

# Mutability

---

# 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

**ASCII Code Chart**

"Bell" (\a)
"Line feed" (\n)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0 0 0	0 NUL	1 SOH	2 STX	3 ETX	4 EOT	5 ENQ	6 ACK	7 BEL	8 BS	9 HT	A LF	B VT	C FF	D CR	E SO	F SI	
0 0 1	1 DLE	2 DC1	3 DC2	4 DC3	5 DC4	6 NAK	7 SYN	8 ETB	9 CAN	A EM	B SUB	C ESC	D FS	E GS	F 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

8 rows: 3 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

聾	聾	聾	聽	聵	聶	職	聾
8071	8072	8073	8074	8075	8076	8077	8078
健	腓	腳	腓	腓	腓	腓	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艷	艷	艷	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
菘	菘	荳	菘	菘	菘	荷	菘
8371	8372	8373	8374	8375	8376	8377	8378
葱	菘	葳	葳	葵	葶	葶	蔥

[http://ian-albert.com/unicode\\_chart/unichart-chinese.jpg](http://ian-albert.com/unicode_chart/unichart-chinese.jpg)

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE



(Demo)

## Mutation Operations



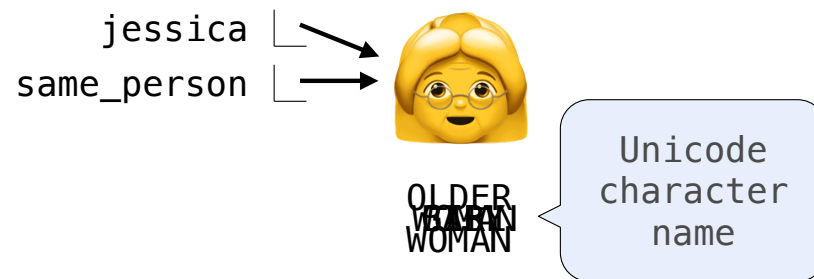
## Some Objects Can Change

---

[Demo]

First example in the course of an object changing state

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



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

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

{Demo}

## Mutation Can Happen Within a Function Call

---

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

```
>>> four = [1, 2, 3, 4]
>>> len(four)
4
>>> mystery(four)
>>> len(four)
2
```

```
def mystery(s):      or      def mystery(s):
    s.pop()           s[2:] = []
    s.pop()
```

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

```
def another_mystery():
    four.pop()
    four.pop()
```

# Tuples

(Demo)

## Tuples are Immutable Sequences

---

Immutable values are protected from mutation

```
>>> turtle = (1, 2, 3)
>>> ooze()
>>> turtle
(1, 2, 3)
```

Next lecture: ooze can change turtle's binding

```
>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
['Anything could be inside!']
```

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

**Name change:**

```
>>> x = 2
>>> x + x
4
>>> x = 3
>>> x + x
6
```

**Object mutation:**

```
>>> x = [1, 2]
>>> x + x
[1, 2, 1, 2]
>>> x.append(3)
>>> x + x
[1, 2, 3, 1, 2, 3]
```

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

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

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

Mutation

## Sameness and Change

---

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

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

## 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**

(Demo)





## Mutable Functions

## A Function with Behavior That Varies Over Time

Let's model a bank account that has a balance of \$100

```
>>> withdraw = make_withdraw_list(100)
```

In a (mutable) list referenced in the parent frame of the function

Return value:  
remaining balance

```
>>> withdraw(25)  
75
```

Argument:  
amount to withdraw

Different  
return value!

```
>>> withdraw(25)  
50
```

Second withdrawal of  
the same amount

```
>>> withdraw(60)  
'Insufficient funds'
```

```
>>> withdraw(15)  
35
```

Where's this balance  
stored?

## Mutable Values & Persistent Local State

