Implementing Dice

Implementing Dice

Random numbers are useful for experimentation

Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

http://www.math.utah.edu/~pa(Random/Random.html
Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,
- Primality tests

Implementing Dice

Random numbers are useful for experimentation

They also appear in lots of algorithms, e.g.,

• Primality tests
• Machine learning techniques

Implementing Dice

Random numbers are useful for experimentation. They also appear in lots of algorithms, e.g.,
• Primality tests
• Machine learning techniques

Implementing Dice

Random numbers are useful for experimentation. They also appear in lots of algorithms, e.g.,
- Primality tests
- Machine learning techniques

```python
def make_dice(sides=6):
    seed = 1
    def dice():
        nonlocal seed
        seed = (16807 * seed) % 2147483647
        return seed % sides + 1
    return dice

```
Implementing Dice

Random numbers are useful for experimentation
They also appear in lots of algorithms, e.g.,
- Primality tests
- Machine learning techniques

def make_dice(sides=6):
    seed = 1
    def dice():
        nonlocal seed
        seed = (16807 * seed) % 2147483647
        return seed % sides + 1
    return dice

Implementing Dice

Random numbers are useful for experimentation. They also appear in lots of algorithms, e.g.,
- Primality tests
- Machine learning techniques

```python
def make_dice(sides=6):
    seed = 1
    def dice():
        nonlocal seed
        seed = (16807 * seed) % 2147483647
        return seed % sides + 1
    return dice
```

Implementing Dice

Random numbers are useful for experimentation. They also appear in lots of algorithms, e.g.,
- Primality tests
- Machine learning techniques

def make_dice(sides=6):
    seed = 1
    def dice():
        nonlocal seed
        seed = (16807 * seed) % 2147483647
        return seed % sides + 1
    return dice


Referential Transparency, Lost
Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.
• Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

\[ \text{mul(add(2, mul(4, 6)), add(3, 5))} \]
Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

```
mul(add(2, mul(4, 6)), add(3, 5))
mul(add(2, 24), add(3, 5))
```
Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

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

\[
\text{mul}\left(\text{add}(2, 24), \text{add}(3, 5)\right)
\]

\[
\text{mul}(26, \text{add}(3, 5))
\]
Referential Transparency, Lost

• Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a program.

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

\[
\text{mul}(\text{add}(2, 24), \text{add}(3, 5))
\]

\[
\text{mul}(26, \text{add}(3, 5))
\]

• Re-binding operations violate the condition of referential transparency because they let us define functions that do more than just return a value; we can change the environment, causing values to mutate.
• Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

```
mul(add(2, mul(4, 6)), add(3, 5))
mul(add(2, 24), add(3, 5))
mul(26, add(3, 5))
```

• Re-binding operations violate the condition of referential transparency because they let us define functions that do more than just return a value; we can change the environment, causing values to mutate.
• Expressions are referentially transparent if substituting an expression with its value does not change the meaning of a program.

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

\[
mul(add(2, 24), add(3, 5))
\]

\[
mul(26, add(3, 5))
\]

• Re-binding operations violate the condition of referential transparency because they let us define functions that do more than just return a value; we can change the environment, causing values to mutate.
Expresssions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

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

\[
\text{mul}(\text{add}(2, 24 ), \text{add}(3, 5))
\]

\[
\text{mul}(26 , \text{add}(3, 5))
\]

Re-binding operations violate the condition of referential transparency because they let us define functions that do more than just return a value; we can change the environment, causing values to mutate.

Demo
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    get, put = make_container('Hi')
```
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    get, put = make_container('Hi')
```
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    get, put = make_container('Hi')
```

Monday, September 26, 2011
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    get, put = make_container('Hi')
```
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = make_container('Hi')
```

Monday, September 26, 2011
Implementing a Mutable Container Object

```python
def make_container(contents):
    get = lambda: contents
    put = lambda value: nonlocal contents; contents = value
    return get, put

g, p = make_container('Hi')
def g(): return g()
def p(value):
    nonlocal contents
    contents = value
    return g, p
```

Monday, September 26, 2011
Implementing a Mutable Container Object

```
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = make_container('Hi')
```
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

g, p = make_container('Hi')
```

---

Monday, September 26, 2011
Implementing a Mutable Container Object

```python
def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
    return get, put

get, put = make_container('Hi')
```
Python Lists

['Demo']

http://docs.python.org/py3k/library/stdtypes.html#mutable-sequence-types
suits = ['♥', '♦', '♠', '♣']
nest = list(suits)
nest[0] = suits
nest[0][2]
suits.append('Joker')
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']

nest = list(suits)
nest[0] = suits
nest[0][2]
suits.append('Joker')
suits = suits
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']

nest = list(suits)
nest[0] = suits

nest[0][2]
suits.append('Joker')
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']
nest = list(suits)
nest[0] = suits
nest[0][2] = suits
suits.append('Joker')
nest[0].pop()
suits = ['\heartsuit', '\diamondsuit', '\spadesuit', '\clubsuit']

nest = list(suits)
nest[0] = suits

nest[0][2]
suits.append('Joker')
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']

nest = list(suits)
nest[0] = suits

nest[0][2]
suits.append('Joker')
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']

nest = list(suits)
nest[0] = suits

nest[0][2]
suits.append('Joker')
nest[0].pop()
suits = ['♥', '♦', '♠', '♣']

nest = list(suits)
nest[0] = suits

nest[0][2]
suits.append('Joker')
nest[0].pop()
Testing for Identity

```python
>>> suits is nest[0]
True
>>> suits is ['♥', '♦', '♠', '♣']
False
>>> suits == ['♥', '♦', '♠', '♣']
True
```
List Comprehensions
List Comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Monday, September 26, 2011
List Comprehensions

$$\left[ \text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \ if \ <\text{filter exp}> \right]$$

Short version: $$\left[ \text{map exp} \ for \ <\text{name}> \ in \ <\text{iter exp}> \right]$$
List Comprehensions

\[<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>]\]

Short version: \[<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}>]\]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.
List Comprehensions

```python
[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

>>> suits = ['heart', 'diamond', 'spade', 'club']
```
List Comprehensions

\[<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>\]

Short version: \[<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}>\]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

```python
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
```
List Comprehensions

\[
[\text{\texttt{map exp}} \ \text{for} \ \text{\texttt{name}} \ \text{in} \ \text{\texttt{iter exp}} \ \text{if} \ \text{\texttt{filter exp}}]
\]

Short version: \[
[\text{\texttt{map exp}} \ \text{for} \ \text{\texttt{name}} \ \text{in} \ \text{\texttt{iter exp}}]
\]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

```python
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]
```
List Comprehensions

[<map exp> for <name> in <iter exp> if <filter exp>]

Short version: [<map exp> for <name> in <iter exp>]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

>>> suits = ['heart', 'diamond', 'spade', 'club']

>>> from unicodedata import lookup

>>> [lookup('WHITE ' + s.upper() + ' SUIT') for s in suits]

[♡, ♢, ♣, ♤]
Dispatch Functions

A technique for packing multiple behaviors into one function
A technique for packing multiple behaviors into one function

def make_pair(x, y):
    """Return a function that behaves like a pair."""

def dispatch(m):
    if m == 0:
        return x
    elif m == 1:
        return y

return dispatch
Dispatch Functions

A technique for packing multiple behaviors into one function

```python
def make_pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common
Dispatch Functions

A technique for packing multiple behaviors into one function

```python
def make_pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:
Dispatch Functions

A technique for packing multiple behaviors into one function

```python
def make_pair(x, y):
    """Return a function that behaves like a pair.""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

• One conditional statement with several clauses
Dispatch Functions

A technique for packing multiple behaviors into one function

```python
def make_pair(x, y):
    """Return a function that behaves like a pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

Message argument can be anything, but strings are most common

The body of a dispatch function is always the same:

• One conditional statement with several clauses

• Headers perform equality tests on the message
A Mutable Container That Uses Message Passing
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):
```

Monday, September 26, 2011
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):
    
def dispatch(message, value=None):
```
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):
    
def dispatch(message, value=None):
        nonlocal contents
```
def make_container_dispatch(contents):
    
def dispatch(message, value=None):
        nonlocal contents
        if message == 'get':

def make_container_dispatch(contents):
    def dispatch(message, value=None):
        nonlocal contents

        if message == 'get':
            return contents
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':
            return contents

        if message == 'put':

            return contents
```

Monday, September 26, 2011
def make_container_dispatch(contents):

    def dispatch(message, value=None):

        nonlocal contents

        if message == 'get':

            return contents

        if message == 'put':

            contents = value
def make_container_dispatch(contents):
    
def dispatch(message, value=None):
        
        nonlocal contents
        
        if message == 'get':

            return contents
        
        if message == 'put':

            contents = value

        return dispatch

Monday, September 26, 2011
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):
    def dispatch(message, value=None):
        nonlocal contents

        if message == 'get':
            return contents

        if message == 'put':
            contents = value

        return dispatch

    return dispatch

def make_container(contents):
    def get():
        return contents

    def put(value):
        nonlocal contents
        contents = value

    return get, put
```

Monday, September 26, 2011
A Mutable Container That Uses Message Passing

def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    return get, put

def make_container_dispatch(contents):
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'get':
            return contents
        if message == 'put':
            contents = value
            return dispatch
        return dispatch
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get
    return get, put
A Mutable Container That Uses Message Passing

```python
def make_container_dispatch(contents):
    def dispatch(message, value=None):
        nonlocal contents

        if message == 'get':
            return contents
        if message == 'put':
            contents = value
            return dispatch

    return dispatch

def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get(), put

    return get, put
```
A Mutable Container That Uses Message Passing

```python
def make_containerDispatch(contents):
    def dispatch(message, value=None):
        nonlocal contents
        if message == 'get':
            return contents
        if message == 'put':
            contents = value
            return dispatch
    return dispatch

def make_container(contents):
    def get():
        return contents
    def put(value):
        nonlocal contents
        contents = value
        return get, put
    return get, put
```

Demo
Implementing Mutable Recursive Lists
Implementing Mutable Recursive Lists

```python
def makemutablerlist():
```

Recursive List Refresher Demo
Implementing Mutable Recursive Lists

def make_mutable_rlist():
    contents = empty_rlist

Recursive List Refresher Demo
Implementing Mutable Recursive Lists

```python
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
```
Implementing Mutable Recursive Lists

```python
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
```
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
Implementing Mutable Recursive Lists

def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        pass
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        ...
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
Implementing Mutable Recursive Lists

```python
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        returngetitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
```

Recursive List Refresher Demo
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
Implementing Mutable Recursive Lists

def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        returngetitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
Implementing Mutable Recursive Lists

```python
def makeMutableRlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
        return f
```
Implementing Mutable Recursive Lists

```python
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
        return f
    elif message == 'str':
        Recursive List Refresher Demo
```

Monday, September 26, 2011
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
        return f
    elif message == 'str':
        return str(contents)
def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
        return f
    elif message == 'str':
        return str(contents)
    return dispatch
Implementing Mutable Recursive Lists

def make_mutable_rlist():
    contents = empty_rlist

def dispatch(message, value=None):
    nonlocal contents
    if message == 'len':
        return len_rlist(contents)
    elif message == 'getitem':
        return getitem_rlist(contents, value)
    elif message == 'push_first':
        contents = make_rlist(value, contents)
    elif message == 'pop_first':
        f = first(contents)
        contents = rest(contents)
        return f
    elif message == 'str':
        return str(contents)

return dispatch

Recursive List Refresher Demo

Demo