61A Lecture 7

Monday, September 12

Pig Contest Rules

- The score for an entry is the sum of win rates against every other entry.
- All strategies must be deterministic functions of the current score! Non-deterministic strategies will be disqualified.
- Winner: 3 points extra credit on Project 1
- Second place: 2 points
- Third place: 1 point
- The real prize: honor and glory
- To enter: submit a file pig.py that contains a function called final_strategy as assignment p1contest by Monday, 9/26

Function Decorators

(demo)

Function Decorators

(demo)

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

Function Decorators

(demo)

Function decorator

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

Function Decorators

(demo)

Function decorator

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

Decorated function

Function Decorators

(demo)

Function decorator

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

Decorated function

is identical to

Function Decorators

(demo)

Function decorator

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

Decorated function

is identical to

```
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)
```

The Art of the Function

Each function should have exactly one job

Don't repeat yourself (DRY)

Functions should be defined generally

The Art of the Function

Practical guidance

Each function should have exactly one job

Don't repeat yourself (DRY)

Functions should be defined generally

The Art of the Function

Practical guidance

Each function should have exactly one job

Separation of concerns

Don't repeat yourself (DRY)

Functions should be defined generally

Practical guidance

The Art of the Function

Each function should have exactly one job

Separation of concerns

Testing functions stay small

Don't repeat yourself (DRY)

Functions should be defined generally

The Art of the Function

Practical guidance

Each function should have exactly one job

Separation of concerns

Testing functions stay small

Don't repeat yourself (DRY)

Revisions should require few code changes

Functions should be defined generally

The Art of the Function

Practical guidance

Each function should have exactly one job

Separation of concerns

Testing functions stay small

Don't repeat yourself (DRY)

Revisions should require few code changes

Isolates problems

Functions should be defined generally

The Art of the Function

Practical guidance

Each function should have exactly one job

- *Separation of concerns*
- *Testing functions stay small*

Don't repeat yourself (DRY)

- *Revisions should require few code changes*
- *Isolates problems*

Functions should be defined generally

- *Writing fewer lines of code saves you time*

The Art of the Function

Practical guidance

Each function should have exactly one job

- *Separation of concerns*
- *Testing functions stay small*

Don't repeat yourself (DRY)

- *Revisions should require few code changes*
- *Isolates problems*

Functions should be defined generally

- *Writing fewer lines of code saves you time*
- *Copy/Paste has a steep price*

The Art of the Function

Practical guidance

Each function should have exactly one job

Separation of concerns

Testing functions stay small

Don't repeat yourself (DRY)

Revisions should require few code changes

Isolates problems

Functions should be defined generally

Writing fewer lines of code saves you time

Copy/Paste has a steep price

These are guidelines, not strict rules!

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over

Practical guidance

Choosing Names

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over
d	dice

Practical guidance

Choosing Names

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over
d	dice
play_helper	take_turn

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	To:
boolean	turn_is_over
d	dice
play_helper	take_turn

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

Practical guidance

Choosing Names

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over
d	dice
play_helper	take_turn

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

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over
d	dice
play_helper	take_turn

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

Practical guidance

Choosing Names

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	To:
boolean	turn_is_over
d	dice
play_helper	take_turn

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

Choosing Names

Practical guidance

Names typically *don't* matter for correctness

but

they matter tremendously for legibility

From:	**To:**
boolean	turn_is_over
d	dice
play_helper	take_turn

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

Functional Abstractions

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

```
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 to use it?

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 to use it?

- Square takes one argument.

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 to use it?

- Square takes one argument. **Yes**

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 to use it?

- 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 to use it?

- Square takes one argument. **Yes**
- Square has the intrinsic name “square”. **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 to use it?

- 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 to use it?

- Square takes one argument. **Yes**
- Square has the intrinsic name “square”. **No**
- Square computes the square of a number. **Yes**

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 to use it?

- 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

```
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 to use it?

- 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 to use it?

- 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

```
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 to use it?

- 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
```

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 to use it?

- 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**

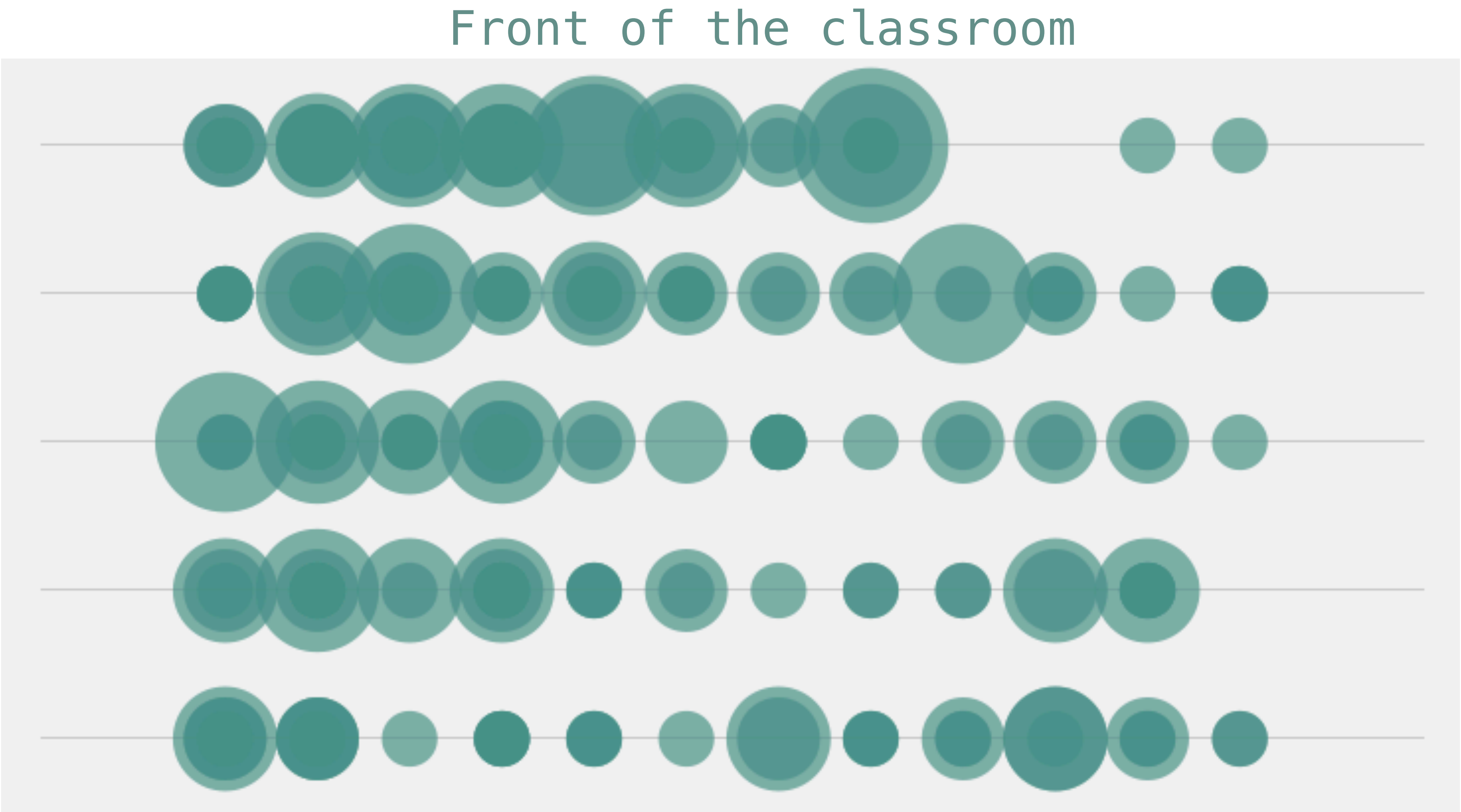
```
def square(x):
    return pow(x, 2)
```

```
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/

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)

Python Objects

In Python, every value is an object.

Python Objects

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`

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

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 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 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 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.

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

Native Data Types

In Python, every object has a type.

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

Properties of native data types:

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.

```
>>> 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

```
Three numeric types in Python:
```

Numeric Data Types

Three numeric types in Python:

```
>>> type(2)
```

Numeric Data Types

Three numeric types in Python:

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

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

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

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

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

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

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

Represents real numbers approximately

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

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

Represents real numbers approximately

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

Numeric Data Types

Three numeric types in Python:

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

Represents integers exactly

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

Represents real numbers approximately

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

Numeric Data Types

Four

~~Three~~ numeric types in Python:

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

Represents integers exactly

(demo)

```
>>> 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!

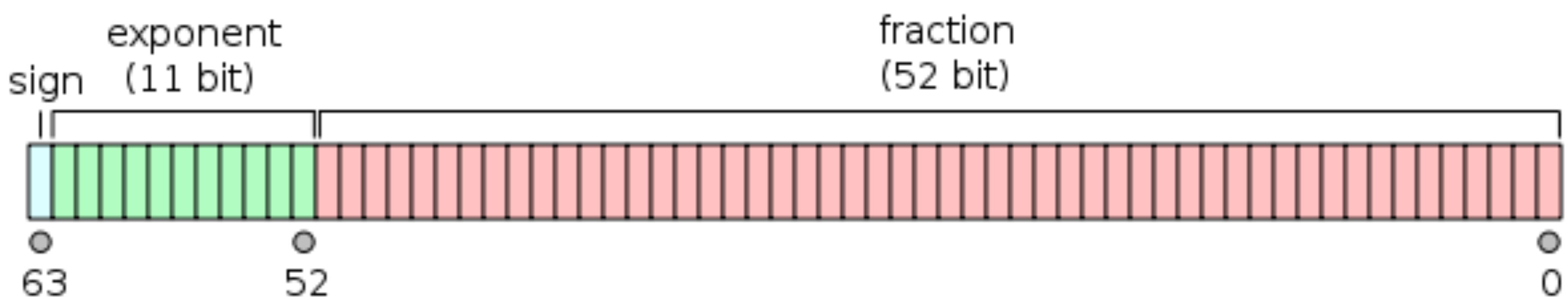
(Demo)

Working with Real Numbers

Care must be taken when computing with real numbers!

(Demo)

Representing real numbers:



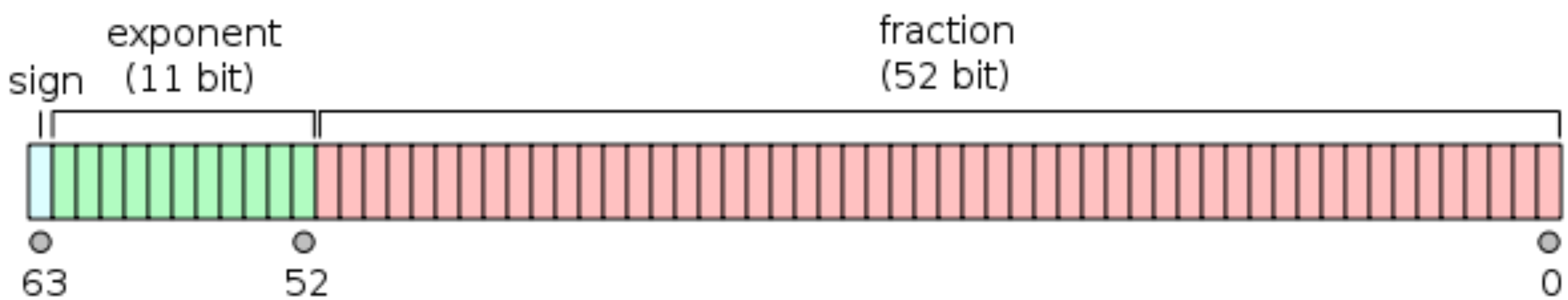
http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

Working with Real Numbers

Care must be taken when computing with real numbers!

(Demo)

Representing real numbers:



1/3 =

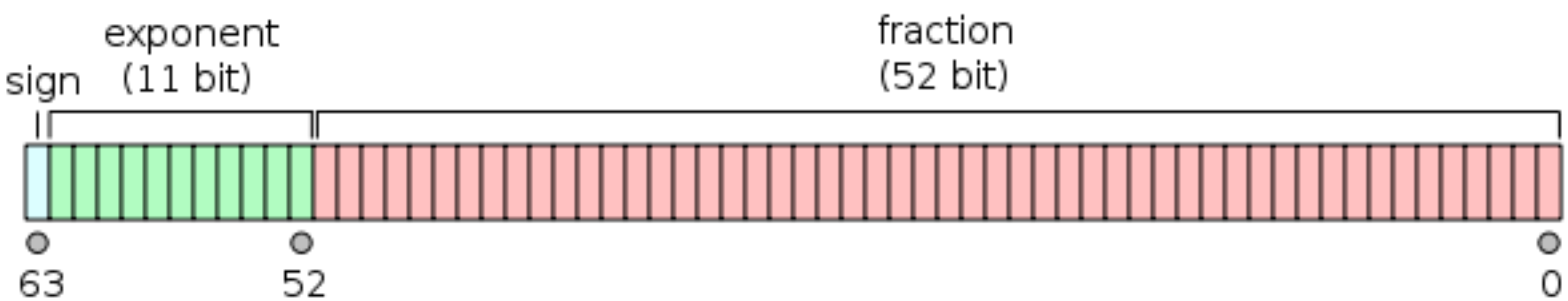
http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

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

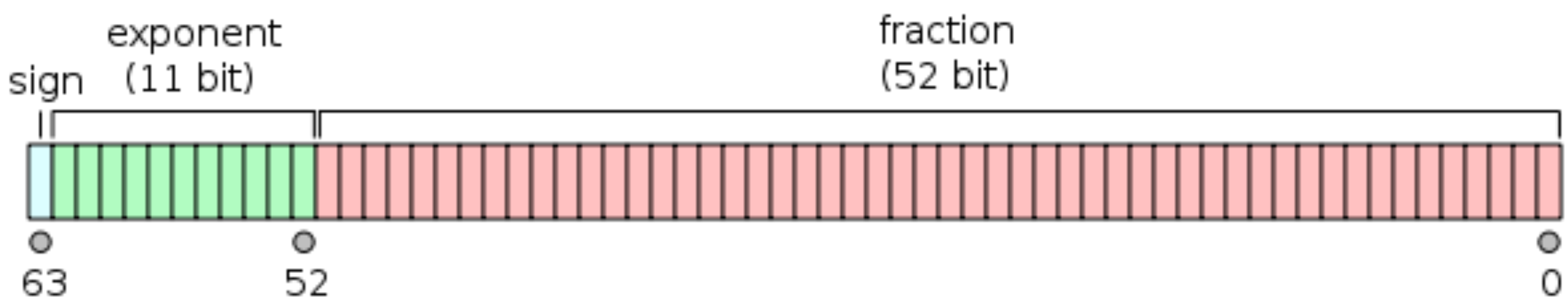
http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

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

False in a Boolean contexts:

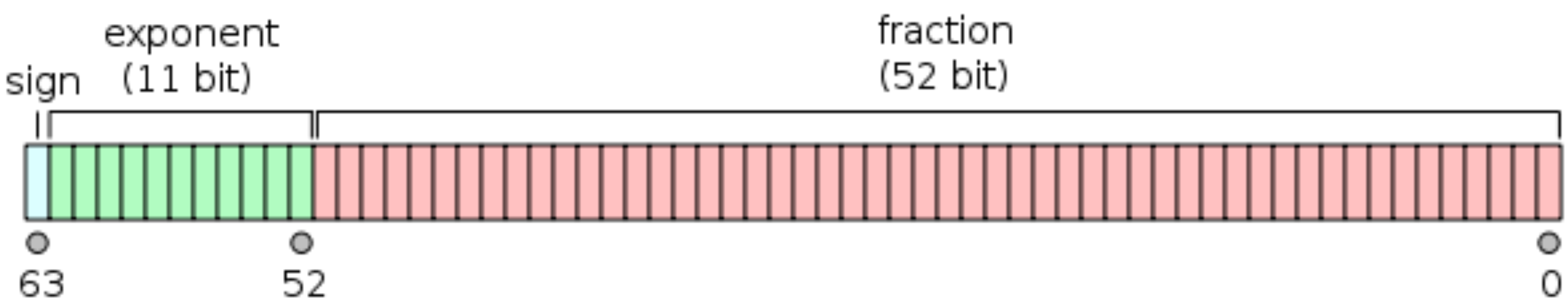
http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

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

False in a Boolean contexts:

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

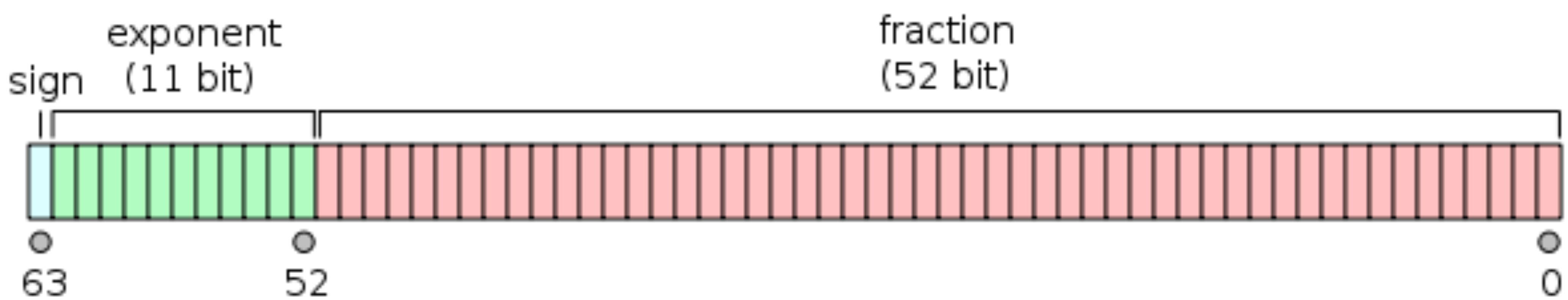
http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

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`

False in a Boolean contexts:

`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`

http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg

Working with Real Numbers

Working with Real Numbers

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

Working with Real Numbers

```
>>> 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):
```

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
```

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):
```

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:
```

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

```
>>> 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)
```

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
        return approx_eq_1(x, y) or approx_eq_2(x, y)
```

```
or approx_eq_2(y,x)
```

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
        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

```
>>> 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!
```

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