions with print

```python
>>> print(print(1), print(2))
1
None
display "1"
```
Nested expressions with print

```python
>>> print(print(1), print(2))
1
None
```
Nested expressions with print

1

```
print
```

`display "1"

2

```
print
```

`None`

```python
>>> print(print(1), print(2))
1
None
```
Nested expressions with print

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

```python
1
print
display "1"

2
print
display "2"
```
Nested expressions with print

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

print
display "1"

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

print
display "2"

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

print
display "1"
```
Nested expressions with print

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

1

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

1

None

None

display “2”
Nested expressions with print

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

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

Diagram:

1. `print`
   - `display "1"`
   - `None`

2. `print`
   - `display "2"`
   - `None`
Nested expressions with print

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

>>> print(print(1), print(2))
1
2
None None
display "None None"
```
Nested expressions with print

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

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

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

display “1”

display “2”

display “None None None”
More Functions
More Functions

- The operands of a call expression can be any expression
More Functions

- The operands of a call expression can be any expression
- What about the expression `square`?
More Functions

• The operands of a call expression can be any expression
• What about the expression `square`?

```python
>>> four = describe(square, -2)
```
More Functions

- The operands of a call expression can be any expression
- What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
```
More Functions

• The operands of a call expression can be any expression
• What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
```

More Functions

- The operands of a call expression can be any expression
- What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
```
More Functions

- The operands of a call expression can be any expression
- What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
```
More Functions

• The operands of a call expression can be any expression
• What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
```
More Functions

- The operands of a call expression can be any expression
- What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4

>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
>>> sixteen
```
More Functions

• The operands of a call expression can be any expression
• What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
>>> sixteen
16
```
More Functions (demo)

• The operands of a call expression can be any expression
• What about the expression `square`?

```python
>>> four = describe(square, -2)
Calling function with argument -2
Result was 4
>>> four
4
>>> sixteen = describe(square, four)
Calling function with argument 4
Result was 16
>>> sixteen
16
```