CS 61A Lecture 13

Wednesday, October 1

Announcements

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm
- Optional Hog Contest Due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm
- Optional Hog Contest Due Wednesday 10/1 @ 11:59pm
- Project 2 Due Thursday 10/9 @ 11:59pm

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm
- Optional Hog Contest Due Wednesday 10/1 @ 11:59pm
- Project 2 Due Thursday 10/9 @ 11:59pm
  - Project party Monday 10/6, 6pm-8pm in location TBD

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm
- Optional Hog Contest Due Wednesday 10/1 @ 11:59pm
- Project 2 Due Thursday 10/9 @ 11:59pm
  - Project party Monday 10/6, 6pm–8pm in location TBD
- Special event on Tuesday 10/14 @ 7pm in Wheeler:
  Fireside chat with Founder & CEO of DropBox Drew Houston, hosted by John

Announcements

- Homework 3 Due Wednesday 10/1 @ 11:59pm
- Optional Hog Contest Due Wednesday 10/1 @ 11:59pm
- Project 2 Due Thursday 10/9 @ 11:59pm
  - Project party Monday 10/6, 6pm-8pm in location TBD
- Special event on Tuesday 10/14 @ 7pm in Wheeler:
  Fireside chat with Founder & CEO of DropBox Drew Houston, hosted by John
- You can submit questions, and I'll ask them: http://goo.gl/HtkXFf

Dictionaries

```
{'Dem': 0}
```

Limitations on Dictionaries

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary **cannot be** a list or a dictionary (or any *mutable type*)
- Two **keys cannot be equal;** There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value

Linked Lists

Linked List Data Abstraction

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a linked list s."""


def rest(s):
    """Return the rest of the elements of a linked list s."""
```

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a linked list s."""


def rest(s):
    """Return the rest of the elements of a linked list s."""
```

Behavior condition(s):

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a linked list s."""


def rest(s):
    """Return the rest of the elements of a linked list s."""
```

Behavior condition(s):

If a linked list s is constructed from a first element a and a linked list b, then

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a linked list s."""


def rest(s):
    """Return the rest of the elements of a linked list s."""
```

Behavior condition(s):

If a linked list s is constructed from a first element a and a linked list b, then

- first(s) returns a, which is an element of the sequence

Linked List Data Abstraction

Constructor:

```
def link(first, rest):
    """Construct a linked list from its first element and the rest."""
```

Selectors:

```
def first(s):
    """Return the first element of a linked list s."""


def rest(s):
    """Return the rest of the elements of a linked list s."""
```

Behavior condition(s):

If a linked list s is constructed from a first element a and a linked list b, then

- first(s) returns a, which is an element of the sequence
- rest(s) returns b, which is a linked list

Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.

Implementing Recursive Lists with Pairs

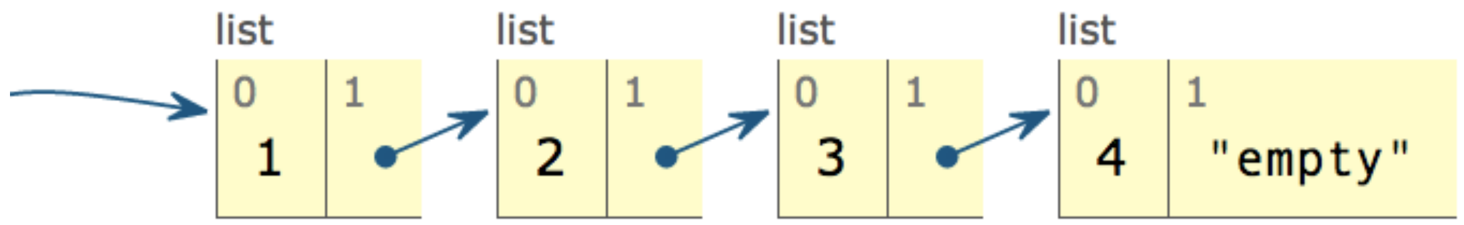
```
We can implement linked lists as pairs.  We'll use two-element lists to represent pairs.
```

1 , 2 , 3 , 4

Implementing Recursive Lists with Pairs

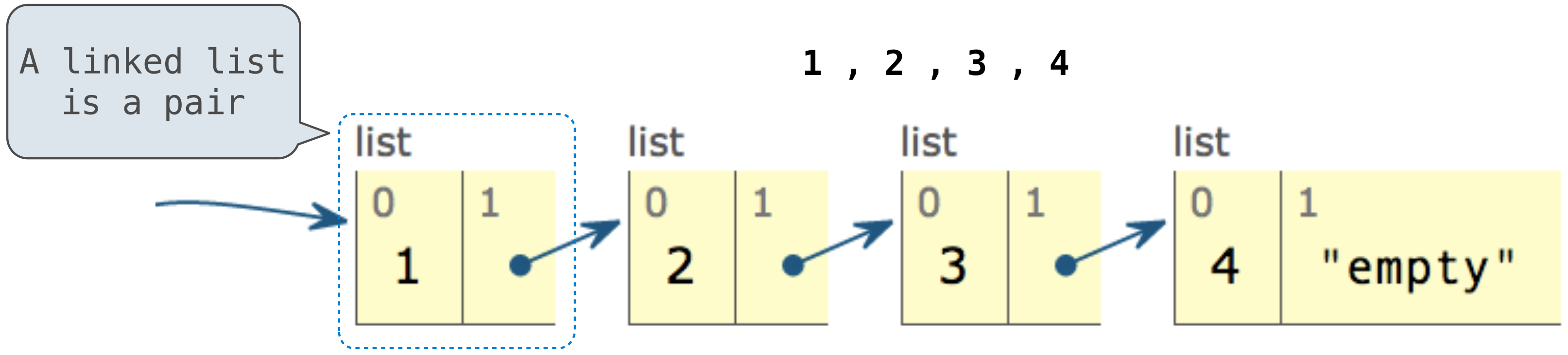
```
We can implement linked lists as pairs.  We'll use two-element lists to represent pairs.
```

1 , 2 , 3 , 4



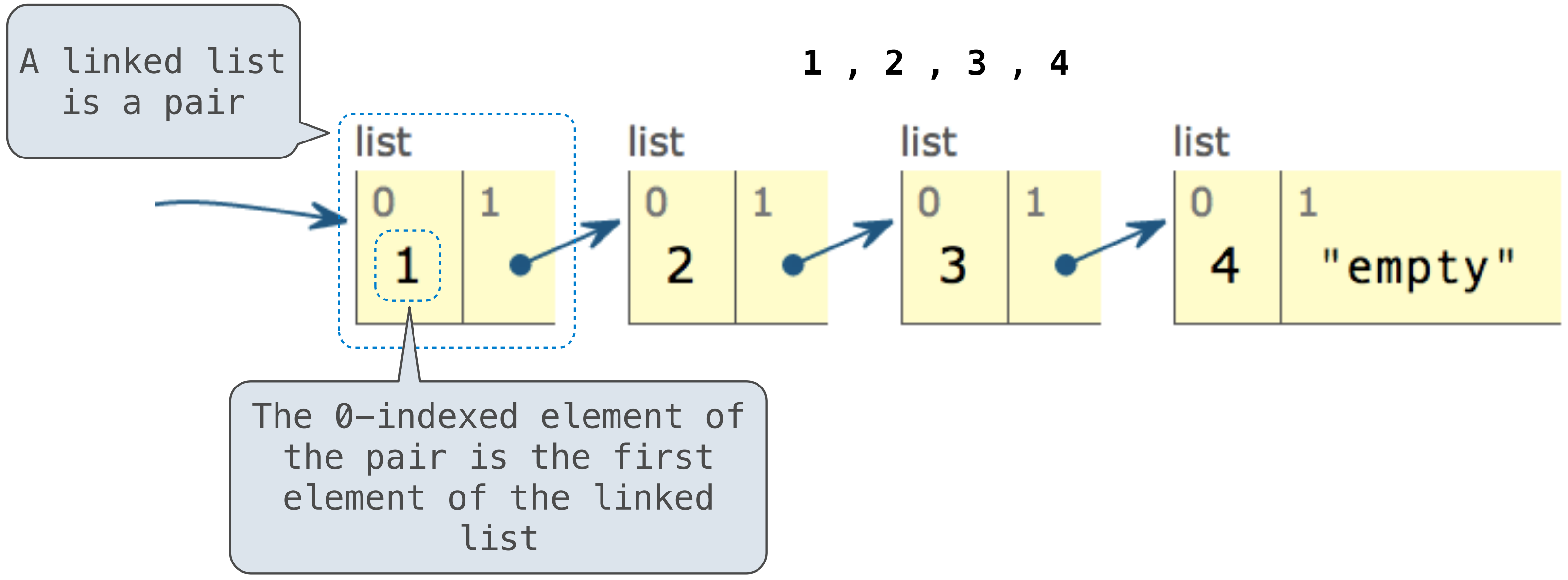
Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.



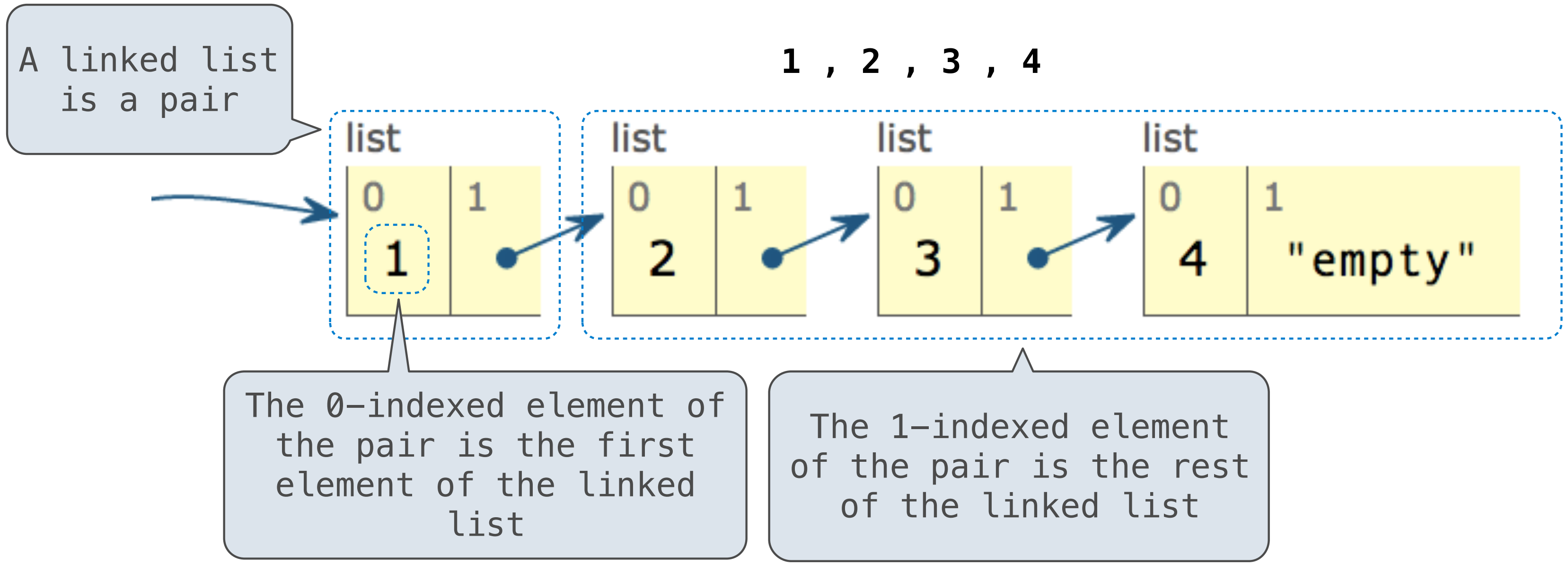
Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.


A linked list is a pair
1 , 2 , 3 , 4
list
0
1
1
list
0
1
2
list
0
1
3
list
0
1
4
"empty"
The 0-indexed element of the pair is the first element of the linked list

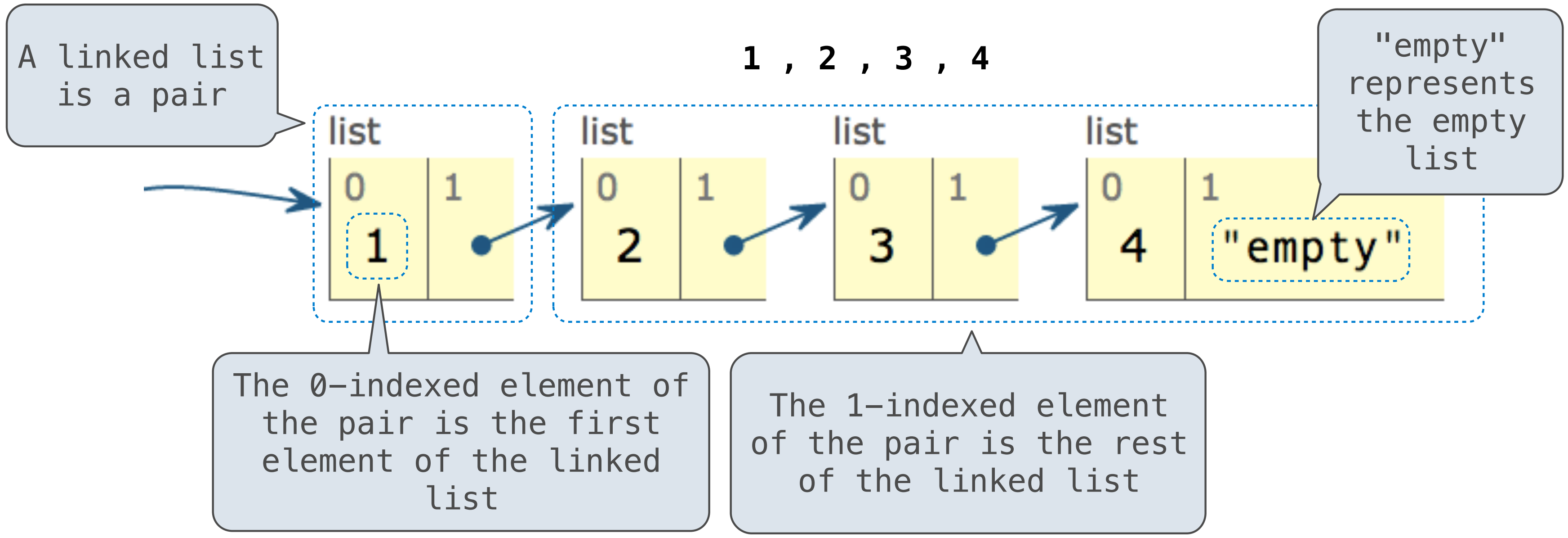
Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.


A linked list is a pair
1 , 2 , 3 , 4
list
0
1
1
list
0
1
2
list
0
1
3
list
0
1
4
"empty"
The 0-indexed element of the pair is the first element of the linked list
The 1-indexed element of the pair is the rest of the linked list

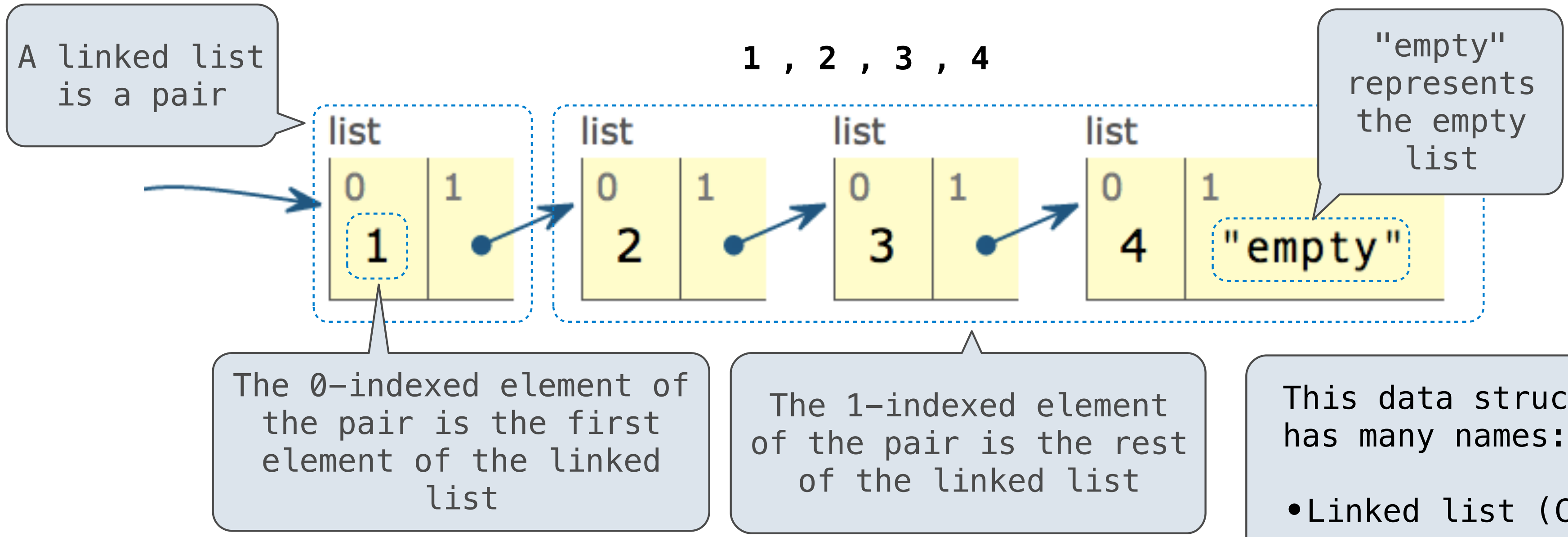
Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.


A linked list is a pair
1 , 2 , 3 , 4
"empty" represents the empty list
list
0
1
1
list
0
1
2
list
0
1
3
list
0
1
4
"empty"
The 0-indexed element of the pair is the first element of the linked list
The 1-indexed element of the pair is the rest of the linked list

Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.



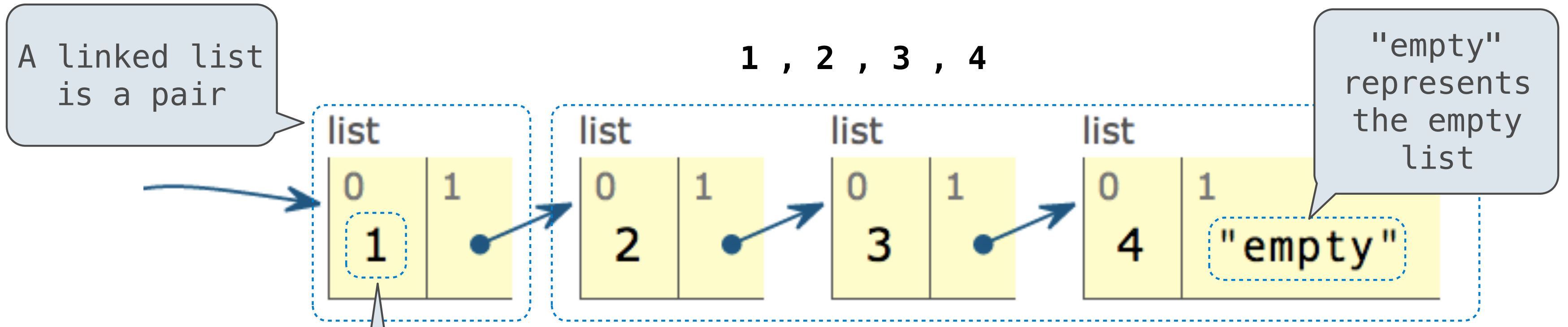
A linked list is a pair
1 , 2 , 3 , 4
list
0
1
1
list
0
1
2
list
0
1
3
list
0
1
4
"empty"
"empty" represents the empty list
The 0-indexed element of the pair is the first element of the linked list
The 1-indexed element of the pair is the rest of the linked list

This data structure has many names:

- Linked list (C, Java)
- List (Lisp)
- Forward list (C++)

Implementing Recursive Lists with Pairs

We can implement linked lists as pairs. We'll use two-element lists to represent pairs.



The 0-indexed element of the pair is the first element of the linked list

The 1-indexed element of the pair is the rest of the linked list

This data structure has many names:

- Linked list (C, Java)
- List (Lisp)
- Forward list (C++)

(Demo)

Sequence Abstraction Implementation

Implementing the Sequence Abstraction

Implementing the Sequence Abstraction

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_link(s):
    """Return the length of linked list s."""
    length = 0
    while s != empty:
        s, length = rest(s), length + 1
    return length
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_link(s):
    """Return the length of linked list s."""
    length = 0
    while s != empty:
        s, length = rest(s), length + 1
    return length

def getitem_link(s, i):
    """Return the element at index i of linked list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

Implementing the Sequence Abstraction

```
def len_link(s):
    """Return the length of linked list s."""
    length = 0
    while s != empty:
        s, length = rest(s), length + 1
    return length

def getitem_link(s, i):
    """Return the element at index i of linked list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

(Demo)

Interactive Diagram

Recursive implementations

(Demo)

Linked List Processing

```
extend
reverse
apply_to_all_link
join_link
partitions
print_partitions
```

(Demo)

Rooted Trees

Rooted Trees Have a Value at the Root of Every Tree

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value

```
[[1, 2], [3, 4], 5]
```

Rooted Trees Have a Value at the Root of Every Tree

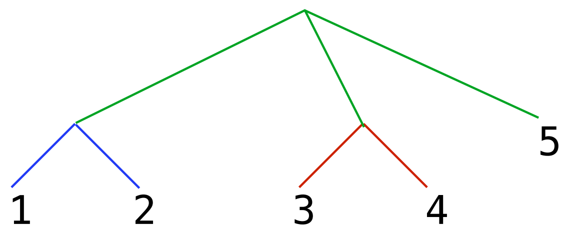
Previously, trees ***either*** had branches ***or*** they were a leaf value

```
[[1, 2], [3, 4], 5]
```

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value

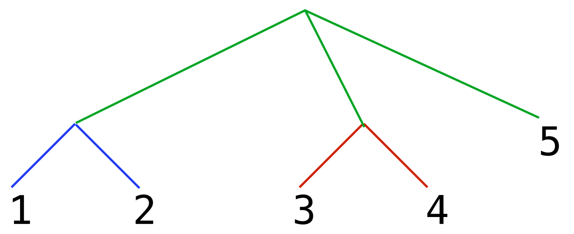
```
[[1, 2], [3, 4], 5]
```



Rooted Trees Have a Value at the Root of Every Tree

`Previously, trees` ***`either`*** `had branches` ***`or`*** `they were a leaf value; Rooted trees have` **`both`**

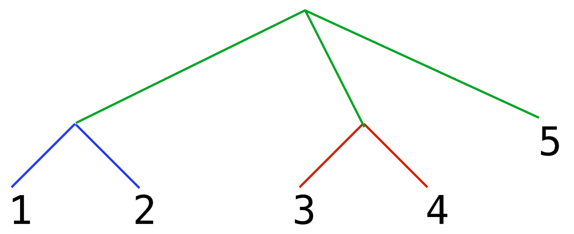
```
[[1, 2], [3, 4], 5]
```

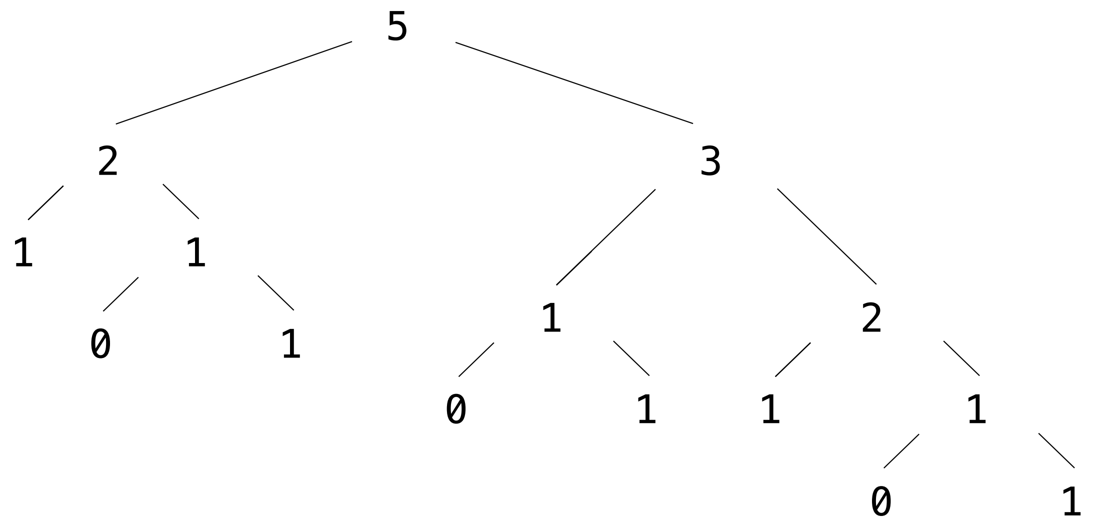


Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

```
[[1, 2], [3, 4], 5]
```

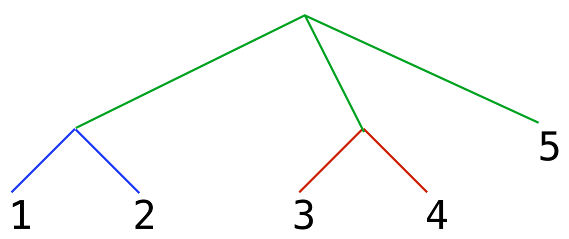


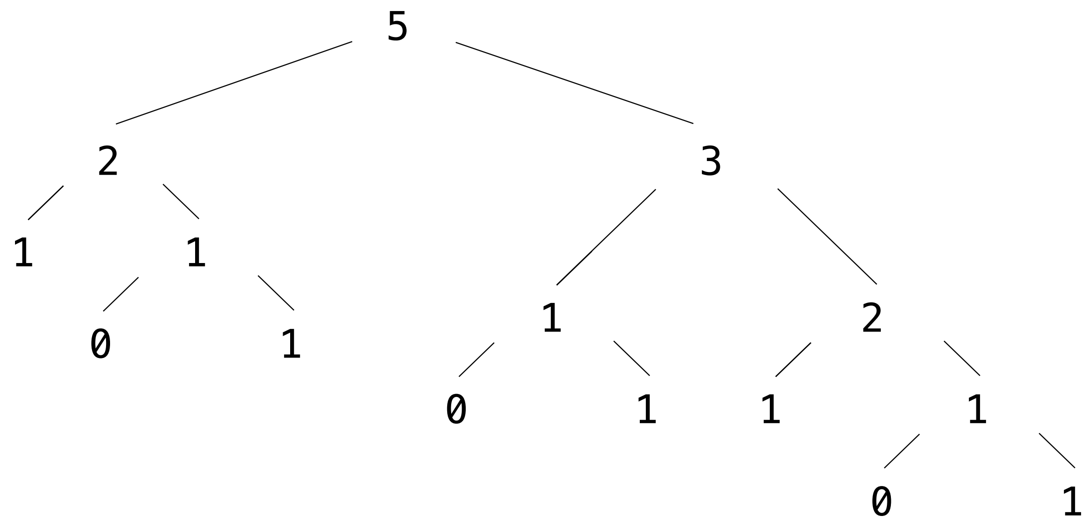


Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

[[1, 2], [3, 4], 5]



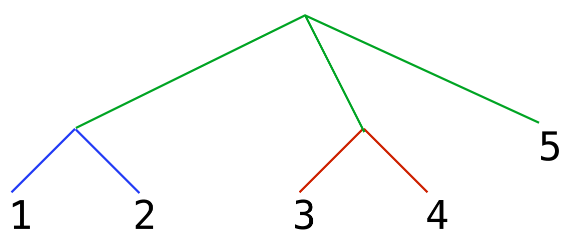


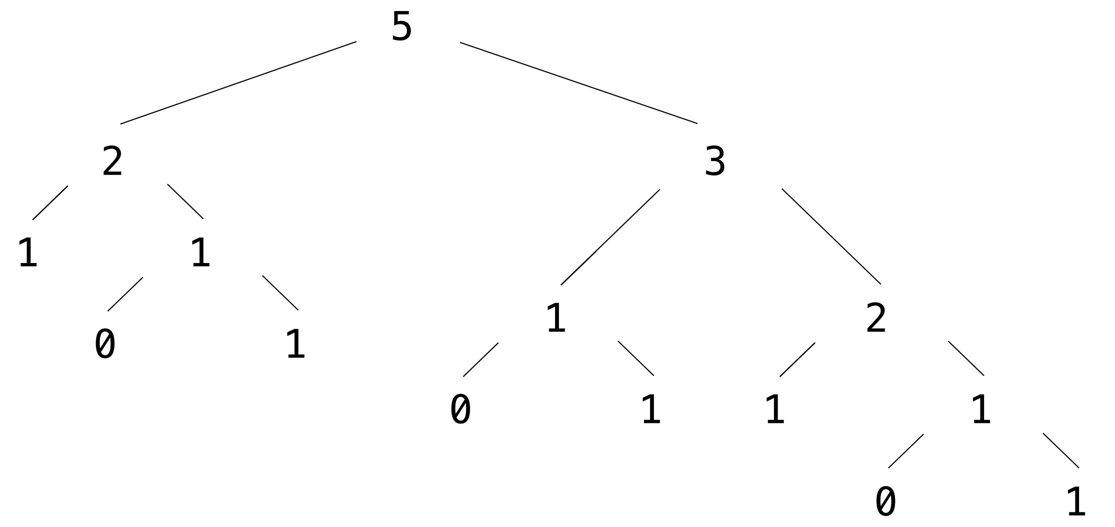
A rooted tree has a root value and a sequence of branches, which are rooted trees

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

`[[1, 2], [3, 4], 5]`





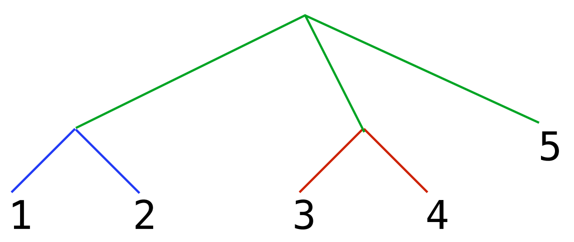
A rooted tree has a root value and a sequence of branches, which are rooted trees

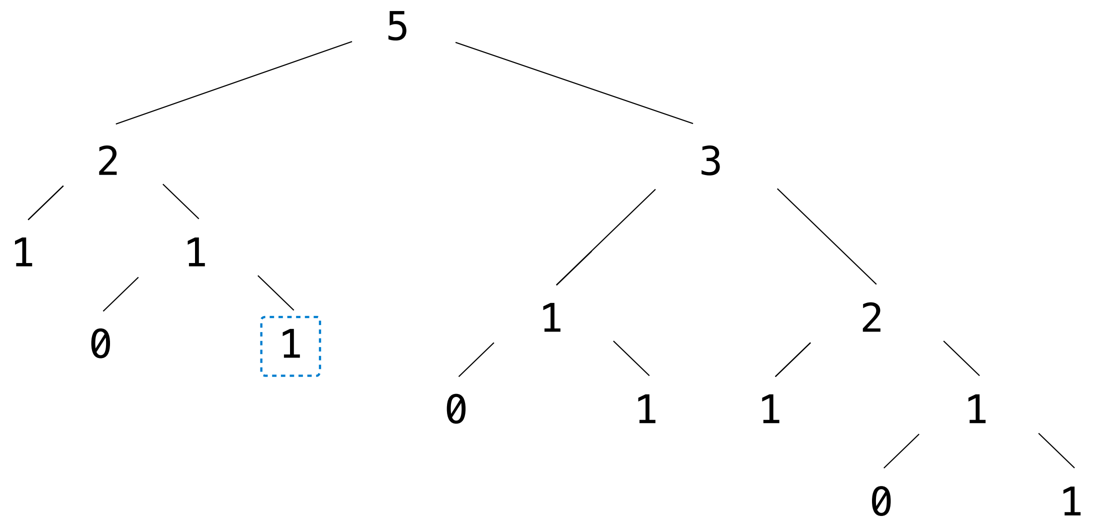
A rooted tree with zero branches is called a leaf

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

`[[1, 2], [3, 4], 5]`





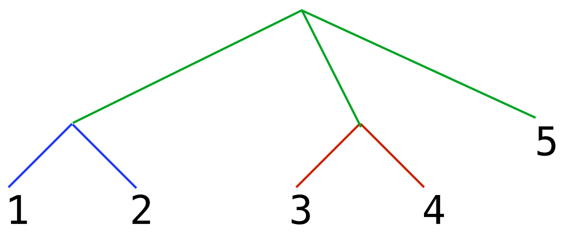
A rooted tree has a root value and a sequence of branches, which are rooted trees

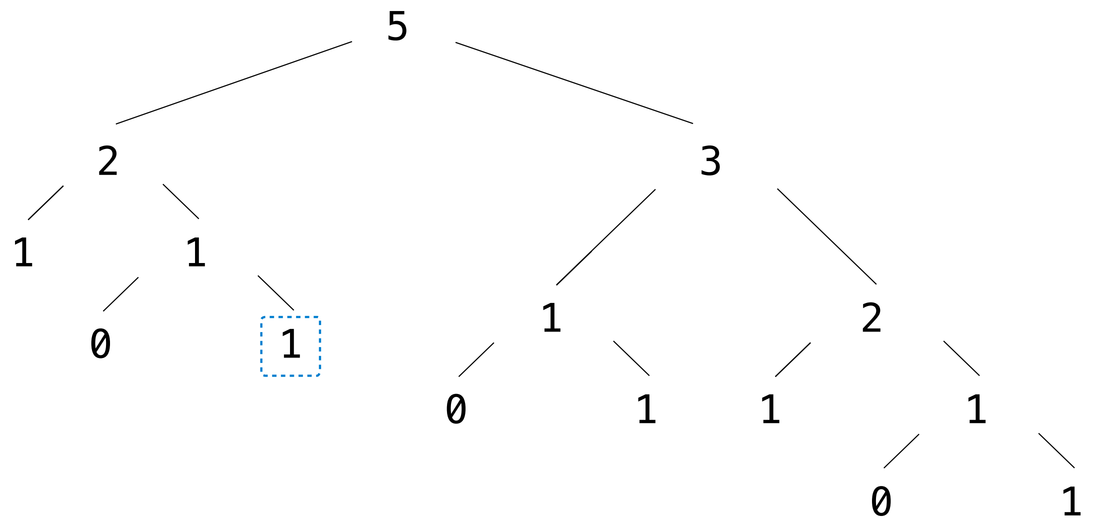
A rooted tree with zero branches is called a leaf

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

[[1, 2], [3, 4], 5]





A rooted tree has a root value and a sequence of branches, which are rooted trees

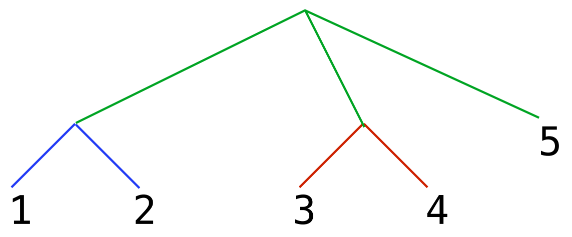
A rooted tree with zero branches is called a leaf

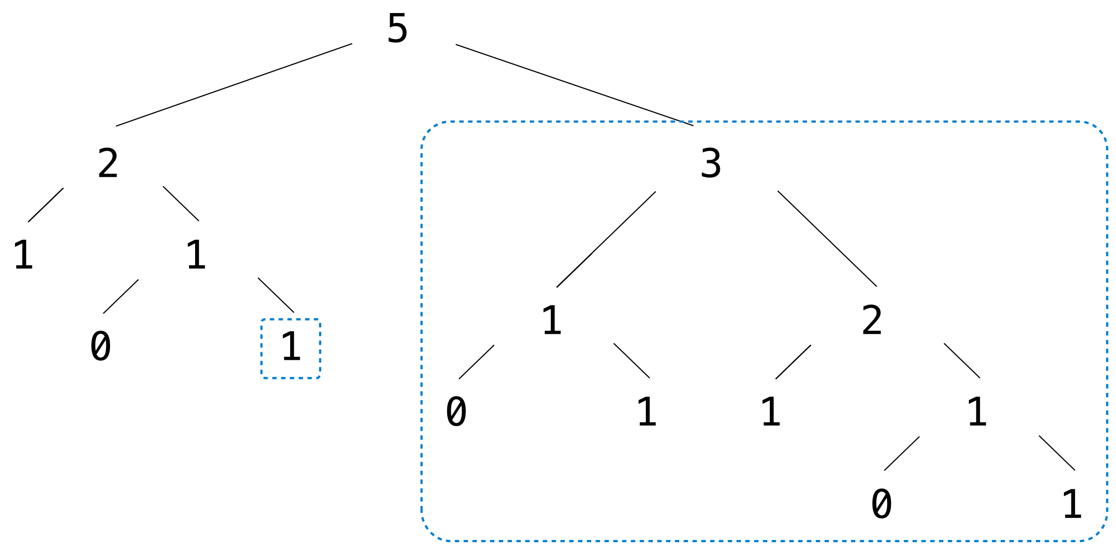
The root values of sub-trees within a rooted tree are often called node values or nodes

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

`[[1, 2], [3, 4], 5]`





A rooted tree has a root value and a sequence of branches, which are rooted trees

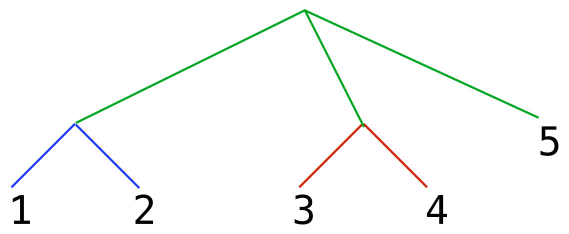
A rooted tree with zero branches is called a leaf

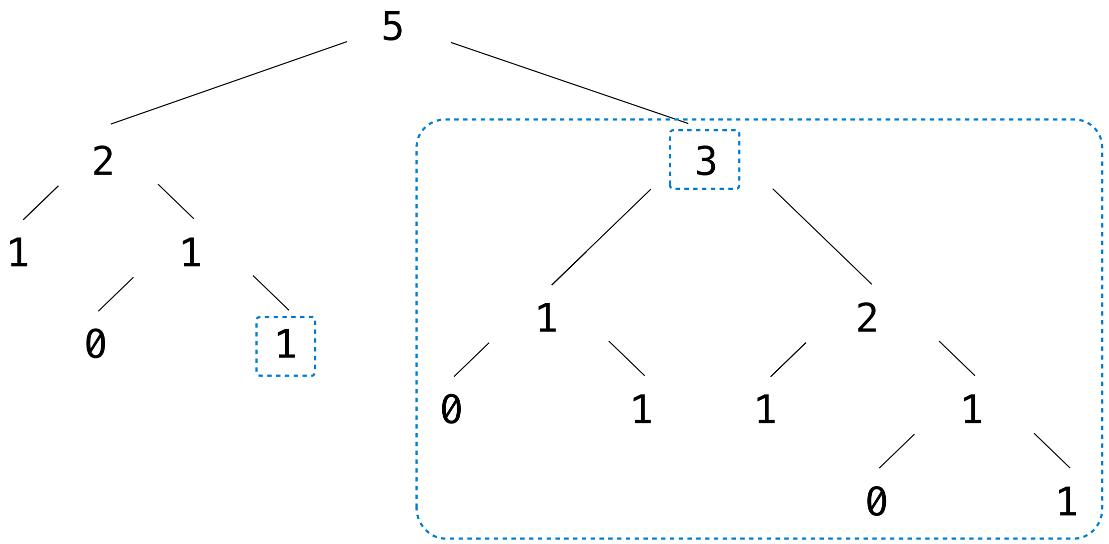
The root values of sub-trees within a rooted tree are often called node values or nodes

Rooted Trees Have a Value at the Root of Every Tree

Previously, trees ***either*** had branches ***or*** they were a leaf value; Rooted trees have **both**

[[1, 2], [3, 4], 5]





A rooted tree has a root value and a sequence of branches, which are rooted trees

A rooted tree with zero branches is called a leaf

The root values of sub-trees within a rooted tree are often called node values or nodes

Implementing the Rooted Tree Abstraction

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

Implementing the Rooted Tree Abstraction

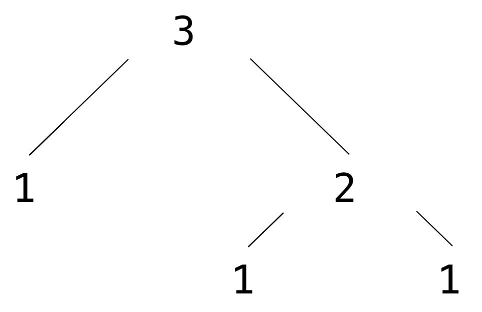
```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

```
A rooted tree has a root value
 and a sequence of branches,
 which are each rooted trees
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

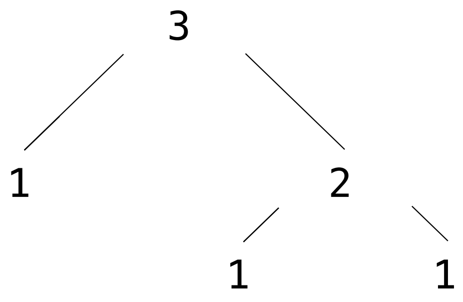
A rooted tree has a root value
and a sequence of branches,
which are each rooted trees



Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

```
A rooted tree has a root value
and a sequence of branches,
which are each rooted trees
```

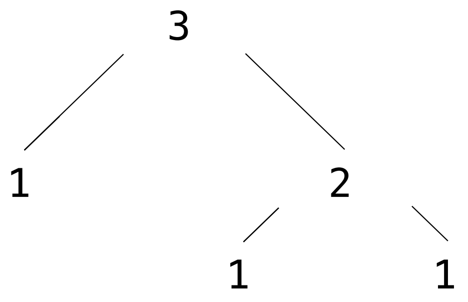


```
>>> rooted(3, [rooted(1, []),
...            rooted(2, [rooted(1, []),
...                       rooted(1, [])])])
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

A rooted tree has a root value
and a sequence of branches,
which are each rooted trees

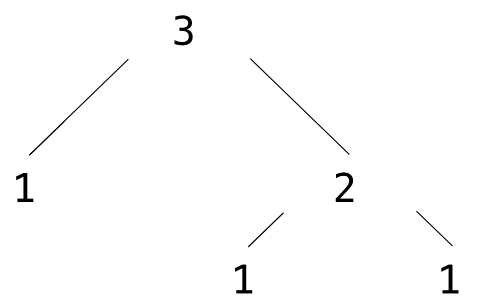


```
>>> rooted(3, [rooted(1, []),
...                rooted(2, [rooted(1, []),
...                           rooted(1, [])])])
[3, [1], [2, [1], [1]]]
```

Implementing the Rooted Tree Abstraction

```
def rooted(value, branches):
    for branch in branches:
        assert is_rooted(branch)
    return [value] + list(branches)

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_rooted(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_rooted(branch):
            return False
    return True
```

A rooted tree has a root value and a sequence of branches, which are each rooted trees



```
>>> rooted(3, [rooted(1, []),
...            rooted(2, [rooted(1, []),
...                       rooted(1, [])])])
[3, [1], [2, [1], [1]]]
```

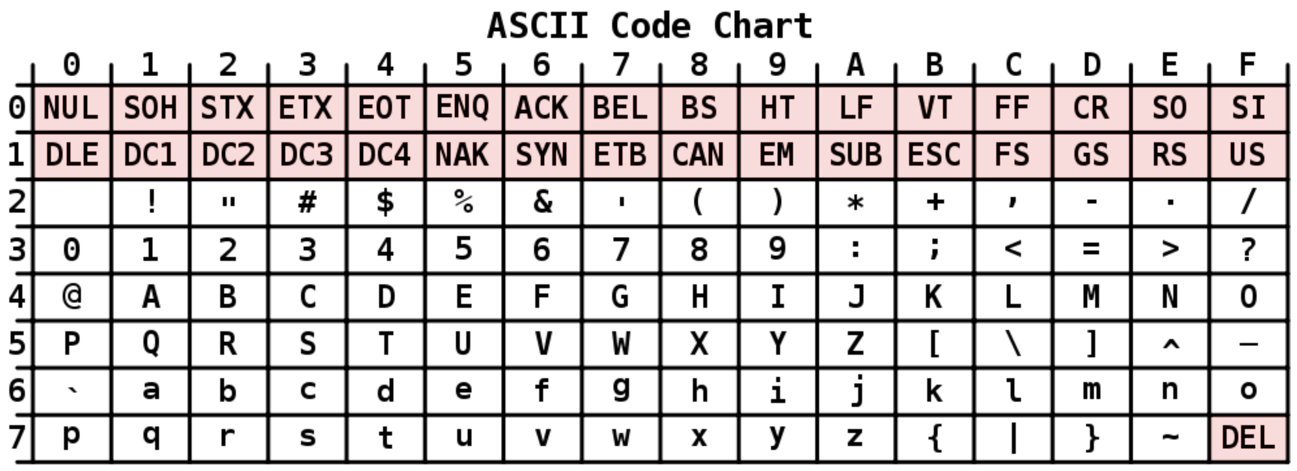
(Demo)

Encoding Strings

(Bonus Material)

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

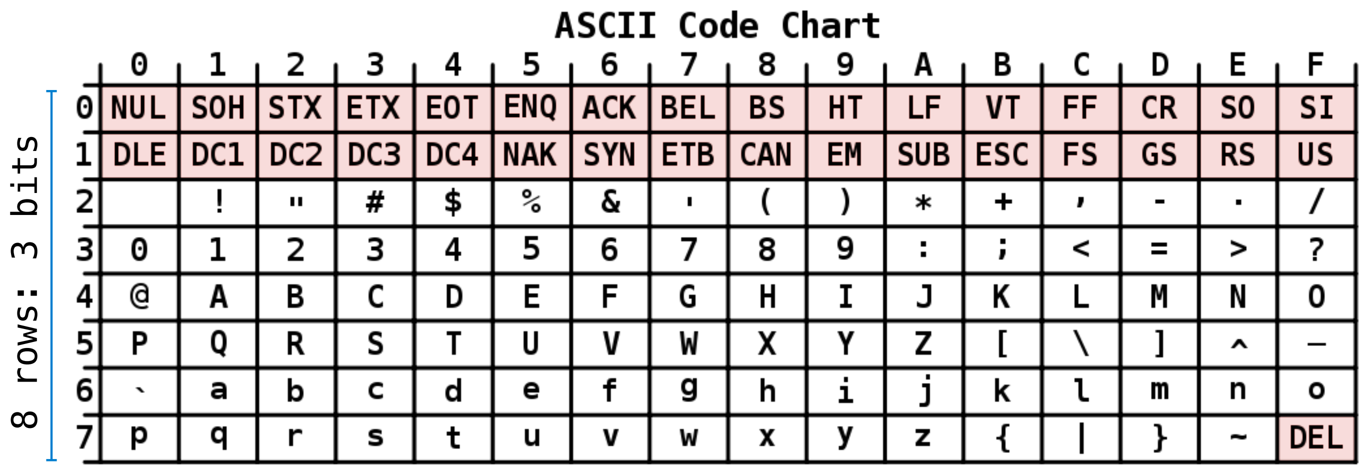


ASCII Code Chart

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

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange



ASCII Code Chart

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

8 rows: 3 bits

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

ASCII Code Chart

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

8 rows: 3 bits

16 columns: 4 bits

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

ASCII Code Chart

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

8 rows: 3 bits

16 columns: 4 bits

- Layout was chosen to support sorting by character code

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

ASCII Code Chart

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

8 rows: 3 bits

16 columns: 4 bits

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

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

ASCII Code Chart

8 rows: 3 bits

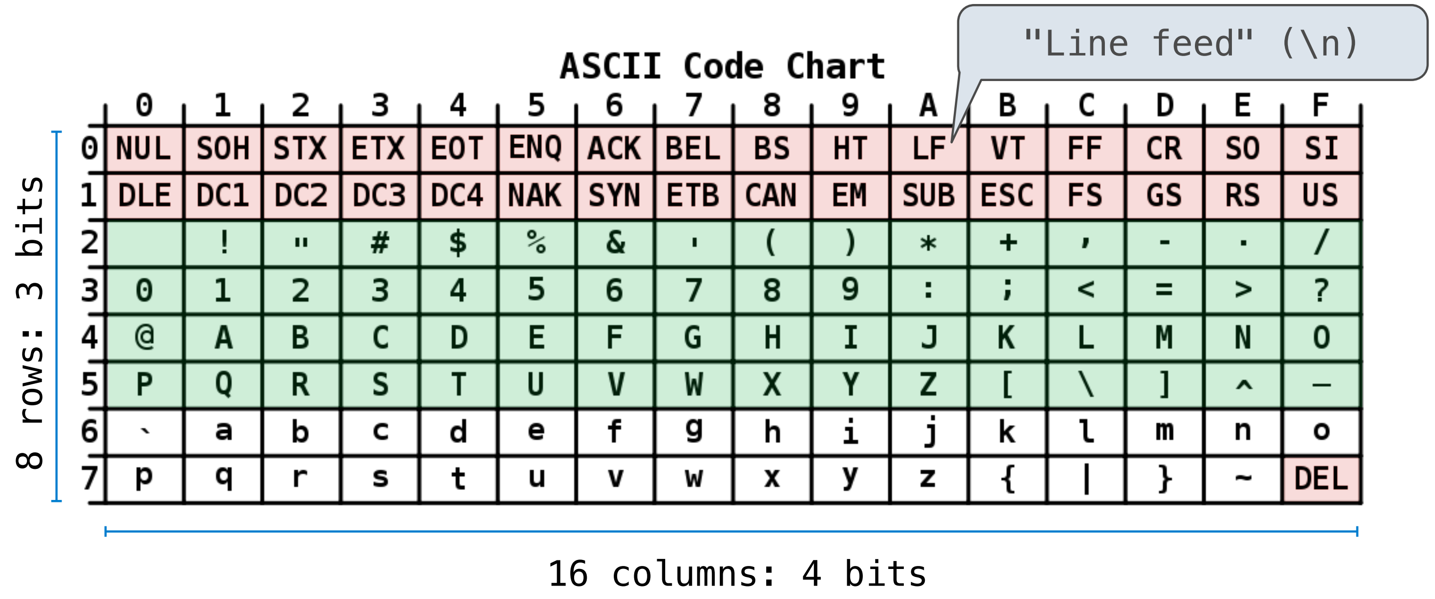
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f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

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange


"Line feed" (\n)
8 rows: 3 bits
16 columns: 4 bits

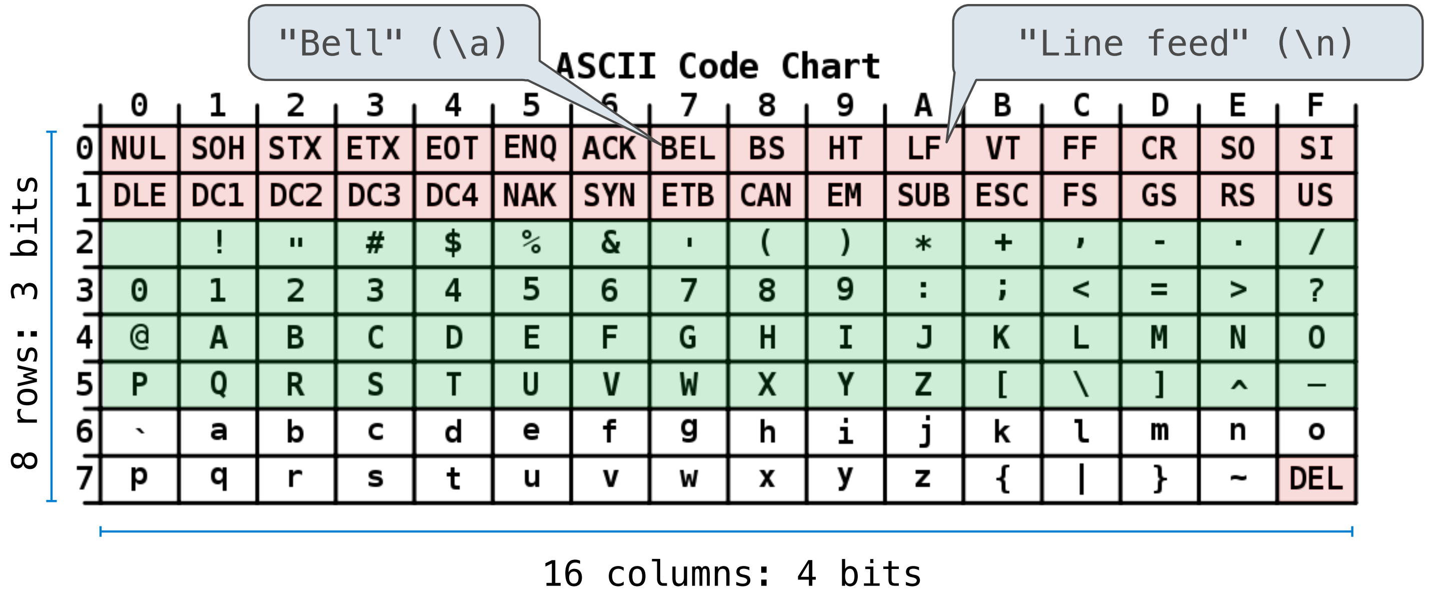
ASCII Code Chart

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

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange

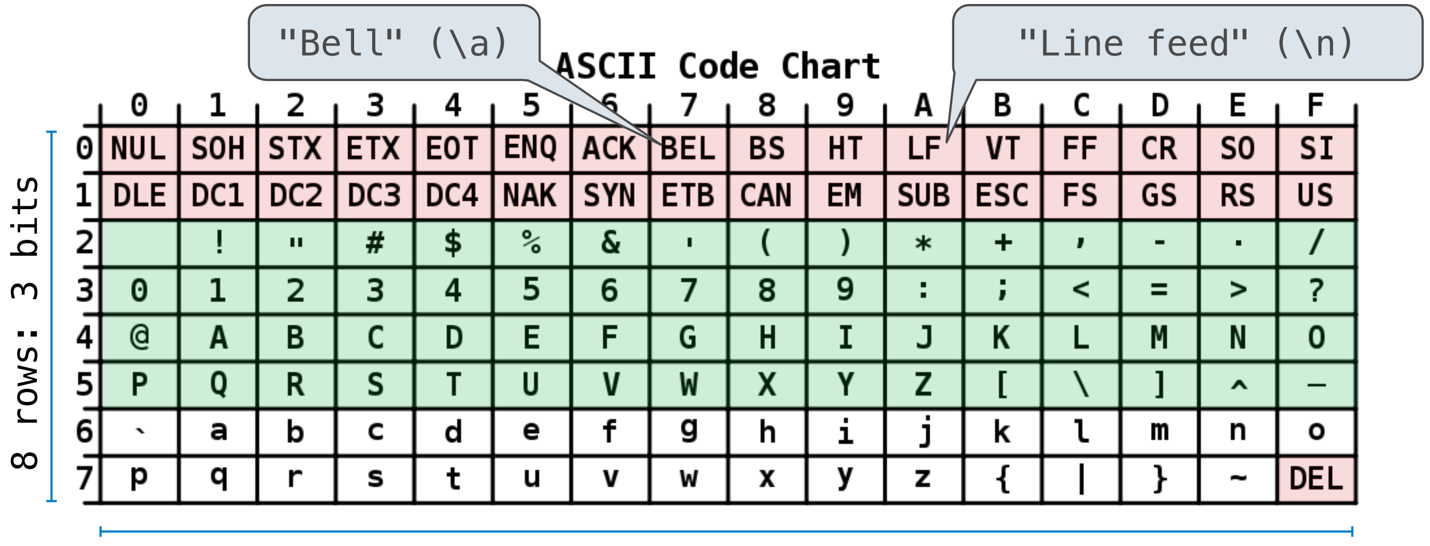

"Bell" (\a)
"Line feed" (\n)
ASCII Code Chart
8 rows: 3 bits
16 columns: 4 bits

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

Representing Strings: the ASCII Standard

American Standard Code for Information Interchange


"Bell" (\a)
"Line feed" (\n)
ASCII Code Chart
8 rows: 3 bits
16 columns: 4 bits

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

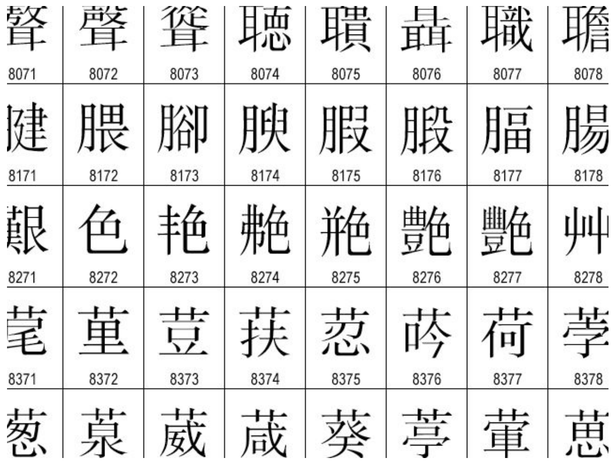
16 columns: 4 bits

- 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)

Representing Strings: the Unicode Standard

Representing Strings: the Unicode Standard

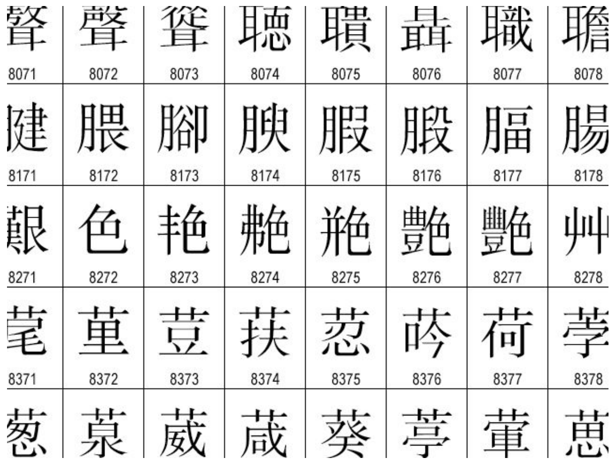


聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

Representing Strings: the Unicode Standard

- 109,000 characters



聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	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

- 109,000 characters
- 93 scripts (organized)

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

Representing Strings: the Unicode Standard

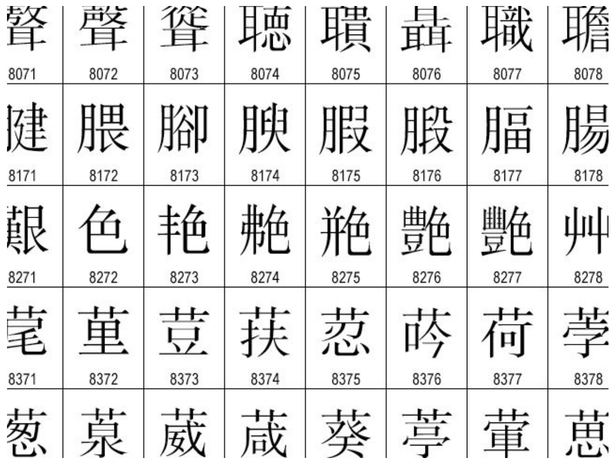
- 109,000 characters
- 93 scripts (organized)
- Enumeration of character properties, such as case

8071
8072
8073
8074
8075
8076
8077
8078
8171
8172
8173
8174
8175
8176
8177
8178
8271
8272
8273
8274
8275
8276
8277
8278
8371
8372
8373
8374
8375
8376
8377
8378

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

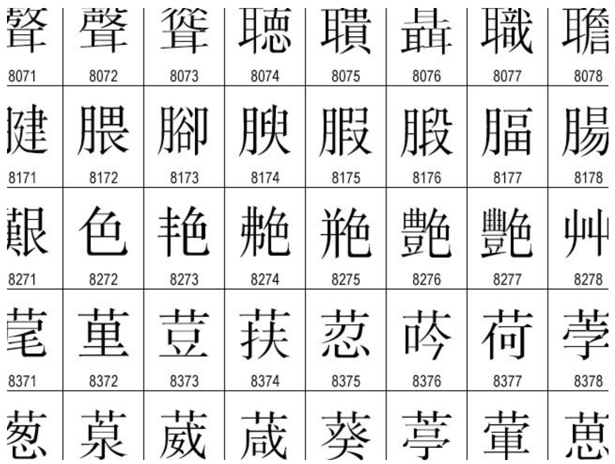


聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

Representing Strings: the Unicode Standard

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

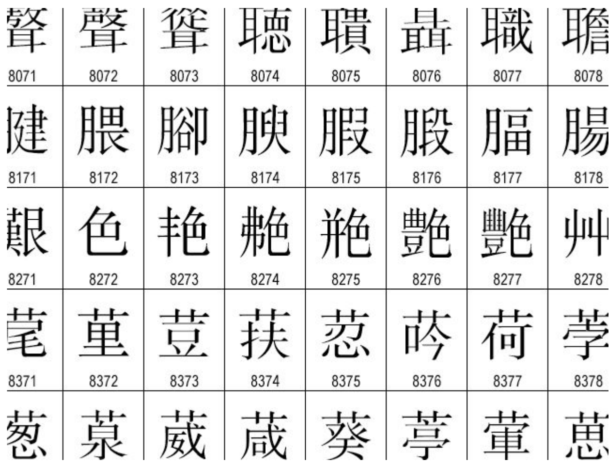


聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

Representing Strings: the Unicode Standard

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



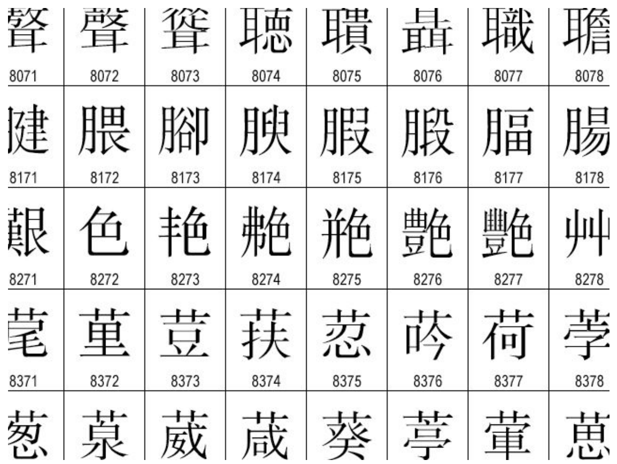
聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

```
U+0058 LATIN CAPITAL LETTER X
```

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



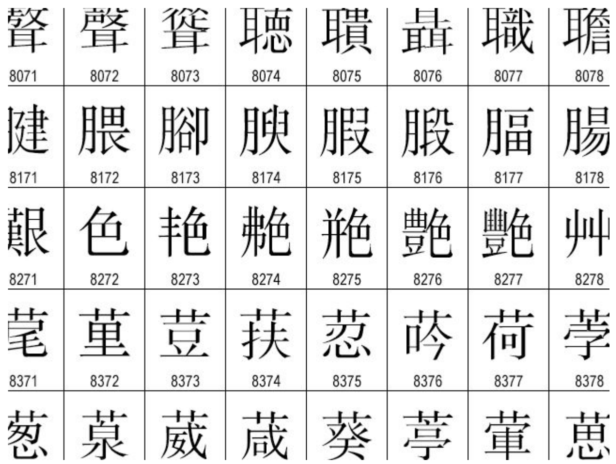
聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

```
U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE
```

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



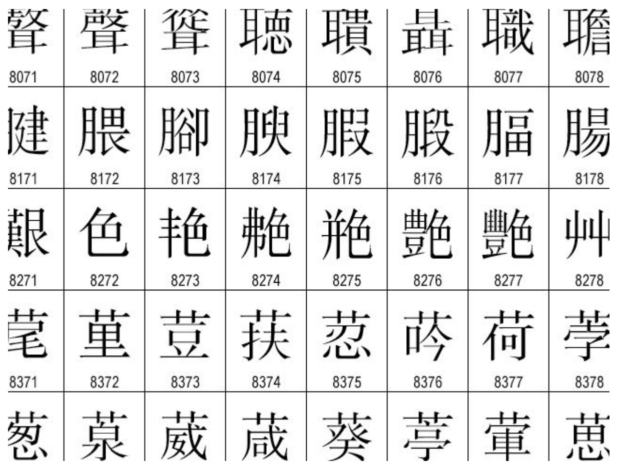
聱	聲	聳	聴	聵	聶	職	聸
8071	8072	8073	8074	8075	8076	8077	8078
腱	腲	腳	腴	腵	腶	腷	腸
8171	8172	8173	8174	8175	8176	8177	8178
艱	色	艳	艴	艵	艶	艷	艸
8271	8272	8273	8274	8275	8276	8277	8278
荱	荲	荳	荴	荵	荶	荷	荸
8371	8372	8373	8374	8375	8376	8377	8378
葱	葲	葳	葴	葵	葶	葷	葸

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

```
U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE
```

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



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

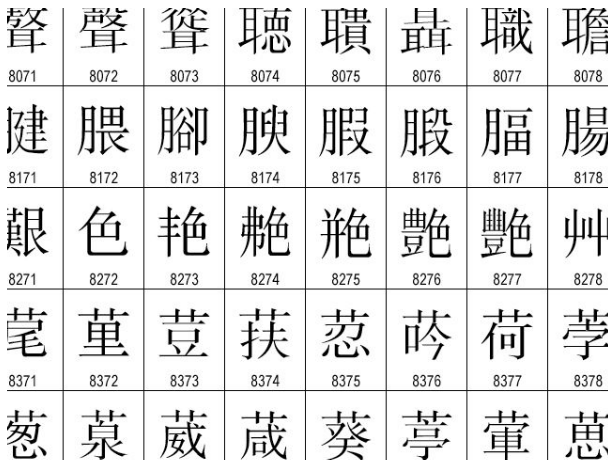
http://ian-albert.com/unicode_chart/unichart-chinese.jpg

```
U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE
```

'☺'

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



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

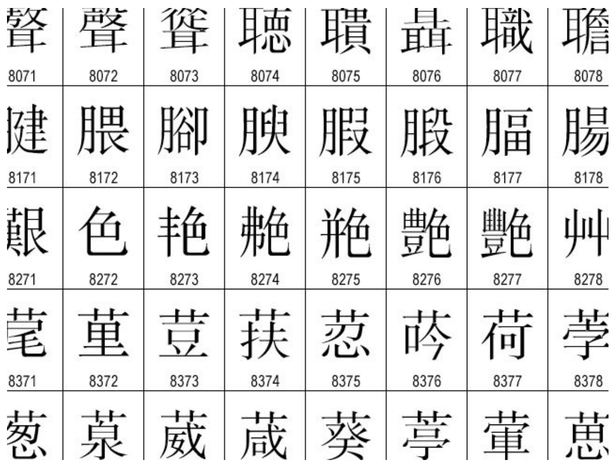
http://ian-albert.com/unicode_chart/unichart-chinese.jpg

```
U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE
```

'☺'   '☹'

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



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

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

```
U+0058 LATIN CAPITAL LETTER X

U+263a WHITE SMILING FACE

U+2639 WHITE FROWNING FACE
```

'☺'   '☹'

(Demo)

Representing Strings: UTF-8 Encoding

Representing Strings: UTF-8 Encoding

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

Representing Strings: UTF-8 Encoding

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

Unicode: Correspondence between characters and integers
```

Representing Strings: UTF-8 Encoding

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

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes
```

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes　　　　　　　　integers

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

	00000000	0	
bytes			integers

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes			integers
	00000000	0	
	00000001	1	

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes	integers
00000000	0
00000001	1
00000010	2

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes	integers
00000000	0
00000001	1
00000010	2
00000011	3

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes	integers
00000000	0
00000001	1
00000010	2
00000011	3

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

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes	integers
00000000	0
00000001	1
00000010	2
00000011	3

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

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

Representing Strings: UTF-8 Encoding

UTF (UCS (Universal Character Set) Transformation Format)

Unicode: Correspondence between characters and integers

UTF-8: Correspondence between those integers and bytes

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

bytes	integers
00000000	0
00000001	1
00000010	2
00000011	3

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

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

(Demo)