# 61A Lecture 14

Friday, October 3

Announcements	

•Homework 4 due Tuesday 10/7 @ 11:59pm (It is small)

- •Homework 4 due Tuesday 10/7 @ 11:59pm (It is small)
- •Project 2 due Thursday 10/9 @ 11:59pm (It is BIG)

- •Homework 4 due Tuesday 10/7 @ 11:59pm (It is small)
- •Project 2 due Thursday 10/9 @ 11:59pm (It is BIG)
  - Project Party Monday 5pm-7pm in 271, 273, & 275 Soda

- Homework 4 due Tuesday 10/7 @ 11:59pm (It is small)
- •Project 2 due Thursday 10/9 @ 11:59pm (It is BIG)
  - ■Project Party Monday 5pm-7pm in 271, 273, & 275 Soda
  - Extra credit point for submitting your project at least 24 hours before the deadline

# **Encoding Strings**

(Bonus Material)

#### American Standard Code for Information Interchange

#### **ASCII Code Chart** NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT SI DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS GS US # & 3 5 7 9 1 6 Ε F S U Χ Ζ d а С е h m

#### American Standard Code for Information Interchange

							Α	SCI	Coo	de Cl	nart						
	لـ	0	1	2	3	4	լ 5	<sub>I</sub> 6	7	8	9	Α	В	C	D	E	<u> </u>
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	Ŀ	VT	FF	CR	S0	SI
ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2		Ţ.	11	#	\$	%	&	-	(	)	*	+	,	-	•	/
Μ	3	0	1	2	3	4	5	6	7	8	9		;	٧	=	^	?
 S	4	@	Α	В	C	D	Е	F	G	Н	I	J	K	L	М	N	0
rows	5	Р	œ	R	S	T	<b>–</b>	٧	W	Х	Υ	Z	[	\	]	^	_
	6	`	а	b	U	d	е	f	g	h	i	j	k	l	m	n	0
∞	7	р	q	r	s	t	u	٧	W	Х	у	Z	{		}	1	DEL

#### American Standard Code for Information Interchange

							A	SCII	Coc	de Cl	nart						
	١	0	1	2	3	4	5	ا 6	7	8	9	Α	В	С	D	E	<u> </u>
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
ts	1]	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2		-:		#	\$	%	&	-	(	)	*	+	,	-	•	/
Μ	3	0	1	2	з	4	5	6	7	8	9		;	٧	II	۸	?
 S	4	@	Α	В	U	D	Е	F	G	H	Ι	ין	K	L	М	N	0
rows	5	Р	œ	R	S	T	5	٧	W	Х	Υ	Z	[	/	]	<	
	6	,	а	р	U	d	e	f	g	h	i	j	k	7	m	n	0
$\infty$	<u></u>	р	q	r	S	t	u	V	W	Х	у	z	{		}	~	DEL

16 columns: 4 bits

4

### American Standard Code for Information Interchange

							A	SCII	Coc	de Cl	nart						
	١	0	1	2	3	4	5	ا 6	7	8	9	Α	В	С	D	E	<u> </u>
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
ts	1]	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2		-:		#	\$	%	&	-	(	)	*	+	,	-	•	/
Μ	3	0	1	2	з	4	5	6	7	8	9		;	٧	II	۸	?
 S	4	@	Α	В	U	D	Е	F	G	H	Ι	ין	K	L	М	N	0
rows	5	Р	œ	R	S	T	5	٧	W	Х	Υ	Z	[	/	]	<	
	6	,	а	р	U	d	e	f	g	h	i	j	k	7	m	n	0
$\infty$	<u></u>	р	q	r	S	t	u	V	W	Х	у	z	{		}	~	DEL

16 columns: 4 bits

• Layout was chosen to support sorting by character code

4

#### American Standard Code for Information Interchange

							A	SCI	Coc	de Cl	nart						
	١	0	1	2	3	4	5	<sub>I</sub> 6	7	8	9	Α	В	C	D	Ε	ı F ı
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
ts	1]	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2			=	#	\$	%	&	-	(	)	*	+	,	•	٠	/
Μ	3	0	1	2	3	4	5	6	7	8	9		;	٧	=	۸	?
 S	4]	@	Α	В	U	D	Е	F	G	Н	I	ין	K	L	М	N	0
rows	5	Р	Q	R	S	T	5	٧	W	Х	Υ	Z	[	\	]	<	_
	6	`	а	Ь	U	d	e	f	g	h	i	j	k	l	m	n	0
$\infty$	<u></u>	р	q	r	S	t	u	V	W	Х	у	z	{		}	ł	DEL

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

#### American Standard Code for Information Interchange

							A	SCII	Coc	de Cl	hart						
	لـ	0	1	2	3	4	5	ا 6	7	8	9	ΙA	В	С	D	Ε	<u> </u>
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	Ŀ	VT	FF	CR	SO	SI
ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2		-:		#	\$	%	&	-	(	)	*	+	,	•	٠	/
Μ	3	0	1	2	3	4	5	6	7	8	9		;	٧	II	۸	?
 S	4	@	Α	В	C	D	Е	F	G	Н	I	J	K	L	М	N	0
rows	5	Р	Q	R	S	Т	5	V	W	Х	Υ	Z	[	\	]	<	_
	6	`	а	р	U	d	e	f	g	h	i	j	k	٦	m	n	0
$\infty$	7	р	q	r	S	t	u	V	W	Х	у	z	{		}	1	DEL

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

#### American Standard Code for Information Interchange

							A	SCII	[ Co	de Cl	hart		"	Line	fee	ed"	(\n)
	١	0	1	2	3	4	<sub>1</sub> 5	6	7	8	9	_ A /,	В	С	D	ΙE	ı F ı
Ī	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
ts	ī	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bit	2		į.	ш	#	\$	%	&	1	(	)	*	+	,	-	•	/
m	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	^	?
 S	4	@	Α	В	С	D	E	F	G	Н	I	J	К	L	М	N	0
rows	5	Р	Q	R	S	Т	U	٧	W	Х	Υ	Z	[	\	]	^	_
	6	`	а	b	С	d	е	f	g	h	i	j	k	l	m	n	0
$\infty$	7	р	q	r	n	t	u	٧	W	Х	у	z	{		}	1	DEL

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

American Standard Code for Information Interchange

				"Be	ell"	(\a	) (4	SCI	[ Co	de Cl	hart		"	Line	fee	ed"	(\n)
		0 1	1	2	<sub> </sub> 3	<sub>I</sub> 4	5	6	7	8	9	L A	В	С	<sub>l</sub> D	Ε	ı F ı
Ţ	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
DICS	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
۲ 2	2		<u>.</u>	=	#	\$	%	&	-	(	)	*	+	,	-	•	/
<b>1</b>	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	4	@	Α	В	С	D	Ε	F	G	Н	I	J	К	L	М	N	0
n ≥ >	5	Р	Q	R	S	Т	U	V	W	Х	Υ	Z	[	\	]	^	_
	6	`	а	b	С	d	е	f	g	h	i	j	k	ι	m	n	0
7	7	р	q	r	s	t	u	٧	W	х	у	z	{		}	1	DEL

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

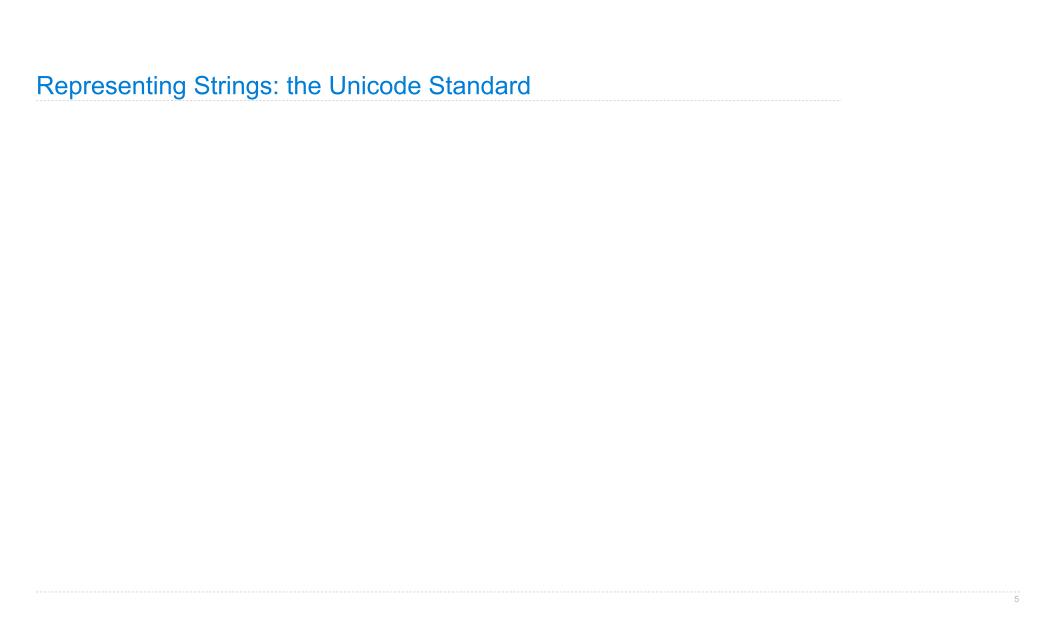
American Standard Code for Information Interchange

				"Be	ell"	(\a	) (4	SCI	[ Co	de Cl	hart		"	Line	fee	ed"	(\n)
		0 1	1	2	<sub> </sub> 3	<sub>I</sub> 4	5	6	7	8	9	L A	В	С	<sub>l</sub> D	Ε	ı F ı
Ţ	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
DICS	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
۲ 2	2		<u>.</u>	=	#	\$	%	&	-	(	)	*	+	,	-	•	/
<b>1</b>	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	4	@	Α	В	С	D	Ε	F	G	Н	I	J	К	L	М	N	0
n ≥ >	5	Р	Q	R	S	Т	U	V	W	Х	Υ	Z	[	\	]	^	_
	6	`	а	b	С	d	е	f	g	h	i	j	k	ι	m	n	0
7	7	р	q	r	s	t	u	٧	W	х	у	z	{		}	1	DEL

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
- $\bullet$  Control characters were designed for transmission

(Demo)



辥	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
健	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
根	色	艳	艴	艵	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

• 109,000 characters

拏	聲	聳	聴	聵	聶	職	聸
建	腲	8073	<b>月</b> 臾	服	殿	周	腸
8171	8172	8173	8174	8175	8176	8177	8178
酿	色	艳	艴	艵	艷	豐色	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 109,000 characters
- 93 scripts (organized)

辥	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
健	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
根	色	艳	艴	艵	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case

辥	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
健	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
根	色	艳	艴	艵	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order

拏	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
鼰	色	艳	艴	艵	艷	豐色	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	夢
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

拏	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
根	色	艳	艴	艵	艷	豐色	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	夢
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

拏	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
銀	色	艳	艴	艵	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	夢
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

http://ian-albert.com/unicode\_chart/unichart-chinese.jpg

U+0058 LATIN CAPITAL LETTER X

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

8171	8172	8173	8174	8175	8176	8177	8178
鼰	色	艳	艴	艵	艶	豐色	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	夢
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

http://ian-albert.com/unicode\_chart/unichart-chinese.jpg

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	恵

腲

色

腳

荳

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE

http://ian-albert.com/unicode\_chart/unichart-chinese.jpg

艷

- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE

<b>拏</b>	聲	聳	原志 8074	聵	最8076	職	鴉
建	腲	腳	<u></u> 則 8174	服	服 8176	届	腸
根	色	艳	艴	艵	艶	艷	艸
影271	<b>1</b> 第272	8273	*************************************	悲		荷	李
怒、	慕	威		8375	8376	**************************************	8378



- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE

簳	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
鼰	色	艳	艴	艵	艷	豐色	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	夢
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸





- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case
- Supports bidirectional display order
- A canonical name for every character

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE

拏	聲	聳	聴	聵	肅	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
銀	色	艳	艴	艵	艷	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
芼	堇	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

http://ian-albert.com/unicode\_chart/unichart-chinese.jpg





(Demo)

Representing Strings: UTF-8 Encoding	
	6

UTF (UCS (Universal Character Set) Transformation Format)

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

6

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

6

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes

6

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes integers

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

0000000

bytes integers

```
UTF (UCS (Universal Character Set) Transformation Format)
```

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

00000000

bytes 00000001 1 integers

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

00000000 0
00000001 1
00000010 2
integers

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes	0000000	0	integers
	0000001	1	
	00000010	2	
	00000011	3	

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes	0000000	0	integers
	0000001	1	
	00000010	2	
	00000011	3	

Variable-length encoding: integers vary in the number of bytes required to encode them.

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes	0000000	Ø	
	0000001	1	integers
	00000010	2	
	00000011	3	

0000000

Variable-length encoding: integers vary in the number of bytes required to encode them.

In Python: string length is measured in characters, bytes length in bytes.

```
UTF (UCS (Universal Character Set) Transformation Format)
```

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

A byte is 8 bits and can encode any integer 0-255.

bytes	0000000	0	integers
	0000001	1	
	00000010	2	
	00000011	3	

0000000

Variable-length encoding: integers vary in the number of bytes required to encode them.

In Python: string length is measured in characters, bytes length in bytes.

(Demo)



[Demo]

[Demo]

First example in the course of an object changing state

[Demo]

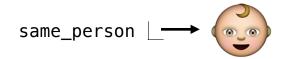
First example in the course of an object changing state

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

[Demo]

First example in the course of an object changing state

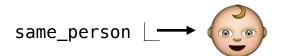
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

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

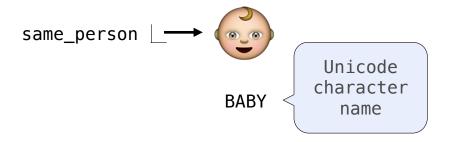


**BABY** 

#### [Demo]

First example in the course of an object changing state

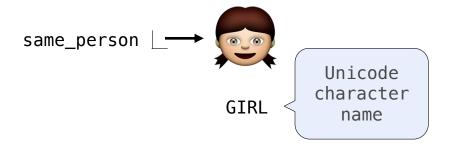
The same object can change in value throughout the course of computation



#### [Demo]

First example in the course of an object changing state

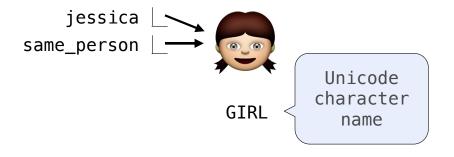
The same object can change in value throughout the course of computation



#### [Demo]

First example in the course of an object changing state

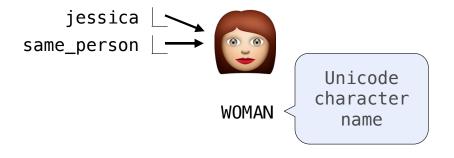
The same object can change in value throughout the course of computation



#### [Demo]

First example in the course of an object changing state

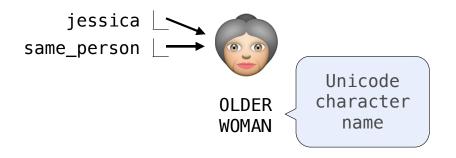
The same object can change in value throughout the course of computation



#### [Demo]

First example in the course of an object changing state

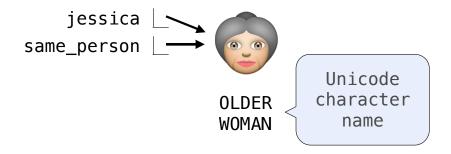
The same object can change in value throughout the course of computation



#### [Demo]

First example in the course of an object changing state

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

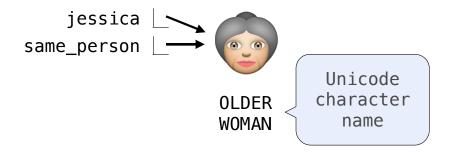


All names that refer to the same object are affected by a mutation

#### [Demo]

First example in the course of an object changing state

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

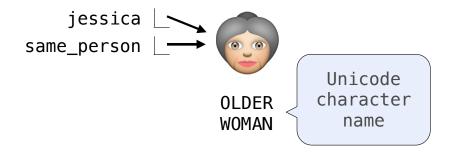


All names that refer to the same object are affected by a mutation Only objects of *mutable* types can change: lists & dictionaries

#### [Demo]

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}

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

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

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

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

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

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

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
```

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

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):
    s.pop()
    s[2:] = []
    s.pop()
```

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):
s.pop()
s[2:] = []
spop()
>>> four = [1, 2, 3, 4]
```

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):
    s.pop()
    s.pop()

    s.pop()

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

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

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

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

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

>>> another_mystery() # No arguments!
>>> len(four)
2
```

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

Tuples

(Demo)

Tuples are Immutable Sequences	

# Tuples are Immutable Sequences

Immutable values are protected from mutation

Immutable values are protected from mutation

Immutable values are protected from mutation

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

Immutable values are protected from mutation

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

Immutable values are protected from mutation

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

Immutable values are protected from mutation

Immutable values are protected from mutation

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

Immutable values are protected from mutation

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

Immutable values are protected from mutation

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

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)

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

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)

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

Immutable values are protected from mutation

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

>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
change turtle's binding

| 'Anything could be inside!']
```

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

Name change:

Immutable values are protected from mutation

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

>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
change turtle's binding

| 'Anything could be inside!']
```

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

>>> X + X

Name change:

>>> X + X

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)

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

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

#### Name change:

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)

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

```
>>> x = 2
>>> x + x
Name change:
>>> x + x
```

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)

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

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)

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

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)

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

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

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)

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

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

```
>>> x = 2

>>> x + x

4

>>> x = 3

>>> x + x

6

Object mutation:

>>> x + x
```

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)

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

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

```
>>> x = 2

>>> x + x

>>> x + x

4

>>> x = [1, 2]

>>> x + x

Object mutation:

>>> x + x
```

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)

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

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

```
Name change: \begin{array}{c} >>> & x = 2 \\ >>> & x + x \\ 4 \\ >>> & x = 3 \\ >>> & x + x \\ 6 \end{array} Object mutation: \begin{array}{c} >>> & x = [1, \ 2] \\ >>> & x + x \\ [1, \ 2, \ 1, \ 2] \\ >>> & x + x \\ >> & x + x \\ >>> & x + x \\ >> & x + x \\ >>> & x + x \\ >> & x +
```

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)

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

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

```
>>> x = 2

>>> x + x

4

>>> x = 3

>>> x = 3

>>> x + x

Object mutation: | 1, 2]

>>> x append(3)

>>> x + x
```

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)

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

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

```
Name change:

>>> x = 2
>>> x + x
4
>>> x = [1, 2]
>>> x + x
[1, 2, 1, 2]
>>> x append(3)
>>> x + x
[1, 2, 1, 2]
>>> x + x
[1, 2, 3, 1, 2, 3]
```

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)

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

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

```
Name change:

>>> x = 2
>>> x + x
4
>>> 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

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)

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

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

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

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

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)

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

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

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

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

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)

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

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

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

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

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)

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

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

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

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

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)

| Next lecture: ooze can
change turtle's binding
| 'Anything could be inside!']
```

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

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

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

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)

| Next lecture: ooze can
change turtle's binding
| 'Anything could be inside!']
```

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

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 ERROR >>> s
```

Immutable values are protected from mutation

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

Next lecture: ooze can
change turtle's binding

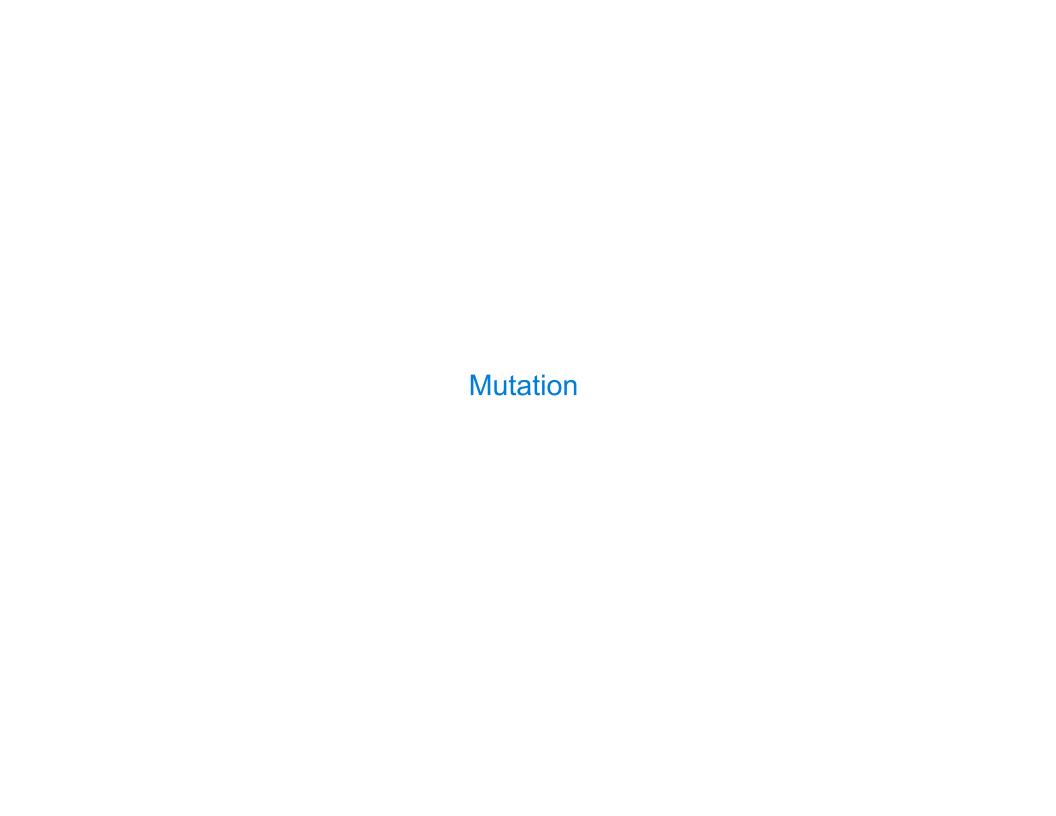
>>> turtle
(1, 2, 3)

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

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

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

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



Sameness and Change	

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

- 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

- 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

- \*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

- 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

- 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

- 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

- 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

- 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

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

- 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

```
>>> a = [10]

>>> b = a

>>> a == b

True

>>> a.append(20)

>>> a == b

True
```

- 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

```
>>> a = [10]

>>> b = a

>>> a == b

True

>>> a.append(20)

>>> a == b

True

>>> a

[10, 20]
```

- 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

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

- · 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 == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

- 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 == b

True

>>> a

[10, 20]

>>> b

[10, 20]
```

- · 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]
```

- · 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]
```

- 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 = [10]
>>> b = a
>>> a == b
                                    >>> a == b
True
                                    True
                                    >>> b_append(20)
>>> a_append(20)
>>> a == b
True
>>> a
[10, 20]
>>> b
[10, 20]
```

- · 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 = [10]
>>> b = a
>>> a == b
                                    >>> a == b
True
                                    True
                                    >>> b_append(20)
>>> a_append(20)
>>> a == b
                                     >>> a
True
                                     [10]
>>> a
[10, 20]
>>> b
[10, 20]
```

- · 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 = [10]
>>> b = a
>>> a == b
                                    >>> a == b
True
                                    True
                                    >>> b_append(20)
>>> a append(20)
>>> a == b
                                     >>> a
True
                                     [10]
>>> a
                                     >>> h
[10, 20]
                                     [10, 20]
>>> b
[10, 20]
```

- · 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 = [10]
>>> b = a
>>> a == b
                                    >>> a == b
True
                                    True
                                    >>> b_append(20)
>>> a append(20)
>>> a == b
                                     >>> a
True
                                     [10]
>>> a
                                     >>> h
[10, 20]
                                    [10, 20]
>>> b
                                     >>> a == b
[10, 20]
                                     False
```

Identity Operators					

## **Identity**

<exp0> is <exp1>

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

#### **Identity**

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

#### **Equality**

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

#### **Identity**

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

#### **Equality**

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

Identical objects are always equal values

#### **Identity**

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

#### **Equality**

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

Identical objects are always equal values

(Demo)

ments are Dang	,	 

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

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

```
>>> def f(s=[]):
... s.append(5)
... return len(s)
```

15

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

15

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

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

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

```
Global frame
>>> def f(s=[]):
                                                               →func f(s) [parent=Global]
         s_append(5)
                                                   f
          return len(s)
                                                                list
                                                                 3
                                                                     3
                                 f1: f [parent=Global]
>>> f()
>>> f()
                                              Return
                                               value
>>> f()
                                 f2: f [parent=Global]
                                              Return
                                               value
                                 f3: f [parent=Global]
                                              Return
                                               value
```

<u>Interactive Diagram</u>

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

```
Global frame
>>> def f(s=[]):
                                                            >func f(s) [parent=Global]
         s append(5)
                                                 f
                                                              list
         return len(s)
                                                                  3
                                                               3
                                f1: f [parent=Global]
>>> f()
>>> f()
                                            Return
                                             value
                                                               Each time the function
                                                                is called, s is bound
>>> f()
                                f2: f [parent=Global]
                                                                  to the same value!
                                            Return
                                             value
                                f3: f [parent=Global]
                                            Return
                                             value
```

<u>Interactive Diagram</u>