

61A Lecture 12

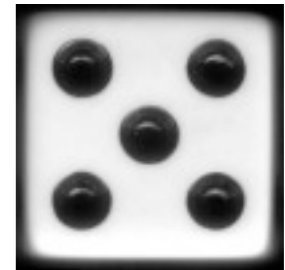
Monday, September 26

Implementing Dice

Random numbers are useful for experimentation

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

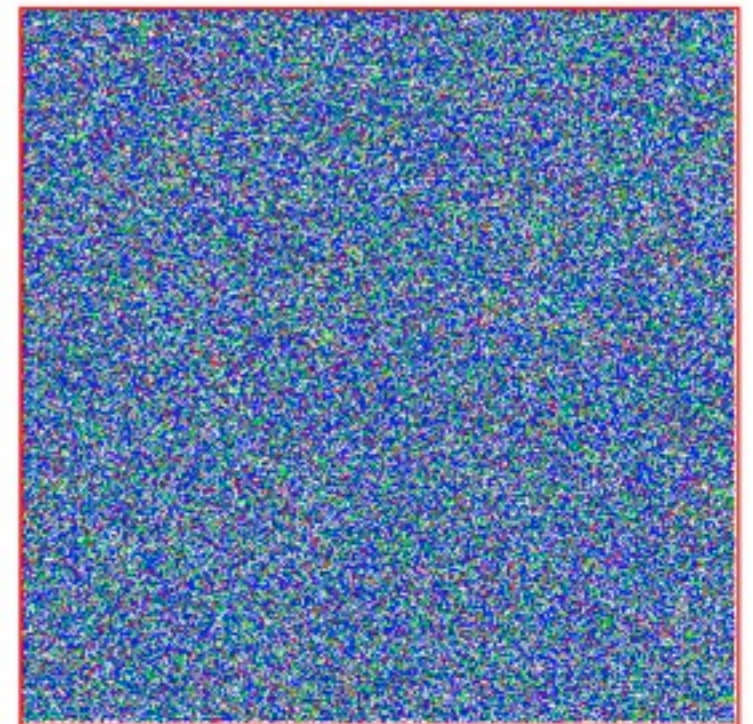
- Primality tests
- Machine learning techniques



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

$2^{31} - 1$

P1 = 16807, P2 = 0, N = 2147483647



100000 dots drawn, seed = 1

<http://www.math.utah.edu/~pa/Random/Random.html>

S.K. Park and K.W. Miller, "Random Number Generators: Good Ones Are Hard To Find", Communications of the ACM, October 1988, pp. 1192-1201.

Referential Transparency, Lost

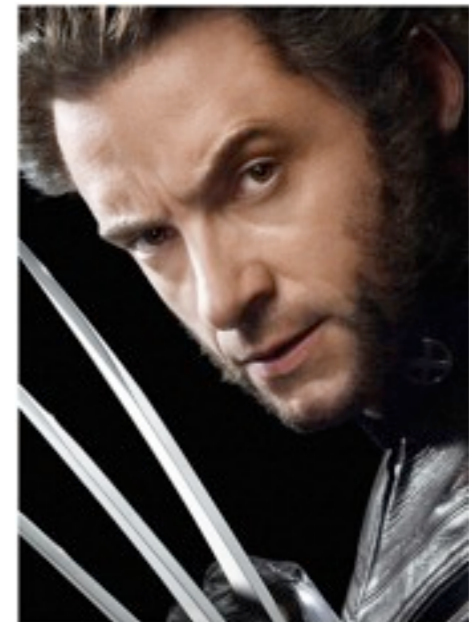
- Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.



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

```
mul(add(2, 24), add(3, 5))
```

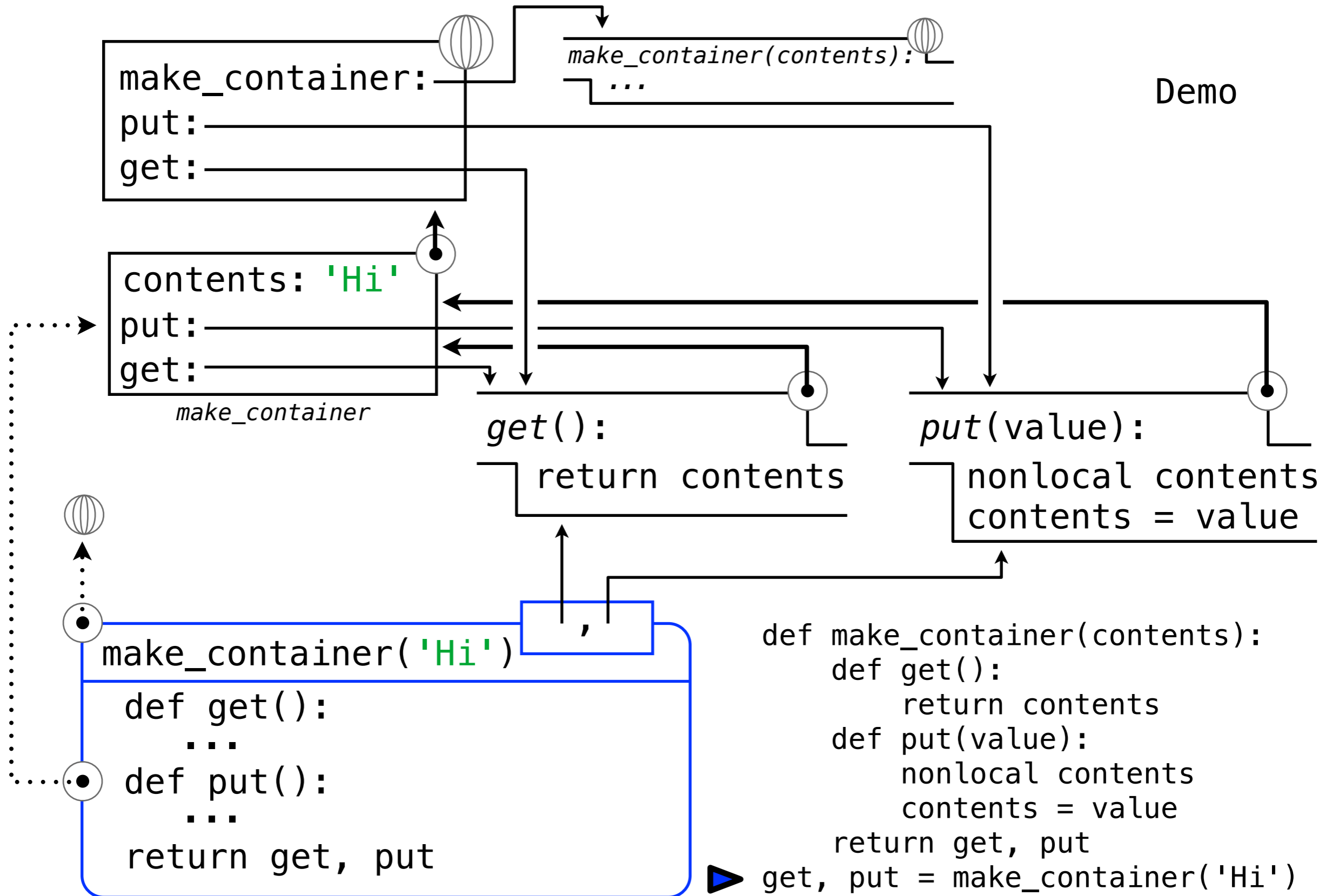
```
mul(26, add(3, 5))
```



- 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.

Demo

Implementing a Mutable Container Object

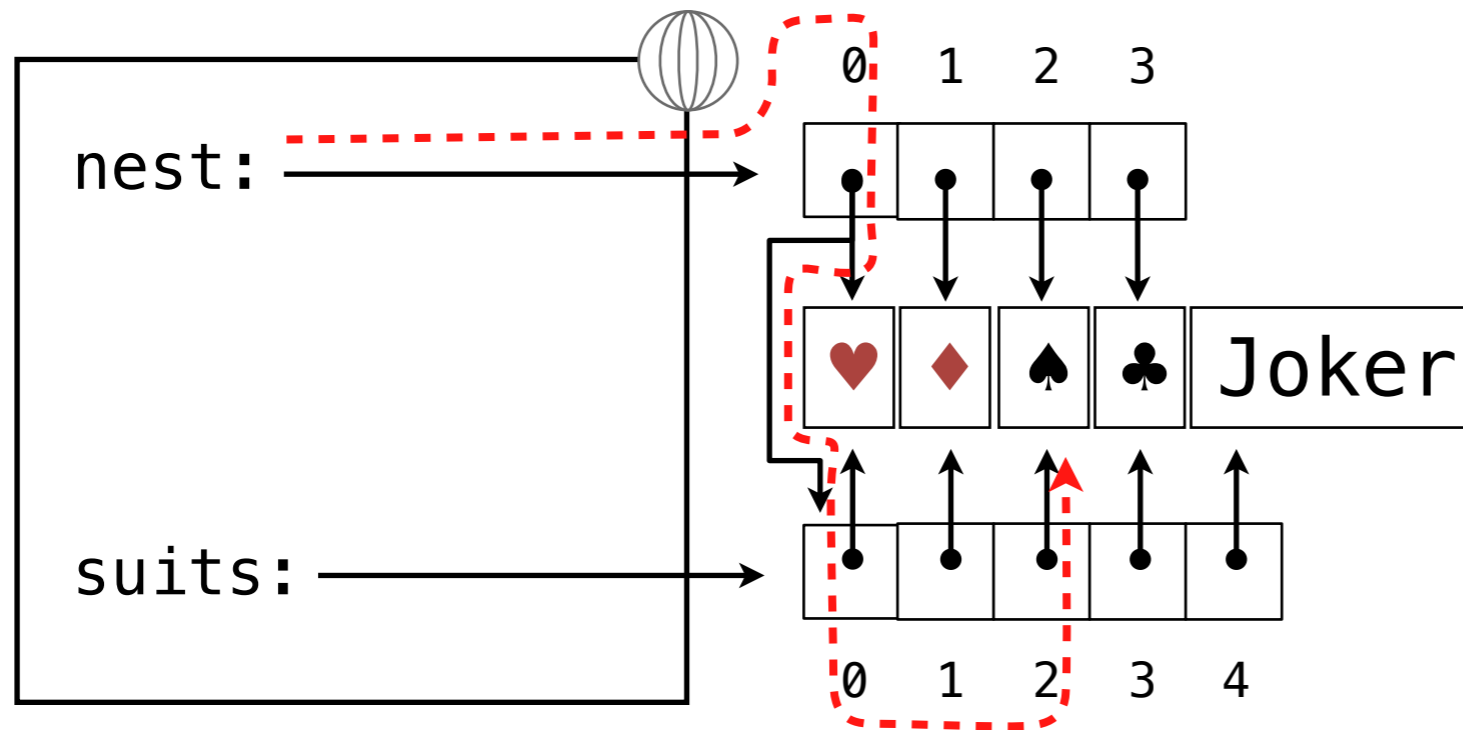


Python Lists

['Demo']

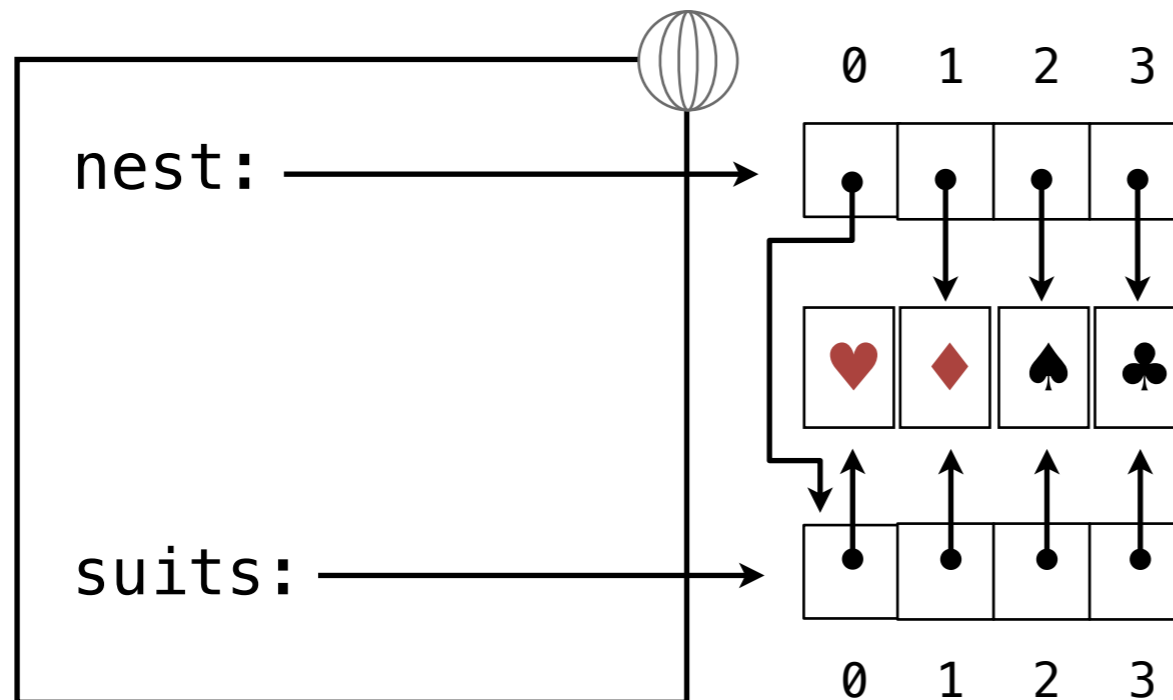
<http://docs.python.org/py3k/library/stdtypes.html#mutable-sequence-types>

Sharing and Identity with Lists



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

Testing for Identity



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

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

```
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

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

```
    def dispatch(message, value=None):
```

```
        nonlocal contents
```

```
        if message == 'get':
```

```
            return contents
```

```
        if message == 'put':
```

```
            contents = value
```

```
    return dispatch
```

```
def make_container(contents):
```

```
    def get():
```

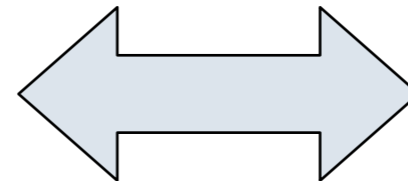
```
        return contents
```

```
    def put(value):
```

```
        nonlocal contents
```

```
        contents = value
```

```
    return get, put
```



Demo

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