Lecture 7

Abstract data types
Mutable Values

Some slides were borrowed from CS88, Berkeley

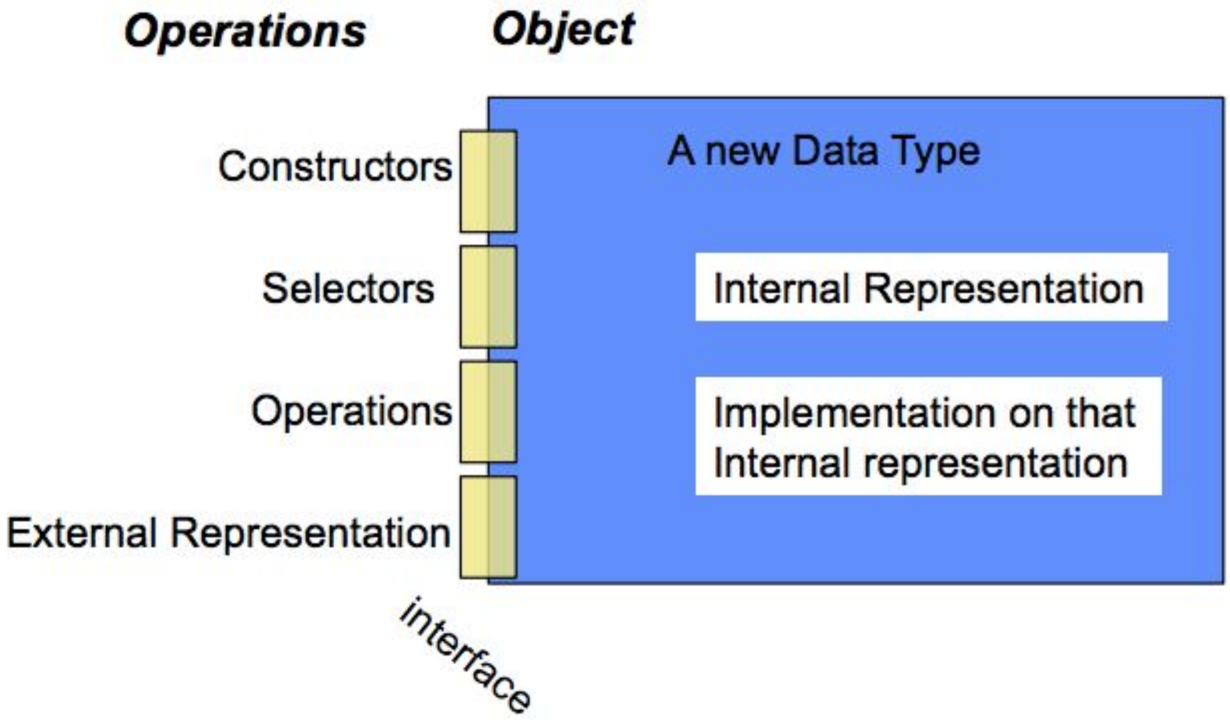
Abstract Data Types

Slides were borrowed from cs88, berkeley.

Data abstraction

- Compound values combine other values together
  - A date: a year, a month, and a day
  - A geographic position: latitude and longitude
- Data abstraction lets us manipulate compound values as *units*
- Isolate two parts of any program that uses data:
  - How data are represented (as parts)
  - How data are manipulated (as units)
- **Data abstraction**: A methodology by which functions enforce an abstraction barrier between representation and use

Abstract Data Type



Operations

Object

Constructors

Selectors

Operations

External Representation

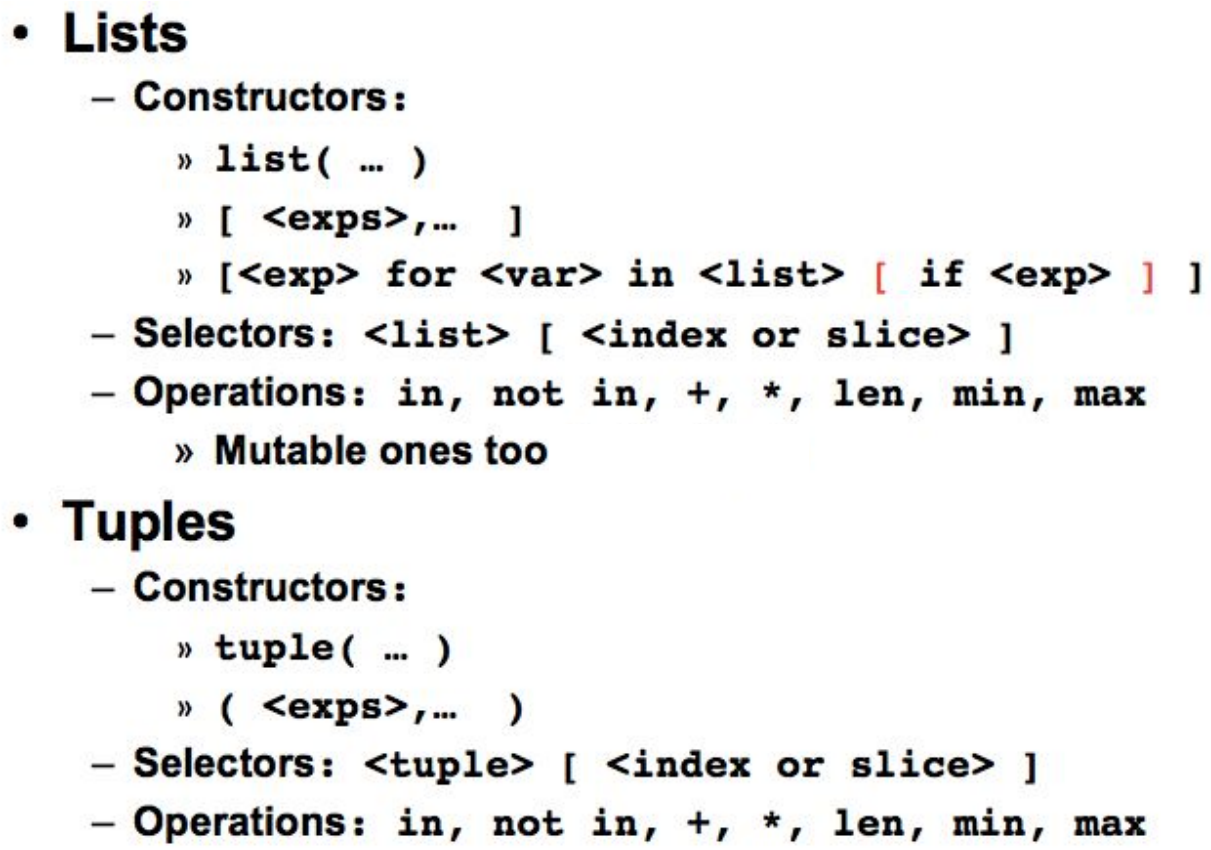
interface

A new Data Type

Internal Representation

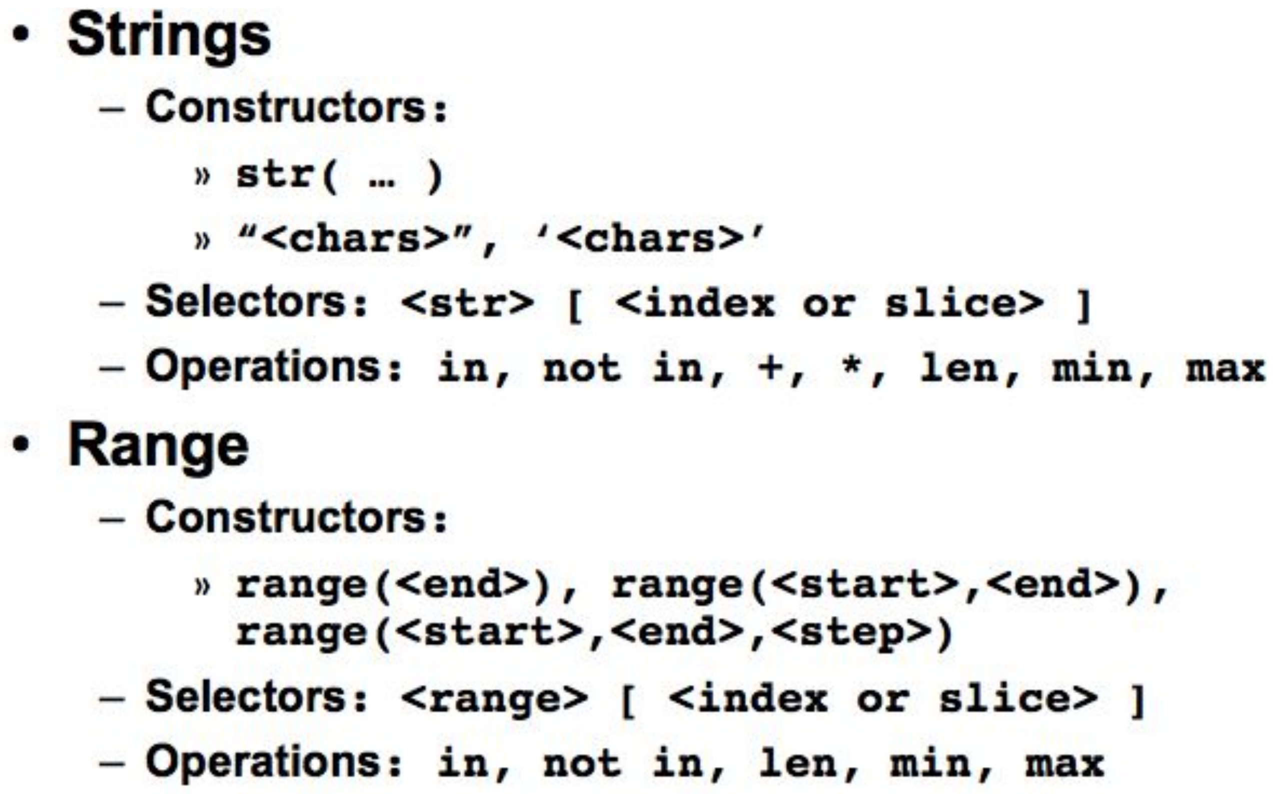
Implementation on that Internal representation

Data Types You have seen



- **Lists**
  - Constructors:
    - » `list( … )`
    - » `[ <exps>,…  ]`
    - » `[<exp> for <var> in <list> [ if <exp> ] ]`
  - Selectors: `<list> [ <index or slice> ]`
  - Operations: `in, not in, +, *, len, min, max`
    - » Mutable ones too
- **Tuples**
  - Constructors:
    - » `tuple( … )`
    - » `( <exps>,…  )`
  - Selectors: `<tuple> [ <index or slice> ]`
  - Operations: `in, not in, +, *, len, min, max`

More “Built-in” Examples



- **Strings**
  - Constructors:
    - » `str( … )`
    - » `“<chars>”, ‘<chars>’`
  - Selectors: `<str> [ <index or slice> ]`
  - Operations: `in, not in, +, *, len, min, max`
- **Range**
  - Constructors:
    - » `range(<end>), range(<start>,<end>), range(<start>,<end>,<step>)`
  - Selectors: `<range> [ <index or slice> ]`
  - Operations: `in, not in, len, min, max`

(Demo)

Abstraction Barriers

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat
Implement selectors and constructor for cities	three-element lists	list literals and element selection

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat
Implement selectors and constructor for cities	three-element lists	list literals and element selection

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat
Implement selectors and constructor for cities	three-element lists	list literals and element selection

Implementation of lists

Abstraction Barriers

Parts of the program that…	Treat city as	Using ...
Use cities to perform computation	whole data values	distance, closer_city
Create cities or implement cities operations	combination of 3 elements	make_city, get_lon, get_lat
Implement selectors and constructor for cities	three-element lists	list literals and element selection

Implementation of lists

Violating Abstraction Barriers

```
closer_city(34, 67,[‘Moscow’, 45, 43], [‘Paris’,65, 78])
```

```
def distance(x, y):
    return sqrt((x[1] - y[1])** 2 + (x[2] - y[2])**2))
```

How many violations can you spot?

A: 0 B: 1 C: 2 D: 3 E: 4

Violating Abstraction Barriers

```
closer_city(34, 67,[‘Moscow’, 45, 43], [‘Paris’, 65, 78])
```

Does not use constructors

Does not use constructors

```
def distance(x, y):
    return sqrt((x[1] - y[1])** 2 + (x[2] - y[2])**2))
```

How many violations can you spot?

A: 0 B: 1 C: 2 D: 3 E: 4

Violating Abstraction Barriers

```
closer_city(34, 67,[‘Moscow’, 45, 43], [‘Paris’, 65, 78])
```

Does not use constructors

Does not use constructors

```
def distance(x, y):
    return sqrt((x[1] - y[1])** 2 + (x[2] - y[2])**2))
```

No selectors!

No selectors!

How many violations can you spot?

A: 0 B: 1 C: 2 D: 3 E: 4

Violating Abstraction Barriers

```
closer_city(34
```

```
ris', 65, 78])
```

Does not use constructors

```
def distance(x
    return sqrt
y[2])**2))
```

No selectors!

How many violation

A: 0 B: 1

Violating Abstraction Barriers

```
(Thank you, John. It was fun :))
```

Time for fun!

Check point

```
lst = [1, -2, -3, 4, 5]
def func1(x):
    return x < 2

it = filter(func1, lst)
print(list(it))
```

What is the output of the code shown?

A: [1, 4, 5]

B: Error

C: [-2, -3]

D: [1, -2, -3, None, None]

E: None of the above

Check point

```
lst = [1, -2, -3, 4, 5]

def func1(x):
    return x < -1

it = map(func1, lst)
print(list(it))
```

What is the output of the code shown?

A: [False, False, False, False, False]

B: [False, True, True, False, False]

C: [True, False, False, True, True]

D: [True, True, True, True, True]

E: None of the above

Questions from last week video lecture

```
d = { (1,2):1, (2,3):2 }

print(d[1,2])
```

What Will Be The Output

A: KeyError

B: 1

C: {(2,3):2}

D: {(1,2):1}

E: None of the above

Question based on last week video lecture

```python
basket = {}

def addone(index):
    if index in basket:
        basket[index] += 1
    else:
        basket[index] = 1

addone('Apple')
addone('Banana')
addone('apple')
addone('Apple')
print (len(basket))
```

What Will Be The Output

A: 1

B: 2

C: 3

D: 4

E: None of the above

Question based on last week video lecture

```
def problem(lst, index):
    try:
        average = sum(lst)/len(lst)
        last_elem = lst[index]
    except IndexError as e:
        print('Index is wrong')
    except ZeroDivisionError as e:
        print("Can't divide by a 0")

    print("I'm safe")

problem( [1, 2, 3], 3)
```

A: Index is wrong

B: Can't divide by a 0

C: Can't divide by a 0
I’m safe

D: Index is wrong
I’m safe

E: None of the above

Question based on last week video lecture

```
def problem(lst, index):
    try:
        average = sum(lst)/len(lst)
        last_elem = lst[index]
    except IndexError as e:
        print('Index is wrong')
    except ZeroDivisionError as e:
        print("Can't divide by a 0")

    print("I'm safe")

problem ( [], 17)
```

A: Index is wrong

B: Can't divide by a 0

C: Can't divide by a 0
I’m safe

D: Index is wrong
I’m safe

E: None of the above

Question

- Is there a way in Python to call filter on a list where the filtering function has a *number of formal parameters* **bound** during the call ?

```
def func (a, b, c):
    return a + b < c
```

```
lst = [10, 20, 30, 40]
filter(func(a=10, c=35), lst) #Want to happen
```

```
def make_filter(a, c):
   def my_filter(b):
       return a + b < c
   return my_filter

filt = make_filter(10, 35)
lst = [10, 20, 30 , 40]

list(filter(filt, lst))
```

```
def func (a, b, c):
    return a + b < c

lst = [10, 20, 30, 40]

list(filter(lambda x: func(10, x, 35), lst))
```

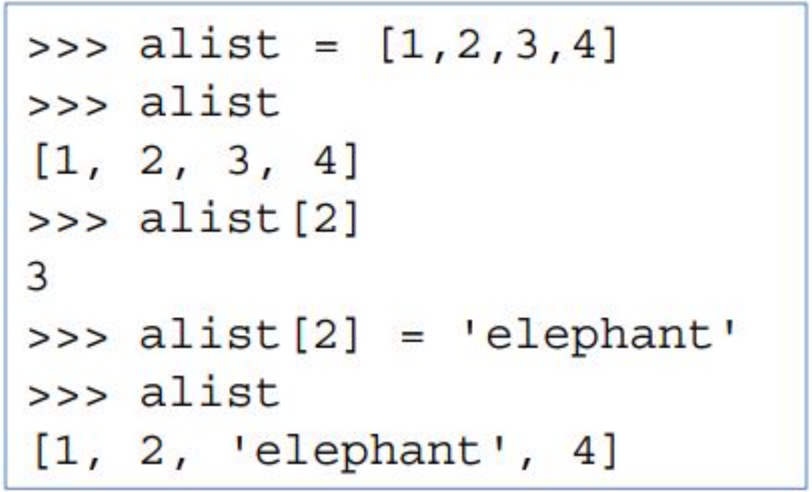
Mutable Data

Mutability

- Immutable – the value of the object cannot be changed:
  - integers, floats, booleans
  - strings, tuples
- Mutable – the value of the object can be changed:
  - Lists
  - Dictionaries

Mutability

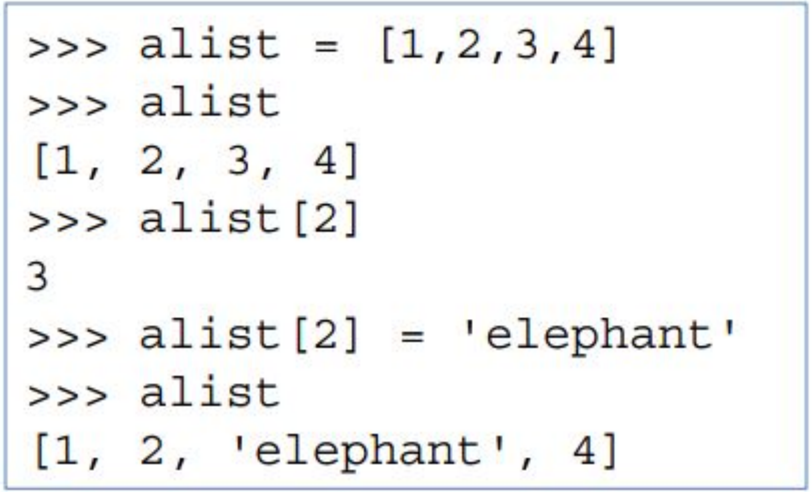
- Immutable – the value of the object cannot be changed:
  - integers, floats, booleans
  - strings, tuples
- Mutable – the value of the object can be changed:
  - Lists
  - Dictionaries



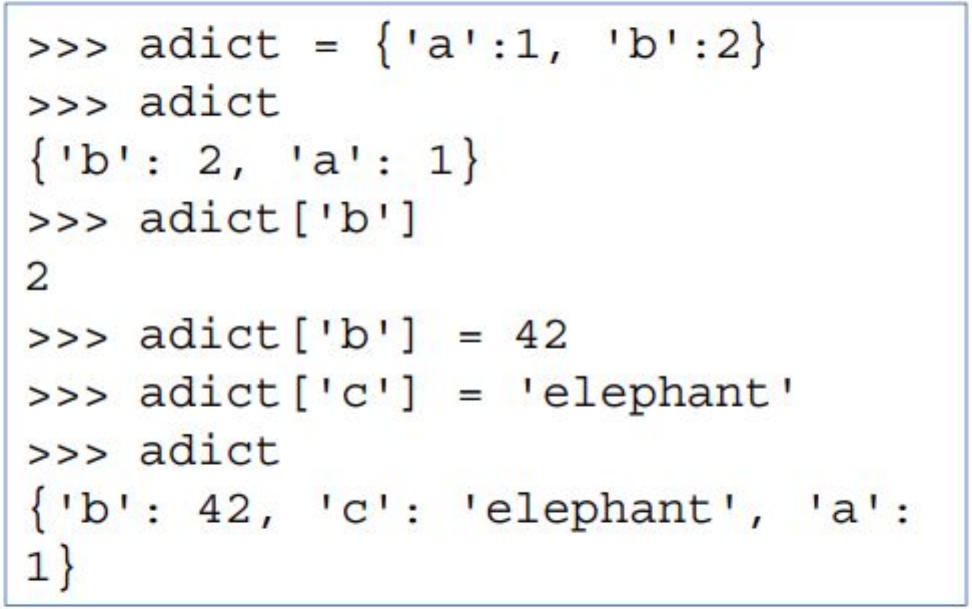
```
>>> alist = [1,2,3,4]
>>> alist
[1, 2, 3, 4]
>>> alist[2]
3
>>> alist[2] = 'elephant'
>>> alist
[1, 2, 'elephant', 4]
```

Mutability

- Immutable – the value of the object cannot be changed:
  - integers, floats, booleans
  - strings, tuples
- Mutable – the value of the object can be changed:
  - Lists
  - Dictionaries



```
>>> alist = [1,2,3,4]
>>> alist
[1, 2, 3, 4]
>>> alist[2]
3
>>> alist[2] = 'elephant'
>>> alist
[1, 2, 'elephant', 4]
```



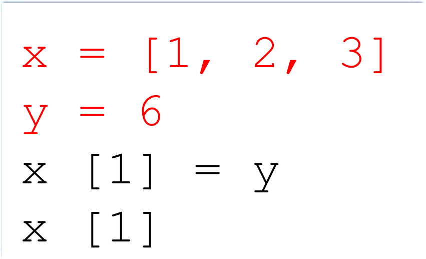
```
>>> adict = {'a':1, 'b':2}
>>> adict
{'b': 2, 'a': 1}
>>> adict['b']
2
>>> adict['b'] = 42
>>> adict['c'] = 'elephant'
>>> adict
{'b': 42, 'c': 'elephant', 'a':
1}
```

From value to storage …

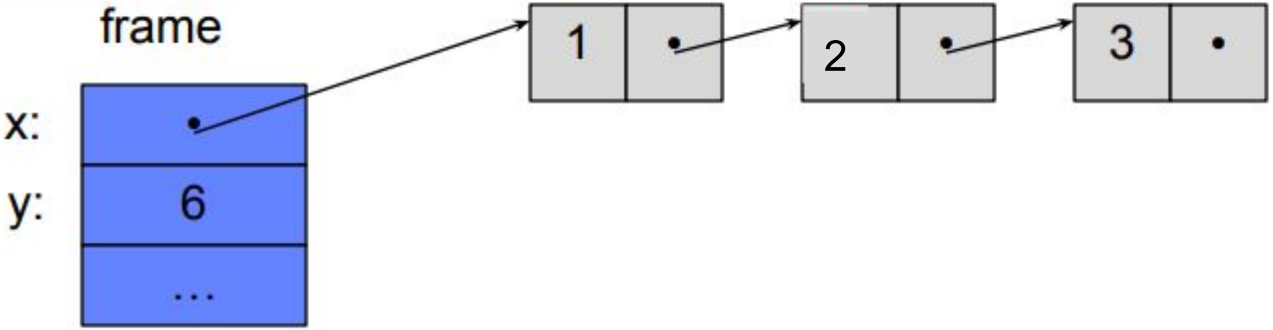
- A variable assigned a compound value (object) is a **reference** to that object.
- Mutable object can be *changed* but the variable(s) still refer to it

From value to storage ...

- A variable assigned a compound value (object) is a **reference** to that object.
- Mutable object can be *changed* but the variable(s) still refer to it

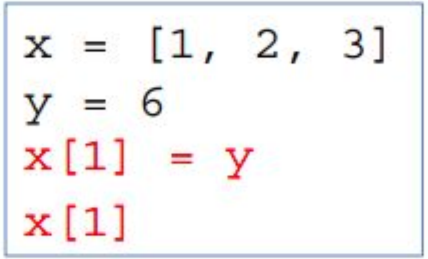


```
x = [1, 2, 3]
y = 6
x [1] = y
x [1]
```

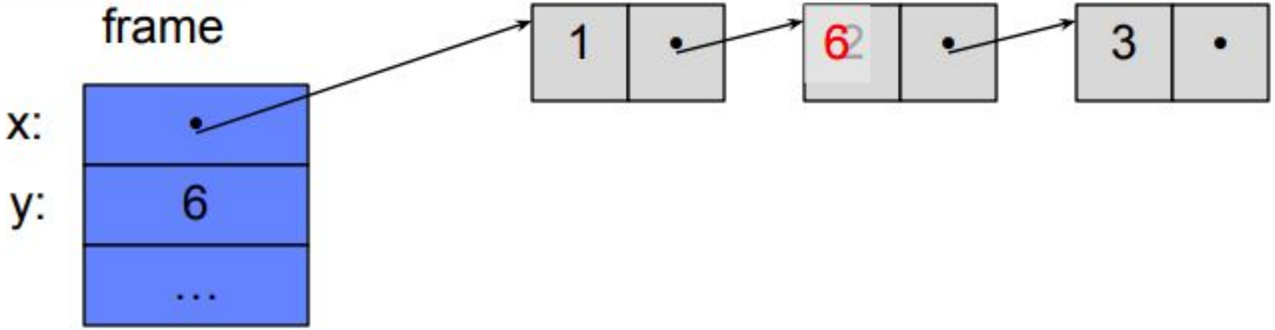


From value to storage ...

- A variable assigned a compound value (object) is a **reference** to that object.
- Mutable object can be *changed* but the variable(s) still refer to it



```
x = [1, 2, 3]
y = 6
x[1] = y
x[1]
```



Examples

```
x = [1, 2 , 3]
y = x
print (y)
x[1] = 11
print (y)
```

```
x = [1, 2]
y = [x, x, x]
print (y)
x[1] = 3
print (y)
y[2] = [1, 3]
print (y[0] == y[2])
print (y[0] is y[2])
```

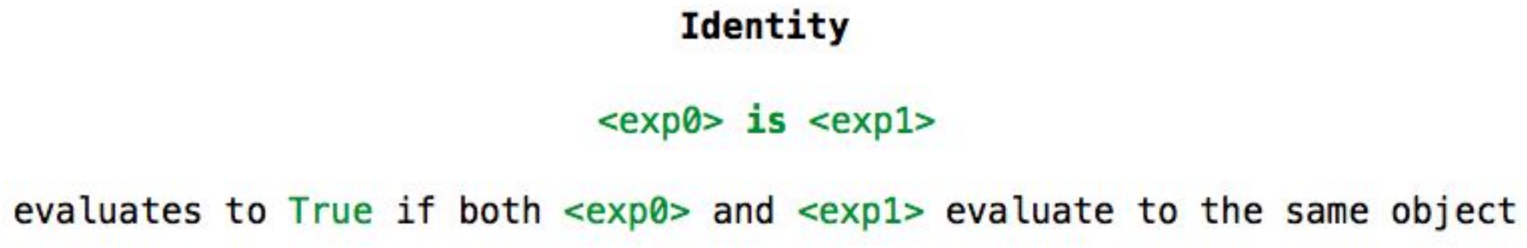
Copies, ‘is’ and ‘==‘

```
 >>> alist = [1, 2, 3, 4]
 >>> alist == [1, 2, 3, 4]  # Equal values?

 True

 >>> alist is [1, 2, 3, 4] # same object?

  False
>>> blist = alist          # assignment refers
>>> alist is blist         # to same object. Shallow copy

 True

 >>> blist = list(alist)   # type constructors copy
 >>> blist is alist         # Deep copy

 False
```

Copies, ‘is’ and ‘==‘

```
>>> alist = [1, 2, 3, 4]
>>> alist == [1, 2, 3, 4]  # Equal values?

True

>>> blist = alist[ : ]     # so does slicing
>>> blist is alist

False

>>> blist
[1, 2, 3, 4]
>>>
```

Identity Operators: is is not

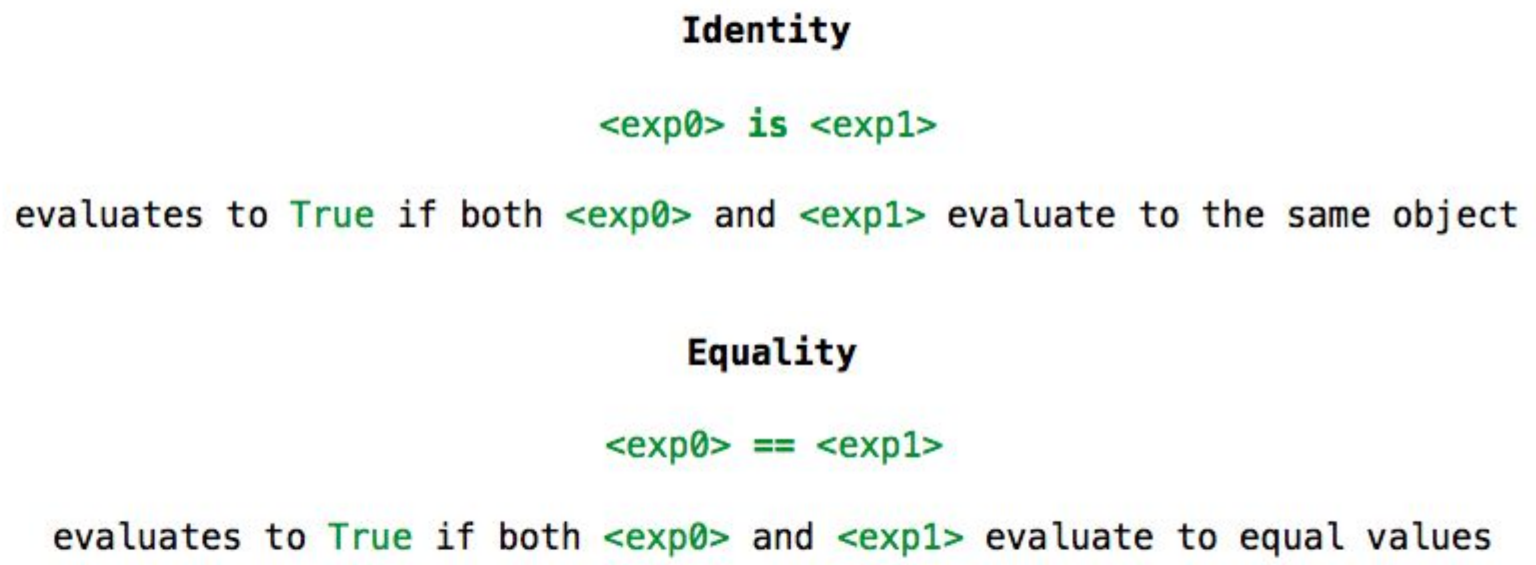


Identity

```
<exp0> is <exp1>
```

evaluates to True if both <exp0> and <exp1> evaluate to the same object

Identity Operators



Identity

`<exp0> is <exp1>`

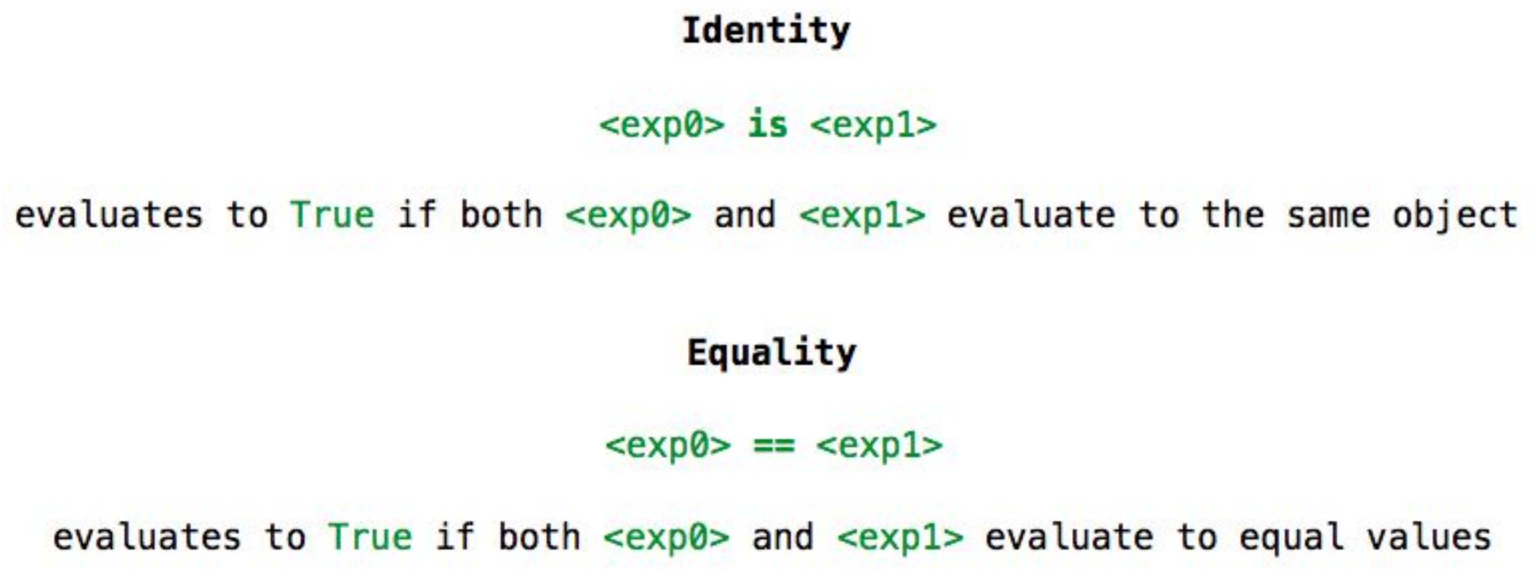
evaluates to True if both <exp0> and <exp1> evaluate to the same object

Equality

`<exp0> == <exp1>`

evaluates to True if both <exp0> and <exp1> evaluate to equal values

Identity Operators



Identity

`<exp0> is <exp1>`

evaluates to True if both <exp0> and <exp1> evaluate to the same object

Equality

`<exp0> == <exp1>`

evaluates to True if both <exp0> and <exp1> evaluate to equal values

Identical objects are always equal values

Parameter passing: Output?

```
def test (x):
    x = x + 1
y = 10
test(y)
print(y)
```

A: 10

B: 11

C: None

D: Error

E: I do not know

Parameter passing: Output?

```
def test (x):
    x[0] = x[0] + 1
y = [1, 2, 3]
test(y)
print(y)
```

A: [1, 2, 3]

B: [2, 2, 3]

C: None

D: Error

E: I do not know