

(Demo)

Objects		

• Objects represent information

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions

- Objects represent information
- They consist of data and behavior, bundled together to create abstractions
- Objects can represent things, but also properties, interactions, & processes

- 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

- 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:

- 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

- 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

- 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

- 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

- 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

- 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)

American Standard Code for Information Interchange

						Α	SCII	Coo	de Cl	hart						
	0	1	₁ 2	3	ι 4	լ 5	6	7	8	ا 9	ΙA	В	C	_ D _	Ε	<u> </u>
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	Ŀ	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
2			=	#	\$	%	w	-	()	*	+	,	-	•	/
3	0	1	2	3	4	5	6	7	8	9		;	٧	II	۸	?
4	@	Α	В	U	D	Е	F	G	Н	I	J	K	Г	М	N	0
5	Р	œ	R	S	Т	5	V	W	Х	Υ	Z]	/]	^	_
6	,	а	b	U	d	е	f	g	h	i	j	k	٦	m	n	0
7	р	q	r	s	t	u	٧	W	Х	У	z	{		}	1	DEL

American Standard Code for Information Interchange

							Α	SCI	Coo	de Cl	nart						
	لـ	0	1	2	3	4	լ 5	_I 6	7	8	9	Α	В	C	D	E	<u> </u>
	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		Ţ.	11	#	\$	%	&	-	()	*	+	,	-	•	/
Μ	3	0	1	2	3	4	5	6	7	8	9		;	٧	=	^	?
 S	4	@	Α	В	С	D	Е	F	G	Н	I	J	K	L	М	N	0
rows	5	Р	œ	R	S	Т	5	٧	W	Х	Υ	Z	[\]	^	
	6	,	а	b	С	d	е	f	g	h	i	j	k	٦	m	n	0
∞	7	р	q	r	s	t	u	٧	W	Х	у	Z	{		}	1	DEL

American Standard Code for Information Interchange

									A	ZCTI	. Coo	de Ci	nart						
			لـ	0	1	₁ 2	3	ι 4	լ 5	6	7	8	9	_I A	В	С	D	E	_F_
0 0	0	I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0 0	1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1	0	bi	2		ļ.	"	#	\$	%	&	1	()	*	+	,	-	•	/
0 1	1	Μ	3	0	1	2	3	4	5	6	7	8	9	:	;	٧	=	۸	?
1 0	0	 S	4	@	Α	В	U	D	Е	F	G	H	I	J	K	L	М	N	0
1 0	1	SMO	5	Р	œ	R	S	Т	U	٧	W	Х	Υ	Z	[\]	^	
1 1	0	_	6	`	а	ь	U	d	е	f	g	h	i	j	k	٦	m	n	0
1 1	1	∞	7	р	q	r	s	t	u	٧	W	Х	у	Z	}		}	ı	DEL

American Standard Code for Information Interchange

										Α	SCII	Coc	de Cl	nart						
				لـ	0	1	2	3	4	5	6	7	8	9	Α	ιВ	С	D	E	L F
0	0	0	I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	Ŀ	VT	FF	CR	SO	SI
0	0	1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0	1	0	bi	2		ļ.	ш	#	\$	%	&	1	()	*	+	,	-	•	/
0	1	1	Μ	3	0	1	2	3	4	5	6	7	8	9		;	٧	=	^	?
1	0	0	 S	4	0	Α	В	C	D	Е	F	G	H	Ι	ין	K	L	М	N	0
1	0	1	SMO.	5	Р	Q	R	S	Т	C	V	W	Χ	Υ	Z	[\]	^	_
1	1	0	_	6	`	а	р	U	d	e	f	g	h	i	j	k	٦	m	n	0
1	1	1	∞		р	q	r	s	t	u	V	W	Х	у	z	{		}	2	DEL

16 columns: 4 bits

6

American Standard Code for Information Interchange

										Α	SCII	Coc	de Cl	nart						
				لـ	0	1	2	3	4	5	6	7	8	9	Α	ιВ	С	D	E	L F
0 (0	0	I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	Ŀ	VT	FF	CR	SO	SI
0 (0	1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1	1	0	bi	2			=	#	\$	%	8	-	()	*	+	,	-	•	/
0 1	1	1	Μ	3	0	1	2	з	4	5	6	7	8	9		;	٧	II	۸	?
1 (0	0	 S	4	@	Α	В	U	ם	Е	F	G	Н	Ι	ר	K	L	М	N	0
1 (0	1	SM0.	5	Р	œ	R	S	Т	5	٧	W	Х	Υ	Z	[/]	<	_
1 3	1	0	_	6	`	а	р	U	d	e	f	g	h	i	j	k	٦	m	n	0
1 3	1	1	∞	7	р	q	r	s	t	u	V	W	х	у	Z	{		}	7	DEL

16 columns: 4 bits

• Layout was chosen to support sorting by character code

6

American Standard Code for Information Interchange

										Α	SCII	Coc	de Cl	nart						
				لـ	0	1	2	3	4	լ 5	6	7	8	9	Α	В	C	D 1	E	L F
0 (9	0	I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0 (0	1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1	1	0	bi	2			=	#	\$	%	w	-	()	*	+	,	-	٠	/
0 1	1	1	\sim	3	0	1	2	з	4	5	6	7	8	9		;	٧	=	۸	?
1 (0	0	 S	4	@	Α	В	U	D	Е	H	G	Н	Ι	ח	K	L	М	N	0
1 (0	1	rows	5	Р	œ	R	S	Т	5	٧	W	Х	Υ	Z	[\]	<	_
1 3	1	0		6	`	а	ь	U	d	Φ	f	g	h	i	j	k	l	m	n	0
1 3	1	1	∞	7	р	q	r	s	t	u	V	W	Х	у	Z	{		}	ı	DEL

- 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

										Α	SCII	Coc	de Cl	nart						
				لـ	0	1	2	3	4	լ 5	6	7	8	9	Α	В	C	D 1	E	L F
0 (9	0	I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0 (0	1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1	1	0	bi	2			=	#	\$	%	W	-	()	*	+	,	-	٠	/
0 1	1	1	\sim	3	0	1	2	з	4	5	6	7	8	9		;	٧	=	۸	?
1 (0	0	 S	4	@	Α	В	U	D	Е	H	G	Н	Ι	ח	K	L	М	N	0
1 (0	1	rows	5	Р	œ	R	S	Т	5	٧	W	Х	Υ	Z	[\]	<	_
1 3	1	0		6	`	а	ь	U	d	Φ	f	g	h	i	j	k	l	m	n	0
1 3	1	1	∞	7	р	q	r	s	t	u	V	W	Х	у	Z	{		}	ı	DEL

- 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	SCI	Coo	de Cl	hart		11	Line	fee	ed"	(\n)
			0	1	2	_I 3	4	₁ 5	_I 6	_I 7	8	9	L A /	В	С	D	E	ı F ı
0 0 0	Ī	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
0 0 1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1 0	bi	2		!	=	#	\$	%	&	1	()	*	+	,	-	•	/
0 1 1	\sim	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	^	?
1 0 0	 S	4	@	Α	В	С	D	Е	F	G	Н	I	J	K	L	М	N	0
1 0 1	rows	5	Р	Q	R	S	T	U	٧	W	Х	Υ	Z	[\]	^	_
1 1 0		6	`	а	b	С	d	е	f	g	h	i	j	k	l	m	n	0
1 1 1	∞	7	р	q	r	s	t	u	v	w	х	у	z	{		}	1	DEL

- 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

					"Ве	ell"	(\a)	SCI	Coo	de Cl	hart		11	Line	fee	ed"	(\n)
		١	0	1	2	3	4	5	6	₁ 7	8	9	L A	В	С	D	Ε	ı F ı
0 0 0	T	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
0 0 1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1 0	bits	2		!	"	#	\$	%	&	-	()	*	+	,	-	•	/
0 1 1	m	3	0	1	2	3	4	5	6	7	8	9	:	;	٧	=	>	?
1 0 0		4	@	Α	В	С	D	Е	F	G	Н	I	J	К	L	М	N	0
1 0 1	rows	5	Р	Q	R	S	Т	U	٧	W	Х	Υ	Z	[\]	^	-
1 1 0		6	,	а	b	С	d	е	f	g	h	i	j	k	ι	m	n	0
1 1 1	∞	7	р	q	r	s	t	u	v	w	х	у	Z	{	T	}	1	DEL

- 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

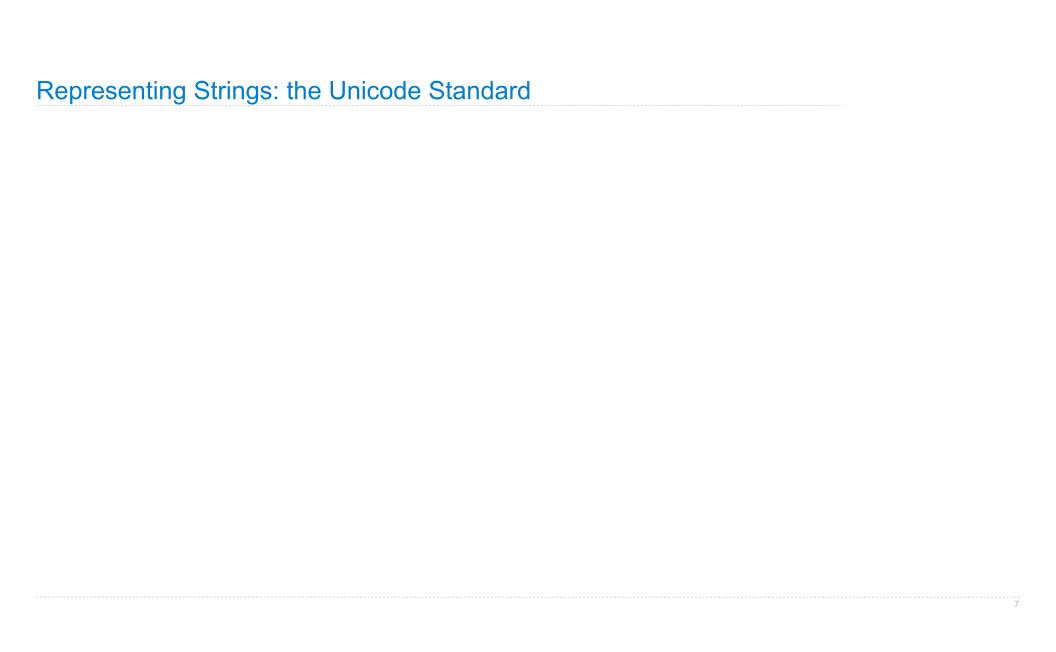
American Standard Code for Information Interchange

					"Be	ell"	(\a)	SCI	[Co	de Cl	hart		11	Line	fee	ed"	(\n)
		١	0	1	2	3	4	5	6	₁ 7	8	9	L A	В	С	_L D	ιE	ı F ı
0 0 0	Ī	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
0 0 1	ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
0 1 0	βi	2		į.	=	#	\$	%	&	-	()	*	+	,	-	•	/
0 1 1	m	3	0	1	2	3	4	5	6	7	8	9		;	<	=	>	?
1 0 0	 S	4	@	Α	В	С	D	Е	F	G	Н	I	J	K	L	М	N	0
1 0 1	rows	5	Р	Q	R	S	T	U	٧	W	Х	Υ	Z	[\]	^	_
1 1 0		6	`	а	ь	U	d	е	f	g	h	i	j	k	ι	m	n	0
1 1 1	∞	7]	р	q	٢	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

(Demo)



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
根	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

• 137,994 characters in Unicode 12.1

拏 8071	聲	聳	1恵	聵	最	職	鴉
建	腲	腳	<u>腴</u>	服	殿	届	腸
製	色	丰色	艴	絶	豊色	豐色	↓ ↓↓↓ 8278
耄	重	芸.	荴	志	芩	荷 8377	李 8378
葱	慕	葳	葴	葵	葶	葷	恵

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

- 1

- 137,994 characters in Unicode 12.1
- 150 scripts (organized)

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
银	色	艳	艴	絶	艶	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

- 1

- 137,994 characters in Unicode 12.1
- 150 scripts (organized)
- Enumeration of character properties, such as case

拏	聲	聳	原志 8074	聵 8075	最8076	職	鴉
建	腲	腳	<u></u> 則 8174	服	服 8176	届	腸
根	色	艳	艴	艵	艶	艷	艸
整71	8272	8273	接	悲		荷	**************************************
刻71	慕	威	武	8375	8376	8377	8378

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

拏	聲	聳	聴	聵	肅	職	聸
建	腲	腳	月 臾	服	服	周	腸
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	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

- 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

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

- 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

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	丱
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

LATIN CAPITAL LETTER A

- 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

拏	聲	聳	腮	購	聶	職	牅
8071	8072	8073	8074	8075	8076	8077	8078
建	腲	腳	<u>胂</u>	服	股	届	腸
根	色	艳	艴	艵	艶	艷	艸
芝	1	8273	*************************************			荷	
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

LATIN CAPITAL LETTER A

DIE FACE-6

- 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

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	艶	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

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

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

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

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

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	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



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

LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

拏	聲	聳	聴	聵	肅	職	聸
建	腲	腳	月 臾	服	服	周	腸
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





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

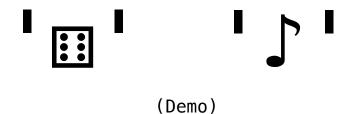
LATIN CAPITAL LETTER A

DIE FACE-6

EIGHTH NOTE

警	聲	聳	题 8074	聵	8076	職	鴉
健	腲	腳	腴	服	腶	腷	腸
製	8172	#色	#色	#色	8176 豊 佐	<u>8177</u>	8178 <u></u>
整71	8272	8273	荴	恕5		荷	**************************************
刻71	慕	威		8375	8376	8377	8378

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



- 1



[<u>Demo</u>]

[<u>Demo</u>]

First example in the course of an object changing state

[<u>Demo</u>]

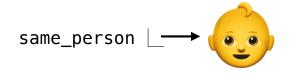
First example in the course of an object changing state

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

[<u>Demo</u>]

First example in the course of an object changing state

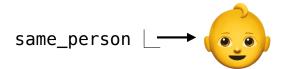
The same object can change in value throughout the course of computation



[<u>Demo</u>]

First example in the course of an object changing state

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

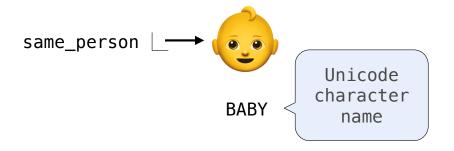


BABY

[<u>Demo</u>]

First example in the course of an object changing state

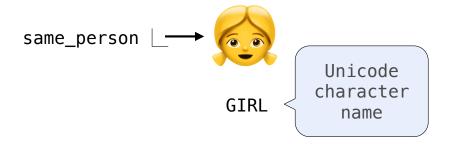
The same object can change in value throughout the course of computation



[<u>Demo</u>]

First example in the course of an object changing state

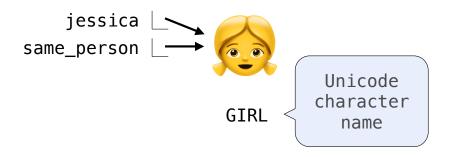
The same object can change in value throughout the course of computation



[<u>Demo</u>]

First example in the course of an object changing state

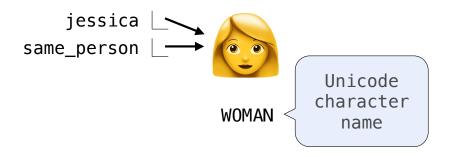
The same object can change in value throughout the course of computation



[<u>Demo</u>]

First example in the course of an object changing state

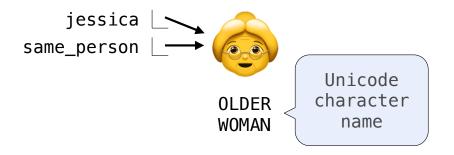
The same object can change in value throughout the course of computation



[<u>Demo</u>]

First example in the course of an object changing state

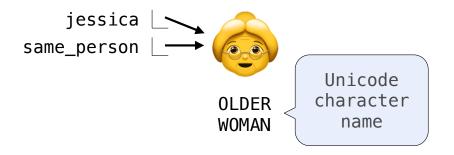
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

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

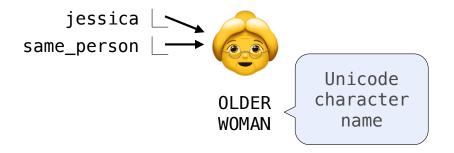


All names that refer to the same object are affected by a mutation

[Demo]

First example in the course of an object changing state

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



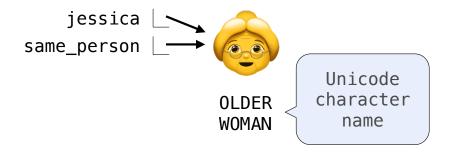
All names that refer to the same object are affected by a mutation Only objects of *mutable* types can change: lists & dictionaries

J

[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

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

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

+ --0

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
```

+--0

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!
```

+ --0

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

+ --0

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.pop()
s.pop()

**Sopop()
**Sopo
```

+--0

Tuples

(Demo)

Tuples are Immutable Sequences	
	1

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
(1, 2, 3)
```

```
>>> 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)
```

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

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]
```

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]
```

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

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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

```
>>> 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

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

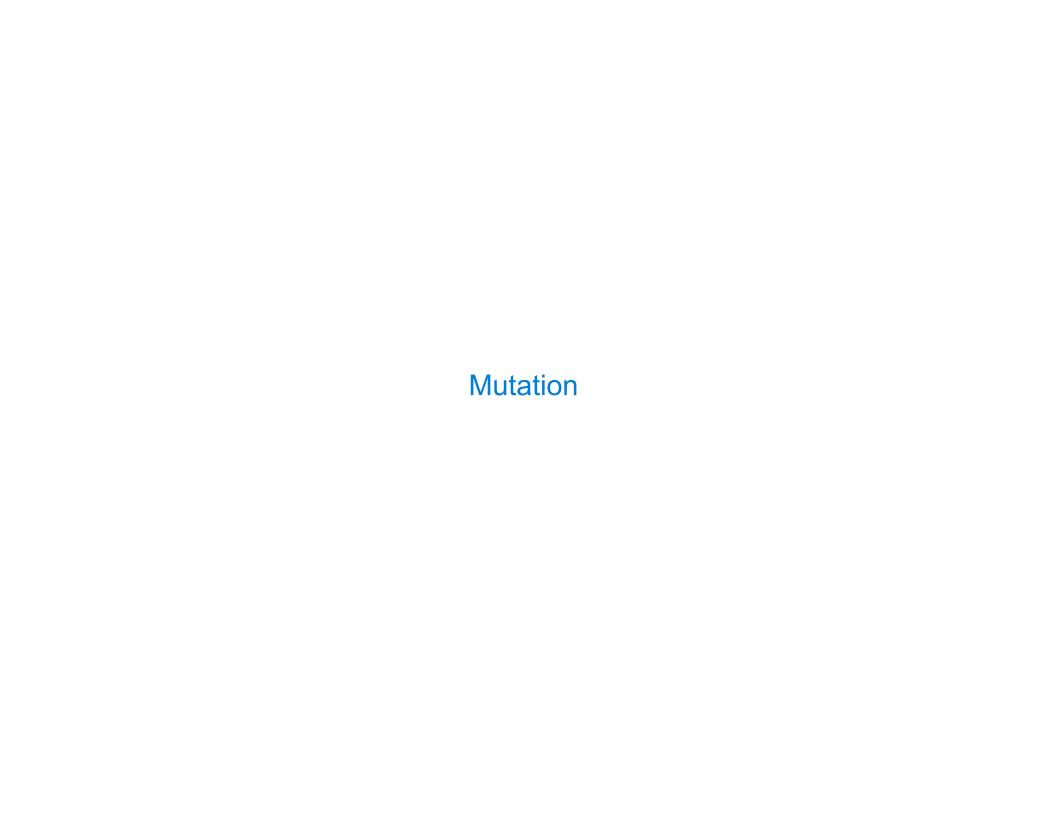
| Next lecture: ooze can
change turtle's binding
| 'Anything could be inside!']
```

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

```
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]
```

```
>>> 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

[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

[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

```
>>> a = [10]

>>> b = a

>>> a == b

True

>>> a.append(20)

>>> a

[10, 20]

>>> b

[10, 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
- ·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
```

- · 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
```

- · 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)

>>> b

[10, 20]

>>> b

[10, 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
- ·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
```

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

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)

Mutable Default Arguments are Dangerous	
	16

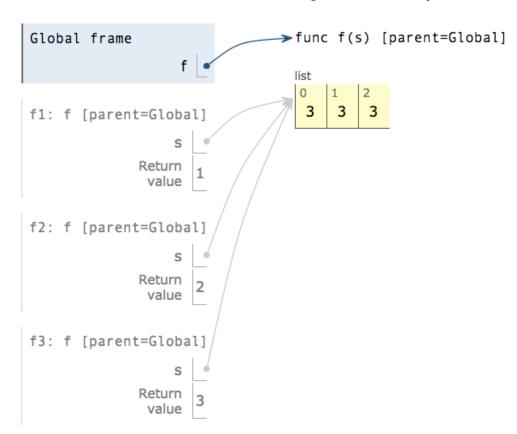
A default argument value is part of a function value, not generated by a call

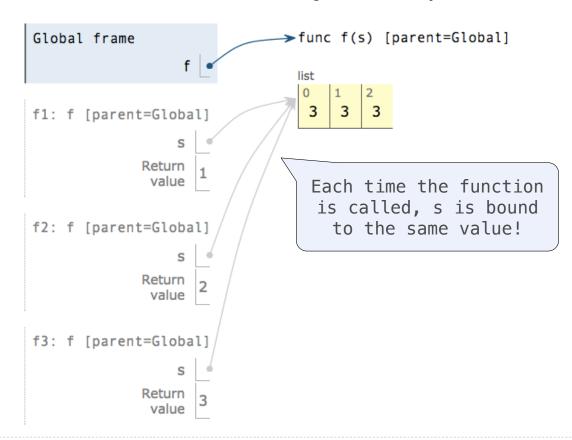
```
>>> def f(s=[]):
... s.append(3)
... return len(s)
```

16

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

16







Lists in Environment Diagrams					

Assume that before each example below we execute:

```
s = [2, 3]
t = [5, 6]
```

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation

Assume that before each example below we execute:

s = [2, 3]t = [5, 6]

Operation Example

Assume that before each example below we execute:

s = [2, 3]t = [5, 6]

Operation Example Result

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result
<pre>append adds one element to a list</pre>		

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	

Global

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result	Glo	bal		
<pre>append adds one element to a list</pre>	s.append(t) t = 0		s t		 list 0 2	3
					list 0 5	16

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

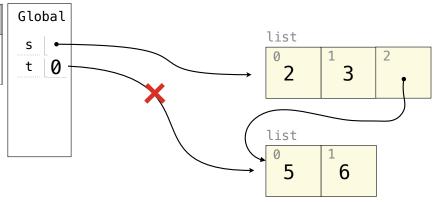
Operation Operation	Example	Result	Global			
<pre>append adds one element to a list</pre>	s.append(t) t = 0		s • t	list 2 list	¹ 3	2

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>		t 0	list 0 2 1 3 2 list 0 5 6

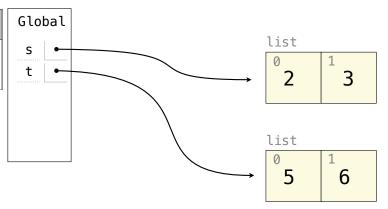
Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	$s \rightarrow [2, 3, [5, 6]]$ $t \rightarrow 0$



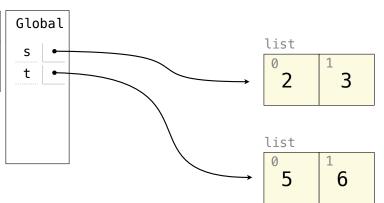
Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	$s \rightarrow [2, 3, [5, 6]]$ $t \rightarrow 0$



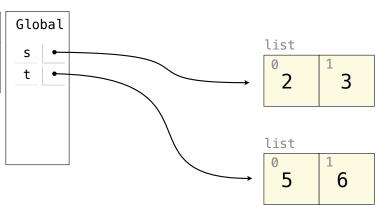
Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	$s \rightarrow [2, 3, [5, 6]]$ t \rightarrow 0
<pre>extend adds all elements in one list to another list</pre>		



Assume that before each example below we execute:

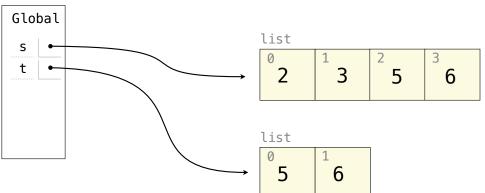
Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	



Assume that before each example below we execute:

$$t = [5, 6]$$

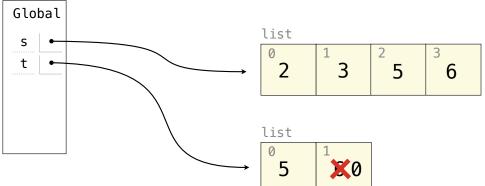
Operation Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	



Assume that before each example below we execute:

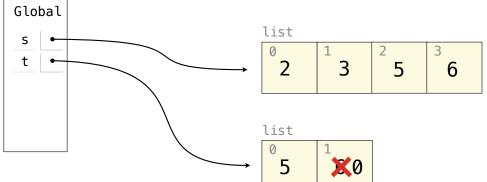
$$t = [5, 6]$$

Operation Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	



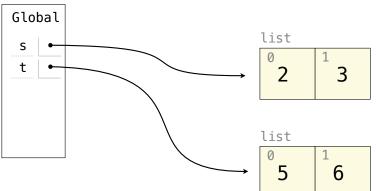
Assume that before each example below we execute:

Operation Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ $t \rightarrow [5, 0]$



Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ $t \rightarrow [5, 0]$



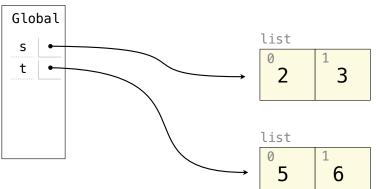
Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

create new lists containing existing

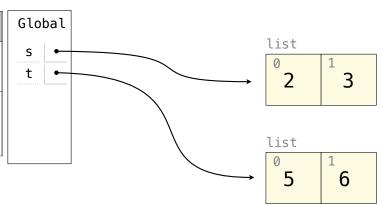
elements

Operation Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]
addition & slicing		·



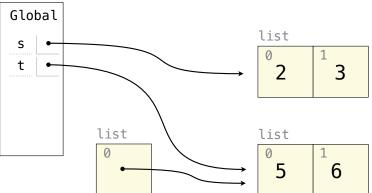
Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>	s → [2, 3, [5, 6]] t → 0
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	



Assume that before each example below we execute:

Operation	Example	Result	Gl
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0	s
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]	
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0		



Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0	s •	$\begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array}$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]		list
addition & slicing create new lists containing existing	a = s + [t] b = a[1:] a[1] = 9			$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$
elements	b[1][1] = 0			0 1 3 2 1 3 2 1 1 3 1 2 1 1 1 1 1 1 1 1 1

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0	s • • • • • • • • • • • • • • • • • • •	$ \begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array} $
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]	a •	list
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0		'[]	$\begin{array}{c c} 0 & & & \\ \hline & 5 & & 6 \end{array}$ list
	!			2 3 2

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>	s → [2, 3, [5, 6]] t → 0	s •	$\begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array}$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]	a •	list
addition & slicing create new lists containing existing	a = s + [t] b = a[1:] a[1] = 9			$\begin{array}{c ccccccccccccccccccccccccccccccccccc$
elements	b[1][1] = 0			1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>	s → [2, 3, [5, 6]] t → 0	s •	$ \begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array} $
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]	a b	list
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0		\	$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$
				2 3 2

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>	s → [2, 3, [5, 6]] t → 0	s t	$\begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array}$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]	a b	list
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0			0 5 1 6
				2 1 3 2 list
				3 1

Assume that before each example below we execute:

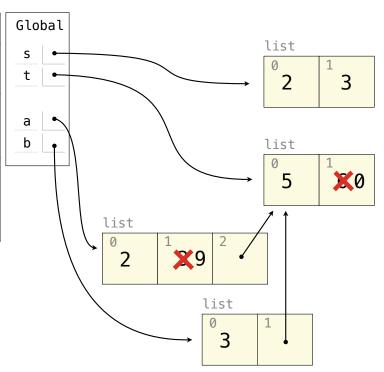
Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	<pre>s.append(t) t = 0</pre>	s → [2, 3, [5, 6]] t → 0	5 • t	$\begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array}$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	s → [2, 3, 5, 6] t → [5, 0]	a b	list
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0		list	5 6
		-	2 1	list
				→ 0 1 1 ·

Assume that before each example below we execute:

Operation	Example	Result	Global	
<pre>append adds one element to a list</pre>	s.append(t) t = 0	s → [2, 3, [5, 6]] t → 0	s • t •	$\begin{array}{c c} & \text{list} \\ \hline & 2 & 3 \end{array}$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ t \rightarrow [5, 0]	a b	list
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0		list	
			2	list
				3 1

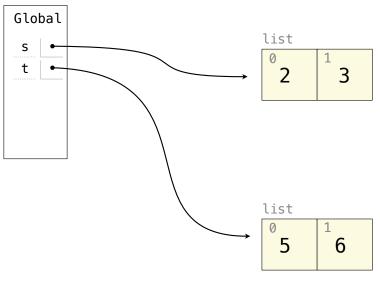
Assume that before each example below we execute:

Operation	Example	Result
<pre>append adds one element to a list</pre>	s.append(t) t = 0	$s \rightarrow [2, 3, [5, 6]]$ $t \rightarrow 0$
<pre>extend adds all elements in one list to another list</pre>	s.extend(t) t[1] = 0	$s \rightarrow [2, 3, 5, 6]$ $t \rightarrow [5, 0]$
addition & slicing create new lists containing existing elements	a = s + [t] b = a[1:] a[1] = 9 b[1][1] = 0	$s \rightarrow [2, 3]$ $t \rightarrow [5, 0]$ $a \rightarrow [2, 9, [5, 0]]$ $b \rightarrow [3, [5, 0]]$



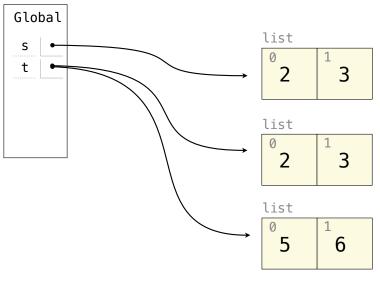
Assume that before each example below we execute:

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



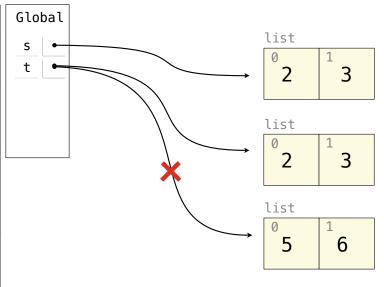
Assume that before each example below we execute:

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



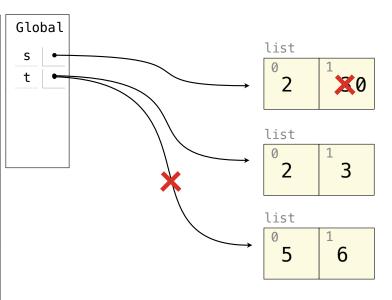
Assume that before each example below we execute:

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



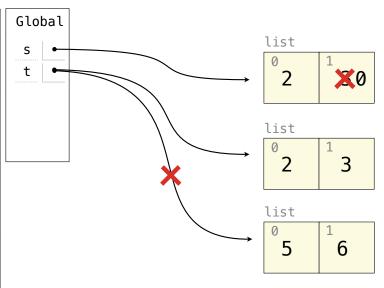
Assume that before each example below we execute:

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



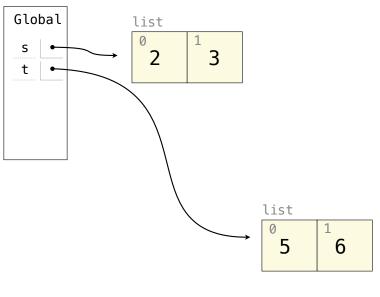
Assume that before each example below we execute:

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



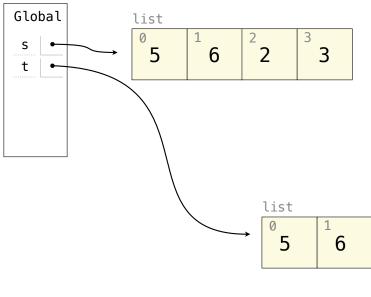
Assume that before each example below we execute:

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



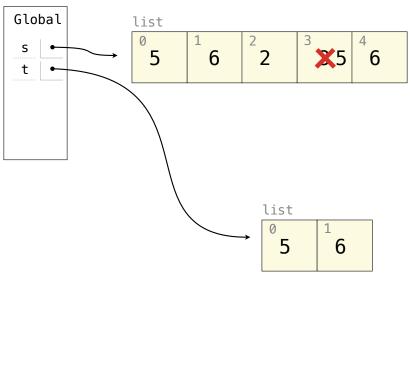
Assume that before each example below we execute:

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



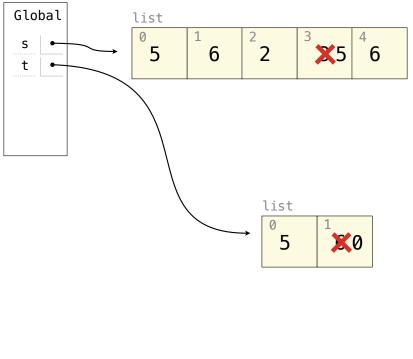
Assume that before each example below we execute:

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



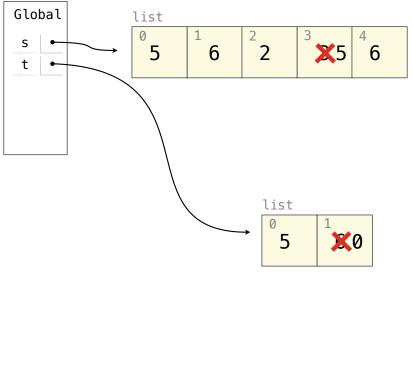
Assume that before each example below we execute:

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



Assume that before each example below we execute:

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



Assume that before each example below we execute:

Assume that before each example below we execute:

Operation	Example	Result

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result
<pre>pop removes & returns the last element</pre>		

22

Assume that before each example below we execute:

s = [2, 3] t = [5, 6]

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	

22

Assume that before each example below we execute:

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	s → [2] t → 3

Assume that before each example below we execute:

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	s → [2] t → 3
<pre>remove removes the first element equal to the argument</pre>		

Assume that before each example below we execute:

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	s → [2] t → 3
remove removes the first element equal to the argument	t.extend(t) t.remove(5)	

Assume that before each example below we execute:

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	s → [2] t → 3
<pre>remove removes the first element equal to the argument</pre>	t.extend(t) t.remove(5)	$s \rightarrow [2, 3]$ $t \rightarrow [6, 5, 6]$

Assume that before each example below we execute:

Operation	Example	Result
<pre>pop removes & returns the last element</pre>	t = s.pop()	s → [2] t → 3
<pre>remove removes the first element equal to the argument</pre>	t.extend(t) t.remove(5)	$s \rightarrow [2, 3]$ t \rightarrow [6, 5, 6]
<pre>slice assignment can remove elements from a list by assigning [] to a slice.</pre>		•

Assume that before each example below we execute:

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

Assume that before each example below we execute:

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

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

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

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



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

```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
[t] evaluates to:
\begin{bmatrix} t \\ t \\ t \end{bmatrix}
\begin{bmatrix} t \\ t \\ t \end{bmatrix}
```

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

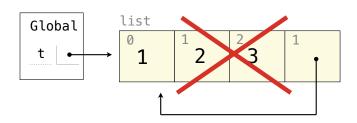
```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
[t] evaluates to:
[t] evaluates to:
[t] formula is the content of the conte
```

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

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

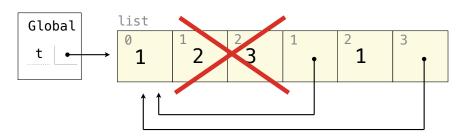
```
t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```

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



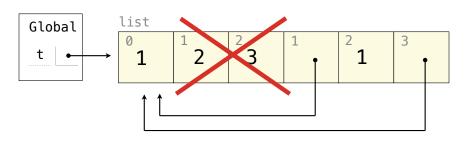
```
t = [[1, 2], [3, 4]]
t[0].append(t[1:2])
```

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



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

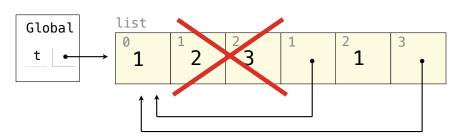
```
t = [1, 2, 3]
t[1:3] = [t]
t.extend(t)
```



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

23

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



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

$$t = [[1, 2], [3, 4]]$$

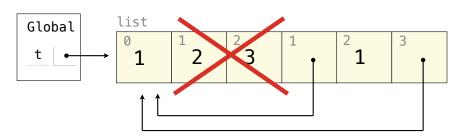
$$t[0].append(t[1:2])$$

$$\downarrow list$$

$$0 \downarrow 1$$

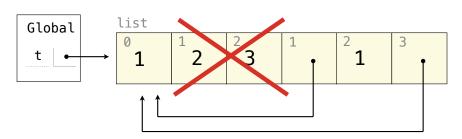
$$0$$

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

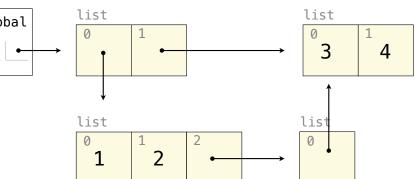


[1, [...], 1, [...]]

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

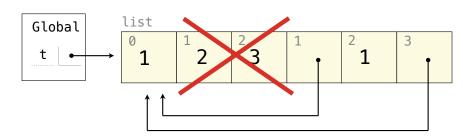


[1, [...], 1, [...]]

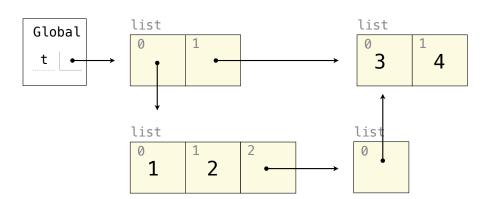


23

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



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



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