CS61A Lecture 14

Monday, September 30th, 2019

Announcements

- Cats Project Party today in Cory 241 from 6:30 to 8:00pm.
- Homework 4 is due next Thursday, worth 4 points instead of 2.

Objects

Objects are special not only because they represent some value, but because they behave like the thing in the world they are supposed to emulate.

> *from Composing Programs, Chapter 2.4*
> Objects combine data values with behavior. Objects represent information, but also behave like the things that they represent. The logic of how an object interacts with other objects is bundled along with the information that encodes the object's value. When an object is printed, it knows how to spell itself out in text. If an object is composed of parts, it knows how to reveal those parts on demand. Objects are both information and processes, bundled together to represent the properties, interactions, and behaviors of complex things.

For example:

```
>>> from datetime import date
>>> date
<class 'datetime.date'>
```

You could represent a date with just numbers, but dates should probably behave more like a real date. So, we could do this:

```
>>> today = date(2019, 9, 30)
>>> freedom = date(2019, 12, 19)
>>> freedom - today
datetime.timedelta(days=80)
```

We can also access particular aspects such as `year` :

```
>>> today.year
2019
```

You can also change how the date is displayed:

```
>>> today.strftime(''%A, $B $d')
'Monday, September 30'
```

How did that happen? Well, this info is contained within the `date` class. Again, this is abstraction: we don't need to know how `strftime` gets the day or month name, we just need to know what it takes in and returns.

Objects represent info, consist of data and behavior, bundled together to create abstractions.

Objects can represent things, properties, interactions and processes.

A type of object is called a class; classes are first-class values in Python.

Objects are a foundational aspect of object-oriented programming:

- Object-oriented programming is a metaphor for organizing large programs
- Special syntax that can improve the comppposition of programs

In Python, every value is an object

- All objects have attributes
- A lot of data manipulation happens through object methods (function-like objects or operations that work on `.` appended to the end of a variable)
- Functions only do one thing, while objects do many related things

For example, strings are objects, and we can manipulate them with methods:

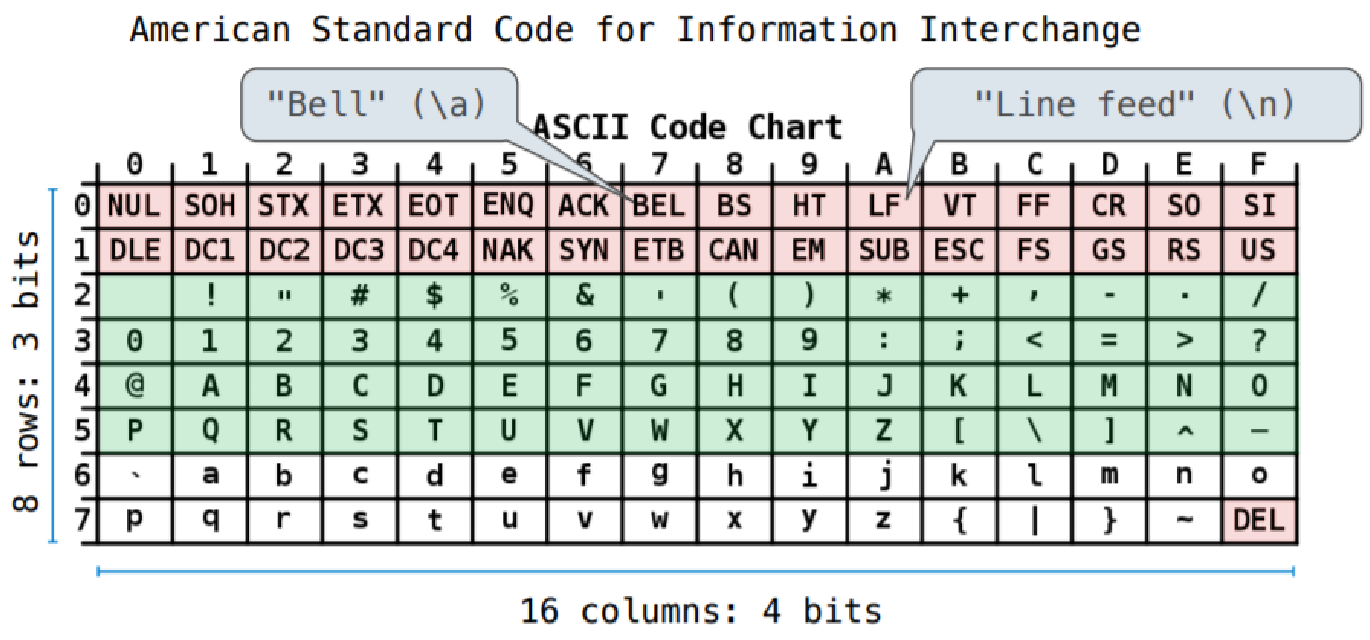
```
>>> 'Hello'.swapcase()
'hELLO'
>>> 'hello'.isalpha()
True
>>> 'hello'.upper
<built-in method...>
```

Strings

> Note: This is an optional part of the course.

ASCII Standard

Strings are an abstraction, because they are not represented by actual letters like we are used to, but instead use something called the ASCII standard.


American Standard Code for Information Interchange
"Bell" (\a)
ASCII Code Chart
"Line feed" (\n)
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

It was 8 rows of 16 columns, where they were designed to be able to be simplified to a smaller subset of 64 characters to get the most useful characters only.

The control characters were designed for transmission, for early machines that needed this. For example, `LF` is line feed, which tells a printer to move the paper up. They still exist in modern programming languages in other forms: `LF` is now `\n` .

However, the ASCII Standard could only represent Latin script, so another standard had to be developed.

Unicode Standard

The Unicode Standard can now represent 137,994 characters as of Unicode 12.1, in more than 150 scripts. They enumerate character properties such as case, support bidirectional display order. They work by giving a caonical name for every character.

```
LATIN CAPITAL LETTER A
DIE FACE-6
EIGHTH NOTE
```

We can import Unicode into Python.

Mutation

Lists in Python can be mutated using methods:

```
>>> suits = ['coin', 'string', 'myriad']
>>> original = suits
>>> suits.pop()
'myriad' #removes the last element and displays
>>> suits.remove('string')
>>> suits
['coin']
>>> suits.append('cup')
>>> suits.append('sword','shield')
>>> suits
['coin', 'cup', 'sword', 'shield']
>>> original
['coin', 'cup', 'sword', 'shield']
```

Notice how `original` is mutated even though we never reassign it to `suits`. `suits` and `original` were merely pointers to a list object, and when the object was mutated via the name `suits`, then `original` pointed to the same mutated list.

Only objects of mutable types can be mutated: lists and dictionaries.

Dictionaries can be mutated too, One notable difference is the `.pop` method, which now takes in a key, displays the value for that key, and then removes the value from the dictionary. The list `.pop` only removes the value, because its order is important. How a dictionary is ordered doesn't matter, which is why its `.pop` method is slightly different.

Functions can mutate values outside their frame. For example:

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

What was mystery?

```
def mystery(s):
    s.pop()
    s.pop()
```

The `.pop` method doesn't print 3 and 4 because the 3 and 4 are contained within the `mystery` frame, and the `mystery` frame doesn't return anything, only mutates the value.

Tuples

Typles are like lists, but are represented with () instead of []. You can slice a tuple and store multiple values within it too, and you can add to tuples and multiply them. You can use `in` to check for a value within a tuple.

The key difference is that a tuple is an immutable value. You cannot modify a tuple like this:

```
>>> t = (3,4)
>>> t[1] = 5
Error
```

And adding a tuple doesn't change it either:

```
>>> t = (3,4)
>>> t * 2
(3,4,3,4)
```

```
>>> t
(3,4)
```

This means that tuples can be used as keys in a dictionary. Since tuples are immutable, they won't change once assigned.

For example, if we did:

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

But once a tuple is assigned, the values within it cannot be changed.

```
>>> t = (1,2,3)
>>> ooze() #This can still change the binding of t as we will learn next lecture, but it
>>> t
(1,2,3)
```

Immutable values can still be mutated if they contain mutable values. Tuples can contain lists.

Sameness and Change

- As long as we never modify objects, a compound object is just the totality of its pieces
- A rational number is just its numerator and denominator
- This view is no longer valid in the presence of change
- A compound data object has an "identity" in addition to the pieces of which it is composed
- A list is still "the same" list even if we change its contents
- Conversely, we could have two lists that happen to have the same contents, but are different

For example:

```
#Case 0
>>> a = [10]
>>> b = a
>>> a.append(20)
[10,20]
>>> a
[10,20]
>>> b
[10,20]

#Case 1
>>> a = [10]
>>> b = [10]
>>> a.append(20)
>>> a
[10,20]
>>> b
[10]
```

How can we tell if the two lists point to the same object? We can use the identity operator.

```
<exp0> is <exp1> #evaluates to True if both <exp0> and <exp1> evaluate to the same objec
```

This is different from the equality operator:

```
<exp0> == <exp1> #evaluates to True if both <exp0> and <exp1> evaluate to equal values.
```

From above:

```
#Case 0
>>> a == b
True
>>> a is b
True

#Case 1
>>> a == b
True
>>> a is b
False
```