Sharing, mutability, and immutability

Rob Thompson
UW CSE 160
Winter 2021
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)
```

See in python tutor
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:
  
  ```python
  myvar = expr
  size = 6
  list2 = list1
  ```

• Mutating (changing) an **object** does not change any **variable** binding
  Two syntaxes:
  ```python
  left_expr = right_expr
  expr.method(args...)
  ```

  Examples:
  ```python
  mylist[3] = myvalue
  mylist.append(myvalue)
  ```

Changes what the variables `size` and `list2` are bound to

Changes something about the object that `mylist` refers to
Example: Variable reassignment or Object mutation?

```python
def no_change(lst):
    """does NOT modify what lst refers to, instead re-binds lst""
    lst = lst + [99]

def change_val(lst):
    """modifies object lst refers to""
    lst[0] = 13

def append_val(lst):
    """modifies object lst refers to""
    lst.append(99)

lst2 = [1, 2]
no_change(lst2)
change_val(lst2)
append_val(lst2)
```
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:

  \[
  [3, 1, 4, 1, 5, 9] \\
  [3, 1, 4] + [1, 5, 9] \\
  \text{mylist} = [[[3, 1], [4, 1]]]
  \]

• An access expression evaluates to an existing value:

  \text{mylist}[1]

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

• Possibly misleading notation:

  list
  │ “four” │ “score” │ “and” │ “seven” │ “years” │

• More accurate, but more verbose, notation:

  list
  ↓ ↓ ↓ ↓ ↓
  “four” “score” “and” “seven” “years”
Aside: 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]
```

**Moral:** Use `==` to check for equality, NOT `is`
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

```
  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"
```

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

Constructors
- Literals: Use parentheses
  ("four", "score", "and", "seven", "years")
  (3, 1) + (4, 1) => (3, 1, 4, 1)

Queries
- Just like lists:
  ```python
tup = ("four", "score", "and", "seven", "years")
print(tup[0]) => "four"
print(tup[-1]) => "years"
```

Mutators
- None!
Immutability

- 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
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)
def increment(uniquewords, word):
    """increment the count for word""
    if word in uniquewords:
        uniquewords[word] = uniquewords[word] + 1
    else:
        uniquewords[word] = 1
mywords = dict()
increment(mywords, "school")
print(mywords)
def increment(value):
    """increment the value??""
    value = value + 1
myval = 5
increment(myval)
print(myval)
Increment Example (cont.)

```python
>>> def increment(uniquewords, word):
...     """increment the count for word""
...     if word in uniquewords:
...         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
```
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 `==`
Remember:
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)

  • **Aside**: **frozenset** must contain immutable values and is itself immutable
    (cannot add and remove elements)
Remember: 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)