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.

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

make_container('Hi')
```

Demo

Python Lists

- Sharing and Identity with Lists

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

http://docs.python.org/py3k/library/stdtypes.html#mutable-sequence-types
Testing for Identity

```
>>> suits is nest[0]
True
>>> suits is ['w', 'd', 's', 'c']
False
>>> suits == ['w', 'd', 's', 'c']
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
```

The body of a dispatch function is always the same:

- One conditional statement with several clauses
- Headers perform equality tests on the message

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