== versus is

```
class Card(object):

  def __init__(self, s, r):
    self.suit = s
    self.rank = r

  def __eq__(self, other):
    return (isinstance(other,Card) and
            (self.suit, self.rank) ==
            (other.suit, other.rank))

  def __ne__(self, other):
    return not self.__eq__(other)

c = Card(3,2)
d = Card(3,2)
e = c
print c     # <Card object at 0x100497b10>
print d     # <Card object at 0x100497b50>
print c == d    # True
print c is d    # False
print e is c    # True
```

- When you define a class, you might like to define what it means for instances to be equal.
- To do this you define the __eq__ and __ne__ methods (overriding the default ones in the class object).[1]
- But now what if you want to tell if two cards are the same object? Use is instead of ==.
- And, whenever you are really talking about equality of object identity, use is (e.g. is None).

[1]Before you do this for real, read about __hash__.

Dispatch on Type

- Sometimes you have an object that might be one of several types, and you need to know which it is.
- Example: have list of GObject instances, want to turn all the ellipses green and the rectangles blue. Leave the other shapes alone. Simple enough:

```
for shape in shapes:
    if type(shape) is GEllipse:
        shape.fillcolor = colormodel.GREEN
    elif type(shape) is GRectangle:
        shape.fillcolor = colormodel.BLUE
```

Finds any object whose class is GEllipse.

Dispatch on Type

- Problem: some of your shapes might actually be subclasses (in A7, a GEllipse might be the Ball).
- Solution: the built-in function isinstance. It answers the question, "Is this object an instance of this class?" and an instance of a subclass counts.

```
for shape in shapes:
    if isinstance(shape, GEllipse):
        shape.fillcolor = colormodel.GREEN
    elif isinstance(shape, GRectangle):
        shape.fillcolor = colormodel.BLUE
```

Finds any object whose class is GEllipse **or a subclass** of GEllipse.

Important example of type-based dispatch

```
try:
    input = raw_input()
    x = float(input)
    print 'The next number is '+str(x+1)
except ValueError:
    print 'Hey! That is not a number!'

try:
    input = raw_input()
    x = float(input)
    print 'The next number is '+str(x+1)
except StandardError:
    print 'Hey! We had a problem!'
```

Matches any exception whose class is ValueError **or any subclass** of ValueError.

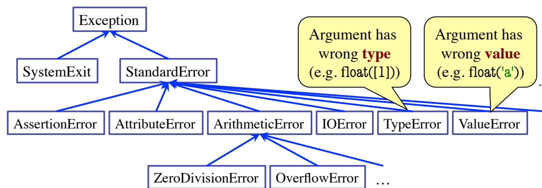
could raise IOError

could raise ValueError

Matches any exception whose class is StandardError **or any subclass** of StandardError (including ValueError or IOError).

- Exception handling effectively uses isinstance to match exceptions to exception handlers.

Recall: Hierarchy of exceptions



http://docs.python.org/library/exceptions.html

Dispatch on Type vs. Method Overriding

```
class Ball(object):
  ...
class SuperBall(Ball):
  ...
class BallOfClay(Ball):
  ...
...
if isinstance(my_ball, SuperBall):
  ball.vy = -0.99 * ball.vy
elif isinstance(my_ball, BallOfClay):
  ball.vy = -0.05 * ball.vy
else:
  ball.vy = -0.5 * ball.vy
```

```
class Ball(object):
  def rebound(self):
    self.vy = -0.5 * self.vy
  ...
class SuperBall(Ball):
  def rebound(self):
    self.vy = -0.99 * self.vy
  ...
class BallOfClay(Ball):
  def rebound(self):
    self.vy = -0.05 * self.vy
  ...
...
my_ball.rebound()
```

Dictionaries (Type dict)

Description

- List of key-value pairs
  - Keys are unique
  - Values need not be
- Example: net-ids
  - net-ids are **unique** (a key)
  - names need not be (values)
  - js1 is John Smith (class '13)
  - js2 is John Smith (class '16)
- Many other applications

Python Syntax

- Create with format: {k1:v1, k2:v2, ...}
- Keys must be non-mutable
  - ints, floats, bools, strings
  - **Not** lists or custom objects
- Values can be anything
- Example:
  d = {'js1':'John Smith', 'js2':'John Smith', 'wmw2':'Walker White'}

Using Dictionaries (Type dict)

- Access elts. like a list
  - d['js1'] evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - d['js1'] = 'Jane'
  - Can add new keys
  - d['aa1'] = 'Allen'
  - Can delete keys
  - del d['wmw2']

d = {'js1':'John','js2':'John', 'wmw2':'Walker'}

Using Dictionaries (Type dict)

- Access elts. like a list
  - d['js1'] evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - d['js1'] = 'Jane'
  - Can add new keys
  - d['aa1'] = 'Allen'
  - Can delete keys
  - del d['wmw2']

d = {'js1':'John','js2':'John', 'wmw2':'Walker'}

Using Dictionaries (Type dict)

- Access elts. like a list
  - d['js1'] evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - d['js1'] = 'Jane'
  - Can add new keys
  - d['aa1'] = 'Allen'
  - Can delete keys
  - del d['wmw2']

d = {'js1':'John','js2':'John', 'wmw2':'Walker'}

Using Dictionaries (Type dict)

- Access elts. like a list
  - d['js1'] evaluates to 'John'
  - But cannot slice ranges!
- Dictionaries are **mutable**
  - Can reassign values
  - d['js1'] = 'Jane'
  - Can add new keys
  - d['aa1'] = 'Allen'
  - Can delete keys
  - del d['wmw2']

d = {'js1':'John','js2':'John', 'wmw2':'Walker'}

Dictionaries and For-Loops

- Dictionaries != sequences
  - Cannot slice them
  - Cannot use in for-loop
- But have methods to give you related sequences
  - Seq of keys: d.keys()
  - Seq of values: d.values()
  - Seq of key-value pairs: d.items()
- Use these in for-loops

```
for k in d.keys():
    print k
    print d[k]

for v in d.values():
    print v

for k,v in d.items():
    print k
    print v
```