Hog Contest Rules

- Two people submit one entry; Max of one entry per person
Hog Contest Rules

- Two people submit one entry; Max of one entry per person
- The score for an entry is the sum of win rates against every other entry.
Hog Contest Rules

- Two people submit one entry; Max of one entry per person
- The score for an entry is the sum of win rates against every other entry.
- All strategies must be deterministic, pure functions of the current player scores! *Non-deterministic strategies will be disqualified.*
Hog Contest Rules

- Two people submit one entry; Max of one entry per person
- The score for an entry is the sum of win rates against every other entry.
- All strategies must be deterministic, pure functions of the current player scores! Non-deterministic strategies will be disqualified.
- To enter: submit proj1contest with a file hog.py that defines a final_strategy function by Monday 9/24 @ 11:59pm
Hog Contest Rules

• Two people submit one entry; Max of one entry per person

• The score for an entry is the sum of win rates against every other entry.

• All strategies must be deterministic, pure functions of the current player scores! Non-deterministic strategies will be disqualified.

• To enter: submit proj1contest with a file hog.py that defines a final_strategy function by Monday 9/24 @ 11:59pm

• All winning entries will receive 2 points of extra credit
Hog Contest Rules

• Two people submit one entry; Max of one entry per person

• The score for an entry is the sum of win rates against every other entry.

• All strategies must be deterministic, pure functions of the current player scores! Non-deterministic strategies will be disqualified.

• To enter: submit proj1contest with a file hog.py that defines a final_strategy function by Monday 9/24 @ 11:59pm

• All winning entries will receive 2 points of extra credit

• The real prize: honor and glory
Hog Contest Rules

- Two people submit one entry; Max of one entry per person
- The score for an entry is the sum of win rates against every other entry.
- All strategies must be deterministic, pure functions of the current player scores! *Non-deterministic strategies will be disqualified.*
- To enter: submit `proj1contest` with a file `hog.py` that defines a `final_strategy` function by **Monday 9/24 @ 11:59pm**
- All winning entries will receive 2 points of extra credit
- The real prize: honor and glory

**Fall 2011 Winners**
Keegan Mann,
Yan Duan & Ziming Li,
Brian Prike & Zhenghao Qian,
Parker Schuh & Robert Chatham
Choosing Names
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility
Choosing Names

Names typically don’t matter for correctness

but

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
</tbody>
</table>
Choosing Names

Names typically don’t matter for correctness

but

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
</tbody>
</table>
Choosing Names

Names typically don’t matter for correctness

**but**

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>

```python
>>> from operator import mul
>>> def square(let):
```
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>

```python
>>> from operator import mul
>>> def square(let):
    return mul(let, let)
```
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>

```python
>>> from operator import mul
>>> def square(let):
    return mul(let, let)
```
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>

>>> from operator import mul
>>> def square(let):
    return mul(let, let)
Choosing Names

Names typically *don’t* matter for correctness

*but*

they matter tremendously for legibility

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>turn_is_over</td>
</tr>
<tr>
<td>d</td>
<td>dice</td>
</tr>
<tr>
<td>play_helper</td>
<td>take_turn</td>
</tr>
</tbody>
</table>

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

*Not stylish*
Which Values Deserve a Name
Which Values Deserve a Name

Repeated compound expressions:
Which Values Deserve a Name

Repeated compound expressions:

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

Repeated compound expressions:

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

h = sqrt(square(a) + square(b))
if h > 1:
x = x + h
```
Repeated compound expressions:

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

Meaningful parts of complex expressions:

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

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

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

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

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

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

Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

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

```
d = sqrt(square(b) - 4 * a * c)
x = (-b + d) / (2 * a)
```

However, not every value needs a name (demo)
Repeated compound expressions:

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

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

Meaningful parts of complex expressions:

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

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

However, not every value needs a name (demo)
Test-Driven Development
Test-Driven Development

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

A test will clarify the (one) job of the function
Test-Driven Development

Write the test of a function before you write the function

A test will clarify the (one) job of the function

Your 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 (one) job of the function

Your 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 (one) job of the function

Your 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 (one) job of the function

Your 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
Function Decorators

(demo)
Function Decorators

(demo)

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

Function Decorators

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

(demo)
Function Decorators

Function decorator

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

Decorated function

(demo)
Function Decorators

(demo)

Function decorator

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

Decorated function

is identical to
Function Decorators

(demo)

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

triple = trace1(triple)
```

is identical to

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

(demo)

Function decorator

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

Decorated function

is identical to

Why not just use this?

def triple(x):
  return 3 * x
triple = trace1(triple)
Functional Abstractions
def square(x):
    return mul(x, x)
Functional Abstractions

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`?
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
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

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

```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
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`.  
  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`. No
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

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

- Square takes one argument.
- Square has the intrinsic name square.
- Square computes the square of a number.
- Square computes the square by calling mul.

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

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

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

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
Data
Student seating preferences at MIT

http://www.skyrill.com/seatinghabits/
Data

Student seating preferences at MIT

Front of the classroom

http://www.skyrill.com/seatinghabits/
Objects
Objects

- Representations of information
Objects

- Representations of information
- Data and behavior, bundled together to create...
Objects

• Representations of information

• Data and behavior, bundled together to create...

Abstractions
Objects

- Representations of information
- Data and behavior, bundled together to create...

*Abstractions*

- Objects represent properties, interactions, & processes
Objects

• Representations of information

• Data and behavior, bundled together to create...

Abstractions

• Objects represent properties, interactions, & processes

• Object-oriented programming:
Objects

- Representations of information
- Data and behavior, bundled together to create...

**Abstractions**

- Objects represent properties, interactions, & processes
- Object-oriented programming:
  - A metaphor for organizing large programs
Objects

- Representations of information
- Data and behavior, bundled together to create...

**Abstractions**

- Objects represent properties, interactions, & processes
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax for implementing classic ideas
Objects

• Representations of information

• Data and behavior, bundled together to create...

  Abstractions

• Objects represent properties, interactions, & processes

• Object-oriented programming:
  • A metaphor for organizing large programs
  • Special syntax for implementing classic ideas

  (Demo)
In Python, every value is an object.
In Python, every value is an object.

- All objects have attributes
Python Objects

In Python, every value is an object.

• All objects have attributes

• A lot of data manipulation happens through methods
In Python, every value is an object.

- All objects have attributes
- A lot of data manipulation happens through methods
- Functions do one thing; objects do many related things
Python Objects

In Python, every value is an object.

- All objects have attributes
- A lot of data manipulation happens through methods
- Functions do one thing; objects do many related things

The next four weeks:
Python Objects

In Python, every value is an object.

- All objects have attributes
- A lot of data manipulation happens through methods
- Functions do one thing; objects do many related things

The next four weeks:

- Use built-in objects to introduce classic ideas
Python Objects

In Python, every value is an object.

- All objects have attributes
- A lot of data manipulation happens through methods
- Functions do one thing; objects do many related things

The next four weeks:

- Use built-in objects to introduce classic ideas
- Create our own objects using the built-in object system
Python Objects

In Python, every value is an object.

- All objects have attributes
- A lot of data manipulation happens through methods
- Functions do one thing; objects do many related things

The next four weeks:

- Use built-in objects to introduce classic ideas
- Create our own objects using the built-in object system
- Implement an object system using built-in objects
Native Data Types

In Python, every object has a type.
Native Data Types

In Python, every object has a type.

```python
>>> type(today)
<class 'datetime.date'>
```
Native Data Types

In Python, every object has a type.

```python
>>> type(today)
<class 'datetime.date'>
```

Properties of native data types:
In Python, every object has a type.

>>> type(today)
<class 'datetime.date'>

Properties of native data types:

1. There are primitive expressions that evaluate to native objects of these types.
Native Data Types

In Python, every object has a type.

```python
>>> type(today)
<class 'datetime.date'>
```

Properties of native data types:

1. There are primitive expressions that evaluate to native objects of these types.

2. There are built-in functions, operators, and methods to manipulate these objects.
Numeric Data Types
Numeric Data Types

Numeric types in Python:
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
```
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>
```
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>
```

Represents integers exactly
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>
```

```
 Represents integers exactly
```

```python
>>> type(1.5)
```

```
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

>>> type(1.5)
<class 'float'>
```

Represents integers exactly.
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

Represents integers exactly
```

```python
>>> type(1.5)
<class 'float'>

Represents real numbers approximately
```
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

Represented integers exactly

>>> type(1.5)
<class 'float'>

Represented real numbers approximately

>>> type(1+1j)
```

- Represents integers exactly
- Represents real numbers approximately
Numeric Data Types

Numeric types in Python:

```python
>>> type(2)
<class 'int'>

Represents integers exactly

>>> type(1.5)
<class 'float'>

Represents real numbers approximately

>>> type(1+1j)
<class 'complex'>
```

Working with Real Numbers

Care must be taken when computing with real numbers!

(Demo)
Care must be taken when computing with real numbers! (Demo)

Representing real numbers:

Care must be taken when computing with real numbers!
(Demo)

Representing real numbers:

1/3 =

Working with Real Numbers

Care must be taken when computing with real numbers!

(Demo)

Representing real numbers:

1/3 = 0011 1111 1101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101

Working with Real Numbers

Care must be taken when computing with real numbers!
(Demo)

Representing real numbers:

1/3 = 0011 1111 1101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101

False in a Boolean contexts:

Working with Real Numbers

Care must be taken when computing with real numbers!

(Demo)

Representing real numbers:

\[
\frac{1}{3} = 0011 \ 1111 \ 1101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101 \ 0101
\]

False in a Boolean contexts:

\[
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000
\]

Working with Real Numbers

Care must be taken when computing with real numbers!
(Demo)

Representing real numbers:

\[
\frac{1}{3} = \text{0011 1111 1101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101 0101}
\]

False in a Boolean contexts:

\[
\begin{align*}
0000 & 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 \\
1000 & 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
\end{align*}
\]

Working with Real Numbers
Working with Real Numbers

```python
>>> def approx_eq_1(x, y, tolerance=1e-18):
```
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance
Working with Real Numbers

>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance
Working with Real Numbers

>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
Working with Real Numbers

>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
        return True
Working with Real Numbers

```python
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
        return True
    return approx_eq_1(x, y) or approx_eq_2(x, y)
```

Bonus Material
Working with Real Numbers

```python
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
        return True
    return approx_eq_1(x, y) or approx_eq_2(x, y)
```

or `approx_eq_2(y, x)`
Working with Real Numbers

```python
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
        return True
    return approx_eq_1(x, y) or approx_eq_2(x, y)

>>> def near(x, f, g):
    or approx_eq_2(y, x)
```
Working with Real Numbers

```python
>>> def approx_eq_1(x, y, tolerance=1e-18):
    return abs(x - y) <= tolerance

>>> def approx_eq_2(x, y, tolerance=1e-7):
    return abs(x - y) <= abs(x) * tolerance

>>> def approx_eq(x, y):
    if x == y:
        return True
    return approx_eq_1(x, y) or approx_eq_2(x, y)

>>> def near(x, f, g):
    return approx_eq(f(x), g(x))
```

or approx_eq_2(y, x)
Moral of the Story
Moral of the Story

Life was better when numbers were just numbers!
Moral of the Story

Life was better when numbers were just numbers!

Having to know the details of an abstraction:
Moral of the Story

Life was better when numbers were just numbers!

Having to know the details of an abstraction:

- Makes programming harder and more knowledge-intensive
Moral of the Story

Life was better when numbers were just numbers!

Having to know the details of an abstraction:

• Makes programming harder and more knowledge-intensive

• Creates opportunities to make mistakes
Moral of the Story

Life was better when numbers were just numbers!

Having to know the details of an abstraction:

• Makes programming harder and more knowledge-intensive
• Creates opportunities to make mistakes
• Introduces dependencies that prevent future changes
Moral of the Story

Life was better when numbers were just numbers!

Having to know the details of an abstraction:

- Makes programming harder and more knowledge-intensive
- Creates opportunities to make mistakes
- Introduces dependencies that prevent future changes

Coming Soon: Data Abstraction