Mutable Values

Announcements

Objects

(Demo)

Objects

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes
- A type of object is called a class; **classes** are first-class values in Python
- Object-oriented programming:
  - A metaphor for organizing large programs
  - Special syntax that can improve the composition of programs
- In Python, every value is an object
  - All **objects** have **attributes**
  - A lot of data manipulation happens through object **methods**
  - Functions do one thing; objects do many related things

Example: Strings

(Demo)

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

"Bell" (\a)   ASCII Code Chart   "Line feed" (\n)

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0 0 0	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0 0 1	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1 0	2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0 1 1	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
1 0 0	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1 0 1	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
1 1 0	6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
1 1 1	7	p	q	r	s	t	u	v	w	x	y	z	{			}	~	DEL

3 bits → rows ; 8 bits

16 columns: 4 bits

- Layout was chosen to support sorting by character code
- Rows indexed 2–5 are a useful 6-bit (64 element) subset
- Control characters were designed for transmission

(Demo)

Representing Strings: the Unicode Standard

- 137,994 characters in Unicode 12.1
- 150 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

http://ian-albert.com/unicode_chart/unichart-chinese.jpg

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

(Demo)

Mutation Operations

[Demo]

First example in the course of an object changing state

The same object can change in value throughout the course of computation

```
jessica ———→ 👵
same_person ———→
OLDER
WOMAN          Unicode
               character
               name
```

All names that refer to the same object are affected by a mutation

Only objects of *mutable* types can change: lists & dictionaries

{Demo}

A function can change the value of any object in its scope

```
>>> four = [1, 2, 3, 4]          def mystery(s):  or  def mystery(s):
>>> len(four)                        s.pop()              s[2:] = []
4                                    s.pop()
>>> mystery(four)
>>> len(four)
2
```

```
>>> four = [1, 2, 3, 4]          def another_mystery():
>>> len(four)                        four.pop()
4                                    four.pop()
>>> another_mystery() # No arguments!
>>> len(four)
2
```

Tuples

(Demo)

Immutable values are protected from mutation

```
>>> turtle = (1, 2, 3)           >>> turtle = [1, 2, 3]
>>> ooze()  [Next lecture: ooze can   >>> ooze()
>>> turtle  [change turtle's binding]  >>> turtle
(1, 2, 3)                        ['Anything could be inside!']
```

The value of an expression can change because of changes in names or objects

```
                >>> x = 2                          >>> x = [1, 2]
                >>> x + x                          >>> x + x
                4                                  [1, 2, 1, 2]
Name change:                    Object mutation:   >>> x.append(3)
                >>> x = 3                          >>> x + x
                >>> x + x                          [1, 2, 3, 1, 2, 3]
                6
```

An immutable sequence may still change if it *contains* a mutable value as an element

```
>>> s = ([1, 2], 3)              >>> s = ([1, 2], 3)
>>> s[0] = 4                     >>> s[0][0] = 4
ERROR                            >>> s
                                 ([4, 2], 3)
```

Mutation

• As long as we never modify objects, a compound object is just the totality of its pieces

• A rational number is just its numerator and denominator

• This view is no longer valid in the presence of change

• A compound data object has an "identity" in addition to the pieces of which it is composed

• A list is still "the same" list even if we change its contents

• Conversely, we could have two lists that happen to have the same contents, but are different

```
>>> a = [10]                     >>> a = [10]
>>> b = a                        >>> b = [10]
>>> a == b                       >>> a == b
True                             True
>>> a.append(20)                 >>> b.append(20)
>>> a                            >>> a
[10, 20]                         [10]
>>> b                            >>> b
[10, 20]                         [10, 20]
>>> a == b                       >>> a == b
True                             False
```

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
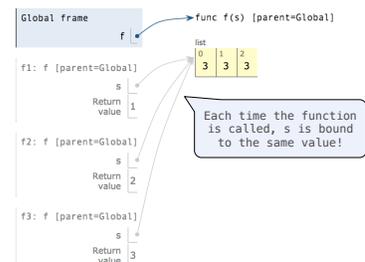
Identical objects are always equal values

(Demo)

A default argument value is part of a function value, not generated by a call

```
>>> def f(s=[]):
...     s.append(3)
...     return len(s)
...
>>> f()
1
>>> f()
2
>>> f()
3
```
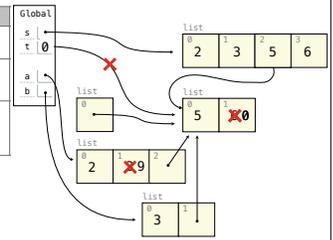
```
Global frame                   func f(s) [parent=Global]
                    f

                               list
                               0   1   2
f1: f [parent=Global]          3   3   3
                    s
           Return
           value    1

                               Each time the function
>>> f()                        is called, s is bound
                               to the same value!
f2: f [parent=Global]
                    s
           Return
           value    2

f3: f [parent=Global]
                    s
           Return
           value    3
```

Lists

Lists in Environment Diagrams

Assume that before each example below we execute:
```
s = [2, 3]
t = [5, 6]
```
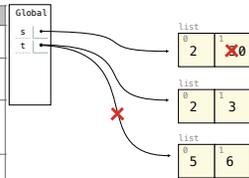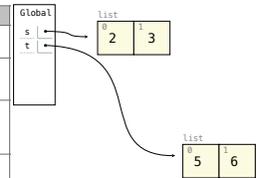
Operation	Example	Result
append adds one element to a list	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
extend adds all elements in one list to another list	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
addition & **slicing** create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	a → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]

Lists in Environment Diagrams

Assume that before each example below we execute:
```
s = [2, 3]
t = [5, 6]
```

Operation	Example	Result
append adds one element to a list	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
extend adds all elements in one list to another list	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
addition & **slicing** create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]
The **list** function also creates a new list containing existing elements	t = list(s) s[1] = 0	s → [2, 0] t → [2, 3]

Lists in Environment Diagrams

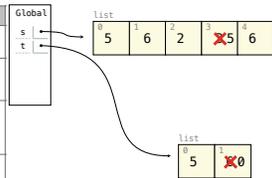**Assume that before each example below we execute:**
```
s = [2, 3]
t = [5, 6]
```

Operation	Example	Result
append adds one element to a list	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
extend adds all elements in one list to another list	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
addition & **slicing** create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]
The **list** function also creates a new list containing existing elements	t = list(s) s[1] = 0	s → [2, 0] t → [2, 3]
slice assignment replaces a slice with new values	s[0:0] = t s[3:] = t t[1] = 0	

Lists in Environment Diagrams

Assume that before each example below we execute:
```
s = [2, 3]
t = [5, 6]
```

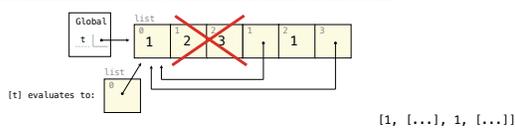| Operation	Example	Result
append adds one element to a list	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
extend adds all elements in one list to another list	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
addition & **slicing** create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	s → [2, 3] t → [5, 0] a → [2, 9, [5, 0]] b → [3, [5, 0]]
The **list** function also creates a new list containing existing elements	t = list(s) s[1] = 0	s → [2, 0] t → [2, 3]
slice assignment replaces a slice with new values	s[0:0] = t s[3:] = t t[1] = 0	s → [5, 6, 2, 5, 6] t → [5, 0]

Lists in Environment Diagrams

Assume that before each example below we execute:
```
s = [2, 3]
t = [5, 6]
```

Operation	Example	Result
pop removes & returns the last element	t = s.pop()	s → [2] t → 3
remove removes the first element equal to the argument	t.extend(t) t.remove(5)	s → [2, 3] t → [6, 5, 6]
slice assignment can remove elements from a list by assigning [] to a slice.	s[:1] = [] t[0:2] = []	s → [3] t → []

Lists in Lists in Lists in Environment Diagrams

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```

[t] evaluates to:

```
[1, [...], 1, [...]]
```

```
t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```

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