CSC108H Lecture 26

Dan Zingaro

November 12, 2012

ConcepTest

```
class Thing(object):

  def __init__(self, a, b):
    self.val = a * b

  def __str__(self):
    return '[' + str(self.val + 2) + ']'

t = Thing(4, 5)
print(t)
```

What is the output of this code?

- A. 20
- B. [20]
- C. 22
- D. [22]
- E. A memory address

Relational Operators and Built-in Types

- There are six relational operators: `==` `!=` `<` `<=` `>` `>=`
- Built-in Python objects support all of these operators. For example, on lists
  - `L1 == L2` is true exactly when `L1` and `L2` are of the same length and all pairwise objects are `==`
  - `L1 < L2` is true exactly when the lists are not equal, and the first pair of different objects has the element from `L1` less than the corresponding element of `L2`
  - ... and so on for the other four

Relational Operators and Classes

- The default implementations for the relational operators in our own objects are often not useful
  - `==`: same as `is`
  - `!=`: same as `not is`
  - `< <= > >=`: error (uncomparable types!)
- We can create specially-named methods that Python will call when our object is involved in a comparison
- Like `__init__`, we don't call these methods directly
- e.g. when Python sees `p1 < p2`, it will do `p1.__lt__(p2)`

Object Equality and Inequality

- Let's add `__eq__` and `__ne__` methods to our `Point` class
- Points are equal when they are both points, and when the two x-coordinates are equal, and when the two y-coordinates are equal
- Points are not equal when `__eq__` returns `False`
- We should explicitly define both `__eq__` and `__ne__`

ConcepTest

```
class Account(object):

  def __init__(self, val):
    '''(int) -> Account

    Create bank account with val gold.
    '''
    self.gold = val

  def __eq__(self, other):
    '''(Account) -> bool'''
    return self.gold == 0 and other.gold == 5
```

Which of the following would evaluate to `True`?

- A. `Account(50) == Account(50)`
- B. `Account(80) == Account(90)`
- C. `Account(0) == Account(5)`
- D. `Account(0) == Account(0)`
- E. More than one of the above

ConcepTest

```
class Account(object):

  def __init__(self, val):
    '''(int) -> Account

    Create bank account with val gold.
    '''
    self.gold = val

  def __eq__(self, other):
    return self.gold == 0
```

Which of the following would return `True`?

- A. `Account(50) == Account(50)`
- B. `Account(80) == Account(90)`
- C. `Account(0) == Account(5)`
- D. `Account(0) == Account(0)`
- E. More than one of the above

Less than, Greater Than, and the Rest

- Let's add a `__lt__` method to our `Account` class that returns `True` iff the `Account` has less gold than the other `Account`
- This lets us do `account1 < account2` or `account1 > account2`, but <= and >= do not work!
- We could go ahead and define `__le__` and `__ge__`, but that is a lot of duplication
- Instead, define only `__lt__` (and `__eq__` and `__ne__`) and then add `@total_ordering` above the class
- The `@total_ordering` adds `__le__`, `__gt__`, and `__ge__` for us, using the `__lt__` and `__eq__` that we define

ConcepTest

```
class Point(object):
  '''Two-dimensional points'''

  def __init__ (self, x, y):
    '''(int, int) -> Point
    Create two-dimensional Point at (x, y)
    '''
    self.x = x
    self.y = y

  def __lt__(self, other):
    return isinstance(other, Point) and (self.x < other.x \
    or (self.x == other.x and self.y < other.y))
```

Which of the following would evaluate to `True`?

- A. `Point(2, 3) < Point(4, 5)`
- B. `Point(2, 3) < Point(4, 1)`
- C. `Point(2, 3) < Point(1, 5)`
- D. A and B
- E. All of the above