Function Examples
Announcements
Hog Contest Rules

[cs61a.org/proj/hog_contest]
Hog Contest Rules

- Up to two people submit one entry;
  Max of one entry per person

[cs61a.org/proj/hog_contest]
Hog Contest Rules

- Up to two people submit one entry;
  Max of one entry per person
- Slight rule changes

[cs61a.org/proj/hog_contest](cs61a.org/proj/hog_contest)
Hog Contest Rules

- Up to two people submit one entry;
  Max of one entry per person
- Slight rule changes
- Your score is the number of entries against which you win more than 50.00001% of the time

[cs61a.org/proj/hog_contest]
Hog Contest Rules

• Up to two people submit one entry;
  Max of one entry per person

• Slight rule changes

• Your score is the number of entries
  against which you win more than
  50.00001% of the time

• Strategies are time-limited

[Link to Hog Contest website] cs61a.org/proj/hog_contest
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person

• Slight rule changes

• Your score is the number of entries against which you win more than 50.00001% of the time

• Strategies are time-limited

• All strategies must be deterministic, pure functions of the players' scores

cs61a.org/proj/hog_contest
Hog Contest Rules

- Up to two people submit one entry; Max of one entry per person
- Slight rule changes
- Your score is the number of entries against which you win more than 50.00001% of the time
- Strategies are time-limited
- All strategies must be deterministic, pure functions of the players' scores
- All winning entries will receive extra credit
Hog Contest Rules

- Up to two people submit one entry; Max of one entry per person
- Slight rule changes
- Your score is the number of entries against which you win more than 50.00001% of the time
- Strategies are time-limited
- All strategies must be deterministic, pure functions of the players' scores
- All winning entries will receive extra credit
- The real prize: honor and glory

cs61a.org/proj/hog_contest
Hog Contest Rules

• Up to two people submit one entry;  
  Max of one entry per person

• Slight rule changes

• Your score is the number of entries  
  against which you win more than 50.00001% of the time

• Strategies are time-limited

• All strategies must be deterministic,  
  pure functions of the players' scores

• All winning entries will receive  
  extra credit

• The real prize: honor and glory

• See website for detailed rules

[cs61a.org/proj/hog_contest]
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Slight rule changes
• Your score is the number of entries against which you win more than 50.000001% of the time
• Strategies are time-limited
• All strategies must be deterministic, pure functions of the players' scores
• All winning entries will receive extra credit
• The real prize: honor and glory
• See website for detailed rules

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

cs61a.org/proj/hog_contest
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Slight rule changes
• Your score is the number of entries against which you win more than 50.00001% of the time
• Strategies are time-limited
• All strategies must be deterministic, pure functions of the players' scores
• All winning entries will receive extra credit
• The real prize: honor and glory
• See website for detailed rules

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

cs61a.org/proj/hog_contest
Hog Contest Rules

- Up to two people submit one entry; Max of one entry per person
- Slight rule changes
- Your score is the number of entries against which you win more than 50.00001% of the time
- Strategies are time-limited
- All strategies must be deterministic, pure functions of the players' scores
- All winning entries will receive extra credit
- The real prize: honor and glory
- See website for detailed rules

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

cs61a.org/proj/hog_contest
Hog Contest Rules

• Up to two people submit one entry; Max of one entry per person
• Slight rule changes
• Your score is the number of entries against which you win more than 50.00001% of the time
• Strategies are time-limited
• All strategies must be deterministic, pure functions of the players' scores
• All winning entries will receive extra credit
• The real prize: honor and glory
• See website for detailed rules

Fall 2011 Winners
Kaylee Mann
Yan Duan & Ziming Li
Brian Prike & Zhenghao Qian
Parker Schuh & Robert Chatham

Fall 2012 Winners
Chenyang Yuan
Joseph Hui

Fall 2013 Winners
Paul Bramsen
Sam Kumar & Kangsik Lee
Kevin Chen

Fall 2014 Winners
Alan Tong & Elaine Zhao
Zhenyang Zhang
Adam Robert Villaflor & Joany Gao
Zhen Qin & Dian Chen
Zizheng Tai & Yihe Li

cs61a.org/proj/hog_contest
Hog Contest Winners

**Spring 2015 Winners**
Sinho Chewi & Alexander Nguyen Tran
Zhaoxi Li
Stella Tao and Yao Ge

**Fall 2015 Winners**
Micah Carroll & Vasilis Oikonomou
Matthew Wu
Anthony Yeung and Alexander Dai

**Spring 2016 Winners**
Michael McDonald and Tianrui Chen
Andrei Kassiantchouk
Benjamin Krieges

**Spring 2017 Winners**
Cindy Jin and Sunjoon Lee
Anny Patino and Christian Vasquez
Asana Choudhury and Jenna Wen
Michelle Lee and Nicholas Chew

**Fall 2017 Winners**
Alex Yu and Tanmay Khattar
James Li
Justin Yokota

**Spring 2018 Winners**
Abstraction
Functional Abstractions
Functional Abstractions

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

def sum_squares(x, y):
    return square(x) + square(y)
Functional Abstractions

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

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

What does sum_squares need to know about square?
Functional Abstractions

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

What does sum_squares need to know about square?

- Square takes one argument.
Functional Abstractions

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

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  
  Yes
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

* Square takes one argument.  
  Yes

* Square has the intrinsic name `square`. 

Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument. **Yes**
- Square has the intrinsic name `square`. **No**
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument. Yes
- Square has the intrinsic name `square`. No
- Square computes the square of a number.
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  - Yes

- Square has the intrinsic name `square`.  
  - No

- Square computes the square of a number.  
  - Yes
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  - Yes

- Square has the intrinsic name `square`.  
  - No

- Square computes the square of a number.  
  - Yes

- Square computes the square by calling `mul`.  
  -
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

* Square takes one argument.  
  Yes

* Square has the intrinsic name square.  
  No

* Square computes the square of a number.  
  Yes

* Square computes the square by calling `mul`.  
  No
**Functional Abstractions**

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  Yes

- Square has the intrinsic name `square`.  
  No

- Square computes the square of a number.  
  Yes

- Square computes the square by calling `mul`.  
  No

```python
def square(x):
    return pow(x, 2)
```
Functional Abstractions

\[
def \text{square}(x):
    \text{return} \ \text{mul}(x, x)
\]

\[
def \text{sum\_squares}(x, y):
    \text{return} \ \text{square}(x) + \text{square}(y)
\]

What does \text{sum\_squares} need to know about \text{square}?

- Square takes one argument. \hspace{1cm} Yes
- Square has the intrinsic name square. \hspace{1cm} No
- Square computes the square of a number. \hspace{1cm} Yes
- Square computes the square by calling mul. \hspace{1cm} No

\[
def \text{square}(x):
    \text{return} \ \text{pow}(x, 2)
\]

\[
def \text{square}(x):
    \text{return} \ \text{mul}(x, x-1) + x
\]
Functional Abstractions

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

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

What does `sum_squares` need to know about `square`?

- Square takes one argument.  
  - Yes
- Square has the intrinsic name `square`.  
  - No
- Square computes the square of a number.  
  - Yes
- Square computes the square by calling `mul`.  
  - No

```python
def square(x):
    return pow(x, 2)
```

```python
def square(x):
    return mul(x, x-1) + x
```

If the name “square” were bound to a built-in function, `sum_squares` would still work identically.
Choosing Names
Choosing Names

Names typically don’t matter for correctness

but

eye matter a lot for composition
Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.
Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.
Choosing Names

Names typically don’t matter for correctness

but

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (`print`), their behavior (`triple`), or the value returned (`abs`).
Choosing Names

Names typically don’t matter for correctness

**but**

they matter a lot for composition

| From: | To: |

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).
Choosing Names

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
</tbody>
</table>

Names typically don’t matter for correctness

*but*

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (*print*), their behavior (*triple*), or the value returned (*abs*).
Choosing Names

Names typically don’t matter for correctness

**but**

they matter a lot for composition

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
</tbody>
</table>

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).
Choosing Names

Names typically don’t matter for correctness

*but*

they matter a lot for composition

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (*print*), their behavior (*triple*), or the value returned (*abs*).
Choosing Names

Names typically don’t matter for correctness

**but**

they matter a lot for composition

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>helper</td>
<td>take_turn</td>
</tr>
<tr>
<td>my_int</td>
<td>num_rolls</td>
</tr>
</tbody>
</table>

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (**print**), their behavior (**triple**), or the value returned (**abs**).
Choosing Names

Names typically don’t matter for correctness

**but**

they matter a lot for composition

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_false</td>
<td>rolled_a_one</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>helper</td>
<td>take_turn</td>
</tr>
<tr>
<td>my_int</td>
<td>num_rolls</td>
</tr>
<tr>
<td>l, I, 0</td>
<td>k, i, m</td>
</tr>
</tbody>
</table>

Names should convey the meaning or purpose of the values to which they are bound.

The type of value bound to the name is best documented in a function's docstring.

Function names typically convey their effect (*print*), their behavior (*triple*), or the value returned (*abs*).
Which Values Deserve a Name

Reasons to add a new name
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))

hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

\[
\text{if } \sqrt{a^2 + b^2} > 1: \\
\quad x = x + \sqrt{a^2 + b^2}
\]

\[
\text{hypotenuse} = \sqrt{a^2 + b^2} \\
\text{if } \text{hypotenuse} > 1: \\
\quad x = x + \text{hypotenuse}
\]

Meaningful parts of complex expressions:

\[
x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```python
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

\[
\text{if } \sqrt{\text{square}(a) + \text{square}(b)} > 1: \\
\quad x = x + \sqrt{\text{square}(a) + \text{square}(b)}
\]

\[
\text{hypotenuse} = \sqrt{\text{square}(a) + \text{square}(b)} \\
\text{if } \text{hypotenuse} > 1: \\
\quad x = x + \text{hypotenuse}
\]

Meaningful parts of complex expressions:

\[
x1 = (-b + \sqrt{\text{square}(b) - 4 \times a \times c}) / (2 \times a)
\]

\[
\text{discriminant} = \text{square}(b) - 4 \times a \times c \\
x1 = (-b + \sqrt{\text{discriminant}}) / (2 \times a)
\]
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```python
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

- Names can be long if they help document your code:

```python
average_age = average(age, students)
```

is preferable to

```python
# Compute average age of students
aa = avg(a, st)
```
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))

hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse

Meaningful parts of complex expressions:

x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)

discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)

More Naming Tips

• Names can be long if they help document your code:

average_age = average(age, students)

is preferable to

# Compute average age of students
aa = avg(a, st)

• Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

n, k, i — Usually integers
x, y, z — Usually real numbers
f, g, h — Usually functions
Which Values Deserve a Name

Reasons to add a new name

Repeated compound expressions:

```python
if sqrt(square(a) + square(b)) > 1:
    x = x + sqrt(square(a) + square(b))
```

```python
hypotenuse = sqrt(square(a) + square(b))
if hypotenuse > 1:
    x = x + hypotenuse
```

Meaningful parts of complex expressions:

```python
x1 = (-b + sqrt(square(b) - 4 * a * c)) / (2 * a)
```

```python
discriminant = square(b) - 4 * a * c
x1 = (-b + sqrt(discriminant)) / (2 * a)
```

More Naming Tips

• Names can be long if they help document your code:

```python
average_age = average(age, students)
```

is preferable to

```python
# Compute average age of students
aa = avg(a, st)
```

• Names can be short if they represent generic quantities: counts, arbitrary functions, arguments to mathematical operations, etc.

```
n, k, i - Usually integers
x, y, z - Usually real numbers
f, g, h - Usually functions
```
Testing
Test-Driven Development
Test-Driven Development

Write the test of a function before you write the function.
Test-Driven Development

Write the test of a function before you write the function.

* A test will clarify the domain, range, & behavior of a function.
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

Tests can help identify tricky edge cases.
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.
Test-Driven Development

Write the test of a function before you write the function.

* A test will clarify the domain, range, & behavior of a function.

* Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

* You can't depend upon code that hasn't been tested.
Test-Driven Development

Write the test of a function before you write the function.

- A test will clarify the domain, range, & behavior of a function.
- Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

- You can't depend upon code that hasn't been tested.
- Run your old tests again after you make new changes.
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

You can't depend upon code that hasn't been tested.

Run your old tests again after you make new changes.

Bonus idea: Run your code interactively.
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.

Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

You can't depend upon code that hasn't been tested.

Run your old tests again after you make new changes.

Bonus idea: Run your code interactively.

Don't be afraid to experiment with a function after you write it.
Test-Driven Development

Write the test of a function before you write the function.

A test will clarify the domain, range, & behavior of a function.
Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.
You can't depend upon code that hasn't been tested.
Run your old tests again after you make new changes.

Bonus idea: Run your code interactively.
Don't be afraid to experiment with a function after you write it.
Interactive sessions can become doctests. Just copy and paste.
Test-Driven Development

Write the test of a function before you write the function.

* A test will clarify the domain, range, & behavior of a function.

* Tests can help identify tricky edge cases.

Develop incrementally and test each piece before moving on.

* You can't depend upon code that hasn't been tested.

* Run your old tests again after you make new changes.

Bonus idea: Run your code interactively.

* Don't be afraid to experiment with a function after you write it.

* Interactive sessions can become doctests. Just copy and paste.
Currying
Function Currying
Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```
Function Currying

```python
def make_adder(n):
    return lambda k: n + k

>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```
Function Currying

``` python
def make_adder(n):
    return lambda k: n + k

>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions.
Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```

```python
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

There's a general relationship between these functions (Demo)
Function Currying

```python
def make_adder(n):
    return lambda k: n + k
```

```python
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

Curry: Transform a multi-argument function into a single-argument, higher-order function
Decorators
Function Decorators

(Demo)
Function Decorators

(Demo)

@trace1
def triple(x):
    return 3 * x
Function Decorators

(Demo)

```python
@trace1
def triple(x):
    return 3 * x
```
Function Decorators

(Demo)

@trace1
def triple(x):
    return 3 * x
Function Decorators

(Demo)

```
@trace1
def triple(x):
    return 3 * x
```

is identical to
Function Decorators

(Demo)

@trace1
def triple(x):
    return 3 * x

is identical to

def triple(x):
    return 3 * x

triple = trace1(triple)
Function Decorators

(Demo)

@trace1
def triple(x):
    return 3 * x

is identical to

def triple(x):
    return 3 * x
triple = trace1(triple)

Why not just use this?
What Would Python Display?
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul
def square(x):
    return mul(x, x)
```
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>square(16)</code></td>
<td>Evaluates to 256</td>
<td></td>
</tr>
</tbody>
</table>

16
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

This expression | Evaluates to | Interactive Output
--- | --- | ---
5 | 5 | 5

print(5)
```
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
</tbody>
</table>
**What Would Python Display?**

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><code>print(5)</code></td>
<td>None</td>
<td>5</td>
</tr>
</tbody>
</table>

What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

This expression | Evaluates to | Interactive Output
-----------------|--------------|-------------------
5                | 5            | 5                 
print(5)         | None         | 5                 
print(print(5))  | None         | 5                 
```
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

print(print(5))  # Evaluates to None. Interactive output: 5

print(5)          # None. Interactive output: 5

print(print(5))  # None. Interactive output: 5 None

This expression | Evaluates to | Interactive Output
-----------------|--------------|---------------------
5                | 5            | 5
print(5)         | None         | 5
print(print(5))  | None         | 5 None
```
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

def delay(arg):
    print('delayed')
    return

def g():
    return arg
    return g
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
</tbody>
</table>

This expression Evaluates to Interactive Output

```
print(print(5))
```

None

```
def delay(arg):
print('delayed')
def g():
    return arg
    return g
```

`None`
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

def delay(arg):
    print('delayed')
    return arg

def g():
    return delay(delay)()(6)
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>delay(delay)()</td>
<td>None</td>
<td>5 None</td>
</tr>
</tbody>
</table>

This expression Evaluates to Interactive Output
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
def delay(arg):
    print('delayed')
    return

def g():
    return arg
return g
```

Names in nested def statements can refer to their enclosing scope.

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)()(6)()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

def print_expressions():
    delay(delay)()(6)()

Names in nested def statements can refer to their enclosing scope
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td><code>delay(delay)()()</code>(6)()</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)()(6)()</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
def delay(arg):
    print('delayed')

def g():
    return arg

return g
```

A function that takes any argument and returns a function that returns that arg

```python
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

This expression | Evaluates to | Interactive Output
--- | --- | ---
5 | 5 | 5
print(5) | None | 5
print(print(5)) | None | 5
None | 5 | None

delay(delay)()(6)() | None | None

Names in nested def statements can refer to their enclosing scope
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul
def square(x):
    return mul(x, x)
def delay(arg):
    print('delayed')
    return
def g():
    return arg
return g
```

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)()(6)()</td>
<td>None</td>
<td>5 None</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

Names in nested def statements can refer to their enclosing scope

def delay(delay):
    print('delayed')
    def g():
        return delay(delay)
    return g()

This expression | Evaluates to | Interactive Output
---|---|---
5 | 5 | 5
print(5) | None | 5
print(print(5)) | None | 5
(delay(delay)()(6))() | None | 5
```
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

def g():
    return arg

Names in nested def statements can refer to their enclosing scope
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>delay(delay)()</td>
<td>delayed</td>
<td></td>
</tr>
</tbody>
</table>
The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
    def g():
        return arg
    return g
```

Names in nested def statements can refer to their enclosing scope

```
What Would Python Display?
```

This expression | Evaluates to | Interactive Output
---|---|---
5 | 5 | 5
print(5) | None | 5
print(print(5)) | None | 5 None
delay(delay)()(6)() | delayed delay | delayed
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
from operator import add, mul

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

def delay(arg):
    print('delayed')
    return g()

def g():
    return arg

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

A function that takes any argument and returns a function that returns that arg

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5 None</td>
</tr>
<tr>
<td>delay(delay)()</td>
<td>delayed</td>
<td>delayed</td>
</tr>
<tr>
<td>(6)()</td>
<td>delayed</td>
<td>6</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

def g():
    return arg

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>delay(delay)()</td>
<td>6</td>
<td>delayed delayed 6</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```python
def delay(arg):
    print('delayed')

def g():
    return arg
return g
```

A function that takes any argument and returns a function that returns that arg

```
from operator import add, mul

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

A function that takes any argument and returns a function that returns that arg

```
def delay(arg):
    print('delayed')
def g():
    return arg
return g
```

Names in nested def statements can refer to their enclosing scope

```
print(print(5))
print(print(print(5)))
print(print(delay(delay)))(6)()
print(delay(print)()(4))
```

This expression | Evaluates to | Interactive Output
---|---|---
5 | 5 | 5
print(5) | None | 5 None
print(print(5)) | None | 5 None
delay(delay)()() | 6 | delayed delayed 6
print(delay(print)())(4)) | | |
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

def g():
    return
    return

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

A function that takes any argument and returns a function that returns that arg

Names in nested def statements can refer to their enclosing scope

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(delay(print)())()()</td>
<td>delayed</td>
<td>delayed</td>
</tr>
<tr>
<td>print(delay(print)())()()</td>
<td>delayed</td>
<td>delayed</td>
</tr>
<tr>
<td>delay(delay)(6)()</td>
<td>delayed</td>
<td>delayed</td>
</tr>
<tr>
<td>delay(delay)(6)()</td>
<td>delayed</td>
<td>delayed</td>
</tr>
</tbody>
</table>

This expression Evaluates to Interactive Output

None

None

None

5

None

5

None

5

5

5
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

def delay(arg):
    print('delayed')
    return arg

def g():
    return delay(delay)()(6)()

print(delay(print)()(4))
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>(delay(delay))()()()()</td>
<td>6</td>
<td>delayed 6</td>
</tr>
<tr>
<td>print(delay(print)()())</td>
<td>4</td>
<td>delayed 4</td>
</tr>
</tbody>
</table>
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

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

A function that takes any argument and returns a function that returns that arg

def delay(arg):
    print('delayed')
    def g():
        return arg
    return g

Names in nested def statements can refer to their enclosing scope

def g():
    return arg

This expression                              Evaluates to  Interactive Output
5                                              5  5
print(5)                                        None   5
print(print(5))                                 None   5
None                                            5
None                                            None
print(delay(print)())                           6  delayed delayed
6                                              6
print(delay(print)()()4))                      None
4                                              None
What Would Python Display?

The print function returns None. It also displays its arguments (separated by spaces) when it is called.

```
from operator import add, mul

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

def delay(arg):
    print('delayed')
    return arg

def g():
    return delay(delay)()(6)()

print(delay(print)()(4))
```

<table>
<thead>
<tr>
<th>This expression</th>
<th>Evaluates to</th>
<th>Interactive Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>print(5)</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>print(print(5))</td>
<td>None</td>
<td>5</td>
</tr>
<tr>
<td>delay(delay)(6)()()</td>
<td>6</td>
<td>delayed delayed 6</td>
</tr>
<tr>
<td>print(delay(print)(4))</td>
<td>None</td>
<td>delayed delayed 4</td>
</tr>
</tbody>
</table>
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```

```
<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>horse</td>
</tr>
<tr>
<td>mask</td>
</tr>
</tbody>
</table>

- `func horse(mask)` [parent=Global]
- `func λ(horse)` [parent=Global]

Return Value
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return mask(horse)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```python
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```

```
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
```
def horse(mask):
    horse = mask
    return horse

def mask(horse):
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)
mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
def mask(horse):
    return horse
return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask

def mask(horse):
    return horse

return horse(mask)

mask = lambda horse: horse(2)

horse(mask)
def horse(mask):
    horse = mask
    return horse

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)
def horse(mask):
    horse = mask
    def mask(horse):
        return horse
    return horse(mask)

mask = lambda horse: horse(2)
horse(mask)