Sharing, mutability, and immutability

Ruth Anderson
CSE 160
University of Washington
Copying and mutation

```python
list1 = ["e1", "e2", "e3", "e4"]
list2 = list1
list3 = list(list1)  # make a copy; also "list1[:]"
print list1, list2, list3
list1.append("e5")
list2.append("e6")
list3.append("e7")
print list1, list2, list3
list1 = list3
list1.append("e8")
print list1, list2, list3
```
Variable reassignment vs. Object mutation

- **Reassigning a variable** changes a binding, it does not change (mutate) any object.
  
  Reassigning is **always** done via the syntax:
  
  \[
  \texttt{myvar} = \texttt{expr} \quad \texttt{size} = 6 \\
  \texttt{list2} = \texttt{list1}
  \]

  Changes what the variables \texttt{size} and \texttt{list2} are bound to

- **Mutating (changing) an object** does not change any variable binding.
  
  Two syntaxes:
  
  \[
  \texttt{left_expr} = \texttt{right_expr} \\
  \texttt{expr.method(args...)}
  \]

  Examples:
  
  \[
  \texttt{mylist[3]} = \texttt{myvalue} \\
  \texttt{mylist.append(myvalue)}
  \]

  Changes something about the object that \texttt{mylist} refers to
New and old values

• Every **expression** evaluates to a value
  – It might be a new value
  – It might be a value that already exists

• A **constructor** evaluates to a **new** value:
  
  \[ [3, 1, 4, 1, 5, 9] \]
  
  \[ [3, 1, 4] + [1, 5, 9] \]
  
  `mylist = [[3, 1], [4, 1]]` # right hand side is a constructor

• An **access** expression evaluates to an **existing** value:
  
  `mylist[1]`

• What does a function call evaluate to?
An aside: List notation

- Possibly misleading notation:

- More accurate, but more verbose, notation:
Object identity

• An object’s identity never changes
• Can think of it as its address in memory
• Its value of the object (the thing it represents) may change

```python
mylist = [1, 2, 3]
otherlist = mylist
mylist.append(4)

mylist is otherlist ⇒ True
    mylist and otherlist refer to the exact same object

mylist == [1, 2, 3, 4] ⇒ True
    The object mylist refers to is equal to the object [1,2,3,4]
    (but they are two different objects)

mylist is [1, 2, 3, 4] ⇒ False
    The object mylist refers to is not the exact same object
    as the object [1,2,3,4]
```

The object identity test “is” is rarely used
Object type and variable type

• An **object’s** type never changes

• A **variable** can get rebound to a value of a different type

  Example: The variable `a` can be bound to an int or a list
  
  ```python
a = 5  # 5 is always an int
a = [1, 2, 3, 4]  # [1, 2, 3, 4] is always a list
  ```

• A **type** indicates:
  – what operations are allowed
  – the set of representable values
  – `type(object)` returns the type of an object
New datatype: tuple

A tuple represents an ordered sequence of values

Example:

tuple

```
"four"  "score"  "and"  "seven"  "years"
```

```
↓  ↓  ↓  ↓  ↓
"four"  "score"  "and"  "seven"  "years"
```
Tuple operations

Constructors

– Literals: Just like lists, but round the square brackets
  ("four", "score", "and", "seven", "years")
– Also \((3, 1) + (4, 1) \Rightarrow (3, 1, 4, 1)\), etc.

Queries

– Just like lists

Mutators

– None!
Immutable datatype

• An immutable datatype is one that doesn’t have any functions in the third category:
  – Constructors
  – Queries
  – Mutators: None!

• Immutable datatypes:
  – int, float, boolean, string, function, tuple, frozenset

• Mutable datatypes:
  – list, dictionary, set
Not every value may be placed in a set

- Set *elements* must be **immutable** values
  - `int`, `float`, `bool`, `string`, `tuple`
  - *not*: `list`, `set`, `dictionary`

- The set itself is **mutable** (e.g. we can add and remove elements)

- **Goal**: only set operations change the set
  - After “`myset.add(x)`”, `x in myset` ⇒ True
  - `y in myset` always evaluates to the same value
  
  Both conditions should hold until `myset` is changed

- **Mutable elements can violate these goals**

- **Aside**: `frozenset` must contain immutable values and is itself immutable (cannot add and remove elements)
Not every value is allowed to be a key in a dictionary

• Keys must be **immutable** values
  – int, float, bool, string, *tuple of immutable types*
  – *not*: list, set, dictionary

• The dictionary itself is **mutable** (e.g. we can add and remove elements)

• **Goal**: only dictionary operations change the keyset
  – after “`mydict[x] = y`”, `mydict[x] ⇒ y`
  – if `a == b`, then `mydict[a] == mydict[b]`

These conditions should hold until `mydict` is changed

• **Mutable keys can violate these goals**
Python’s *Data Model*

- All data is represented by *objects*
- Each object has:
  - an *identity*
    - Never changes
    - Think of this as address in memory
    - Test with `is` (but you rarely need to do so)
  - a *type*
    - Never changes
  - a *value*
    - Can change for *mutable* objects
    - Cannot change for *immutable* objects
    - Test with `==`
Mutable and Immutable Types

- **Immutable**
  - numbers, strings, tuples
- **Mutable**
  - Lists, sets and dictionaries

Note: a set is mutable, but a `frozenset` is immutable
Tuples are immutable
Lists are mutable

def updaterecord(record, position, value):
    """change the value at the given position""
    record[position] = value

mylist = [1,2,3]
mytuple = (1,2,3)
updaterecord(mylist, 1, 10)
print mylist
updaterecord(mytuple, 1, 10)
print mytuple
Mutable and Immutable Types

>>> def increment(uniquewords, word):
...     """increment the count for word""
...     if uniquewords.has_key(word):
...         uniquewords[word] = uniquewords[word] + 1
...     else:
...         uniquewords[word] = 1

>>> mywords = dict()
>>> increment(mywords, "school")
>>> print mywords
{'school': 1}

>>> def increment(value):
...     """increment the value???""
...     value = value + 1
>>> myval = 5
>>> increment(myval)
>>> print myval
5