

(Demo)

• 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

4

- 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

4

- 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

ASCII Code Chart NUL SOH STX ETX EOT ENQ ACK BEL HT LF VT | FF SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB | ESC | # & 3 5 7 9 1 6 С Ε F Q S Χ Ζ С h m X s

American Standard Code for Information Interchange

							Α	SCII	Coc	de Cl	hart						
	لـ	0	1	2	3	4	5	6	7	8	9	Α	В	C	D	E	L F
	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
ts	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ЕТВ	CAN	EM	SUB	ESC	FS	GS	RS	US
bit	2		:-	11	#	\$	%	ß	-	()	*	+	,		•	/
\sim	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	Г	М	N	0
rows	5	Р	Q	R	S	Т	5	٧	W	Х	Υ	Z	[\]	<	
	6	`	а	ь	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	{		}	?	DEL

American Standard Code for Information Interchange

							Α	SCII	Coc	de Cl	hart						
	١	0	1	2	3	4	5	6	7	8	9	Α	В	C	D	E	∟F _J
	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	ഥ	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			=	#	\$	%	w	-	()	*	+	,	•	•	/
\sim	3	0	1	2	з	4	5	6	7	8	9		;	٧	II	۸	?
 S	4	@	Α	В	U	D	Е	F	G	Н	I	ר	K	٦	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	7	m	n	0
∞	7	р	q	r	s	t	u	٧	W	х	у	Z	{		}	~	DEL

American Standard Code for Information Interchange

							A	SCII	Coc	de Cl	hart						
	٦	0	1	2	3	4	5	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			=	#	\$	%	w	-	()	*	+	,	-	•	/
\sim	3	0	1	2	з	4	5	6	7	8	9		;	٧	II	۸	?
 S	4	@	Α	В	U	D	Е	F	G	Н	I	ח	K	٦	М	N	0
rows	5	Р	œ	R	S	T	U	٧	W	Х	Υ	Z	[\]	^	
	6	,	а	ь	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	{		}	~	DEL

16 columns: 4 bits

• Layout was chosen to support sorting by character code

U

American Standard Code for Information Interchange

							Α	SCII	Coc	de Cl	hart						
	لـ	0	1	2	3	4	5	6	7	8	9	Α	В	C	D	E	L F
	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	ഥ	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			Ξ	#	\$	%	w	-	()	*	+	,	1	•	/
\sim	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	Т	J	٧	W	Х	Υ	Z	[\]	^	_
	6	,	а	ь	U	d	e	f	g	h	i	j	k	٦	m	n	0
∞	7	р	q	r	S	t	u	٧	W	х	у	Z	{		}	?	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	hart						
	Ь	0	1	2	3	4	5	6	7	8	9	Α	В	C	D	E	L F
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		-	=	#	\$	%	w	-	()	*	+	,	1	•	/
\sim	3	0	1	2	3	4	5	6	7	8	9		;	٧		۸	?
 S	4	@	Α	В	U	D	Е	F	G	Н	I	ח	K	L	М	N	0
rows	5	Р	Q	R	S	T	5	٧	W	Х	Υ	Z	[\]	^	-
	6	,	а	۵	U	а	ø	f	g	h	i	j	k	٦	m	n	0
∞	<u></u>	р	q	r	S	t	u	٧	W	х	у	Z	{		}	?	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	SCII	Coc	de Cl	hart		11	Line	fee	ed''	(\n)
	_	0	1	2	3	4	5	6	7	8	9	L A /	В	С	_l D	E	ı F ı
I	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	S0	SI
bits	ı	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
bi	2		!	=	#	\$	%	&	-	()	*	+	,	-	•	/
സ	3	0	1	2	3	4	5	6	7	8	9	:	;	٧	=	^	?
	4	@	Α	В	C	D	Е	F	G	Н	I	J	K	L	М	N	0
rows	5	Р	Q	R	S	Т	U	٧	W	Х	Υ	Z	[\]	^	_
	6	`	а	b	U	d	е	f	g	h	i	j	k	ι	m	n	0
∞	7	р	q	r	s	t	u	٧	W	х	у	z	{	Τ	}	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

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

16 columns: 4 bits

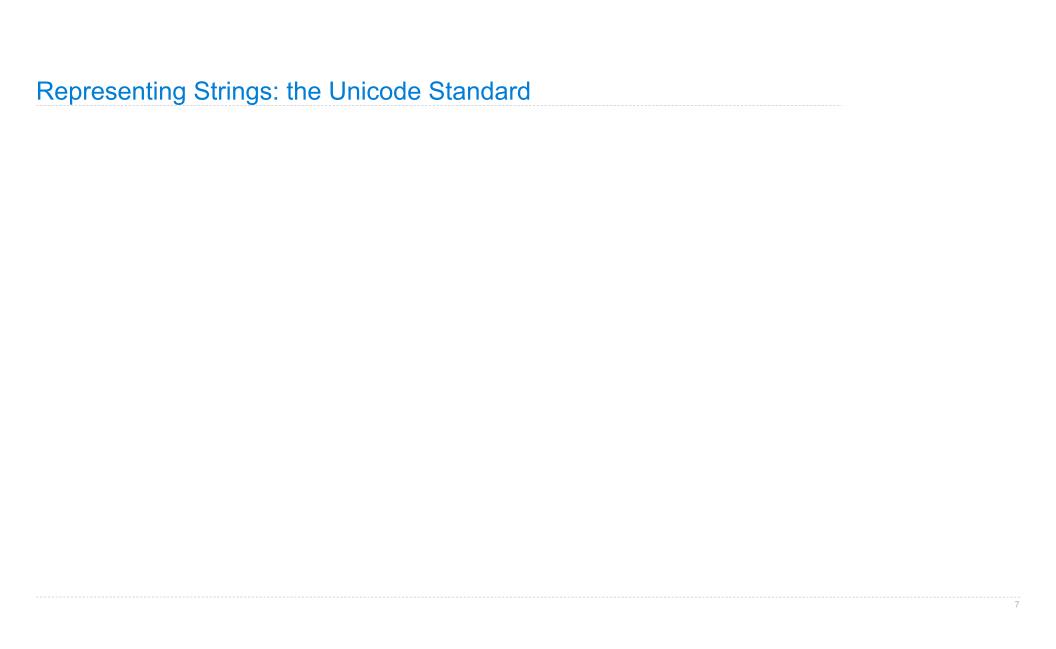
- Layout was chosen to support sorting by character code
- Rows indexed 2-5 are a useful 6-bit (64 element) subset
- Control characters were designed for transmission

American Standard Code for Information Interchange

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

• 109,000 characters

幹 8071	聲	聳	惠 8074	聵	最	職	鴉
建	腲	腳	<u></u> 則 8174	服	服 8176	届	腸
製	色 8272	丰色	弗色	絶	豊色	豐色	### 8278
記 記 8371	1 1 8372	8373	荴	志	荶	荷 8377	学 8378
葱	葲	葳	葴	葵	葶	葷	恵

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

- /

- 109,000 characters
- 93 scripts (organized)

拏	聲	聳 8073	月恵 8074	聵 8075	最8076	職	贍
建	腲	腳	<u>胂</u>	服	服 8176	届	腸
製	色 8272	丰色	売	絶	豐色	豐色	↓ ↓↓↓ 8278
芼	重	豊 8373	荴	悲	荶	荷	学 8378
葱	葲	葳	葴	葵	葶	葷	恵

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

拏	聲	聳	原 8074	聵	最8076	職	鴉
建	腲	腳	<u></u> 則 8174	服	股	届	腸
根	色	艳	艴	艵	艶	艷	艸
整71	8272	8273		悲		荷	**************************************
慰	款		蔵	8375	8376	8377	8378

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

拏	聲	聳	原惠 8074	聵	最8076	職	鴉
建	腲	腳	腴	腵	腶	腷	腸
朝	8172	*E	#色	*************************************	8176 豊色	<u>8177</u>	#178 ##
整271	8272	8273	8274 芙	影75		8277	整278
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	惠 8074	聵	最	職	鴉
建	腲	腳	腴	腵	腶	腷	腸
駅	8172	*E	#色	*************************************	豐色	豐色	#H
整71	1 第272	8273		悲		荷	**************************************
刻71	8372 菜	威	8374	8375	8376	## 章	8378

- 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

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

U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

簳	聲	聳	聴	聵	虛	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	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
- 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	惠 8074	聵	最	職	鴉
建	腲	腳	腴	腵	腶	腷	腸
駅	8172	*E	#色	*************************************	豐色	豐色	#H
整71	1 第272	8273		悲		荷	**************************************
刻71	8372 菜	威	8374	8375	8376	## 章	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

警	聲	聳	惠 8074	聵	最	職	鴉
健	腲	腳	腴	腵	腶	腷	腸
製	8172	*E	#E	絶	<u>8176</u>	#E	8178 44
整271	8272	8273	接	悲		8277	**************************************
刻71	8372 菜	威	8374	8375	8376	8377	8378



Representing Strings: the Unicode Standard

- 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	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	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





Representing Strings: the Unicode Standard

- 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 8073	题 8074	聵 8075	最 8076	職	鴉
建	腲	腳	<u>腴</u> 8174	服	服 8176	届	腸
製	色 8272	丰色		艵	艶	豐色	## 8278
笔	里 8372	式 8373	荴	惹	荶	荷 8377	学 8378
葱	慕	葳	蔵	葵	葶	葷	恵

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





(Demo)



[Demo]

[Demo]

First example in the course of an object changing state

[Demo]

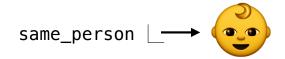
First example in the course of an object changing state

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

[Demo]

First example in the course of an object changing state

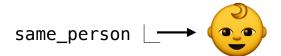
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

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

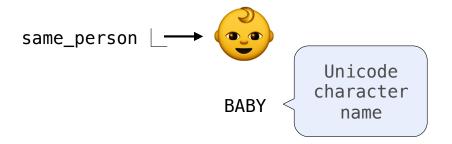


BABY

[Demo]

First example in the course of an object changing state

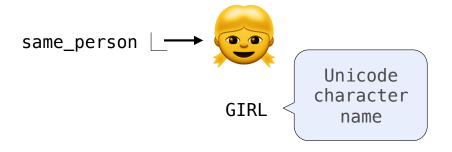
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

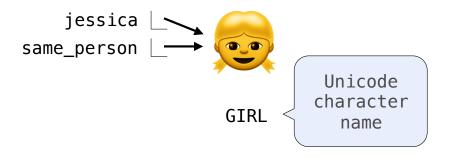
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

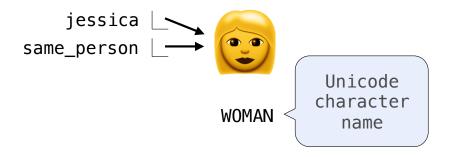
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

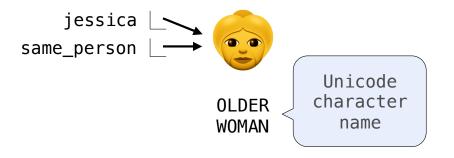
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

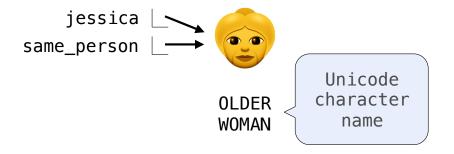
The same object can change in value throughout the course of computation



[Demo]

First example in the course of an object changing state

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

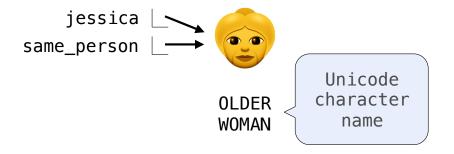


All names that refer to the same object are affected by a mutation

[Demo]

First example in the course of an object changing state

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

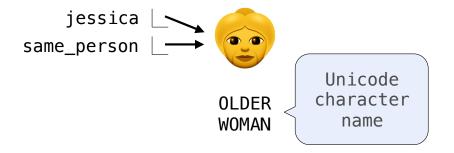


All names that refer to the same object are affected by a mutation Only objects of *mutable* types can change: lists & dictionaries

[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

aram

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

Tuples

(Demo)

Tuples are Immutable Sequences	

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

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

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

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)

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

>>> turtle
['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)
```

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)
```

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

>>> 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]
>>> ooze()
>>> turtle
['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)
```

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

```
>>> x = 2

>>> x + x

4

>>> x = 3

>>> x = 1, 2]

>>> x + x

(1, 2, 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)
```

```
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

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)

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

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

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

```
Name change:

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

4
>>> x = [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
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)

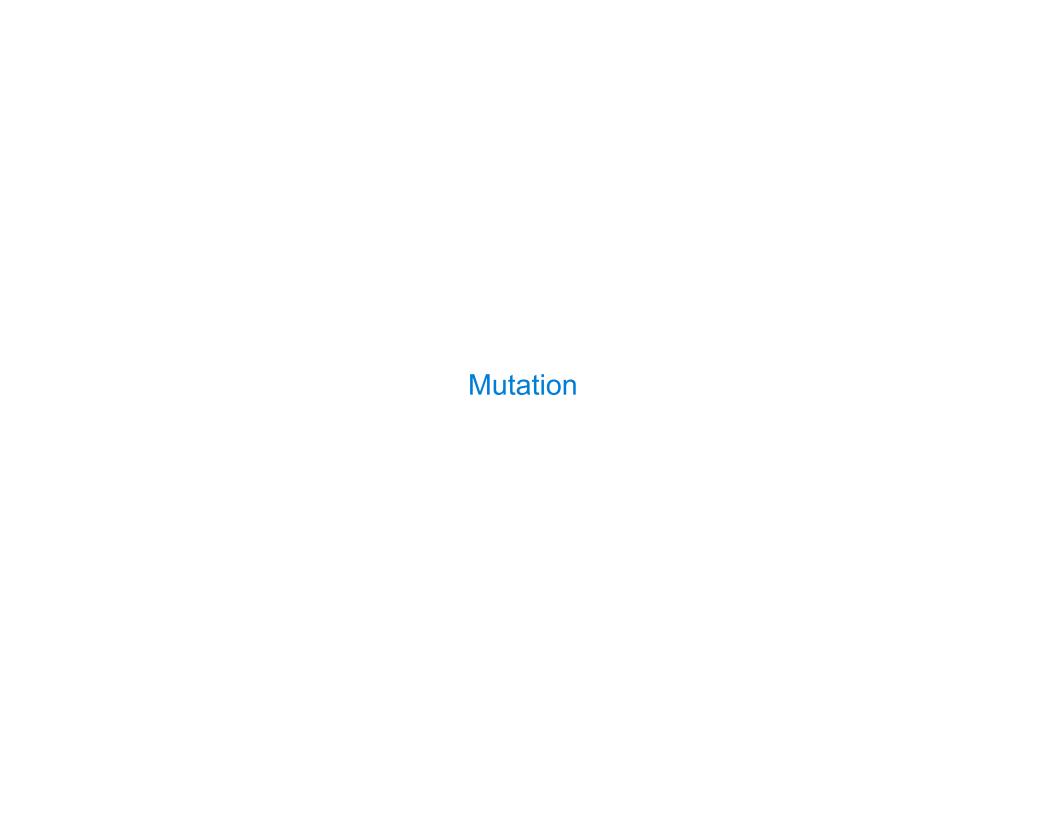
>>> turtle = [1, 2, 3]
>>> ooze()
>>> turtle
['Anything could be inside!']
```

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
```



ameness and C	· · · · · · · · · · · · · · · · · · ·	 	 	

• 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 == 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
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

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

Mutable	Default A	Arguments	are	Dangerou	S

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(3)
... return len(s)
```

16

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

16

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

16

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=[]):
                                  Global frame
                                                                →func f(s) [parent=Global]
          s append(3)
                                                    f •
                                                                 list
          return len(s)
                                                                  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
                                                            >func f(s) [parent=Global]
>>> def f(s=[]):
         s append(3)
                                                 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>