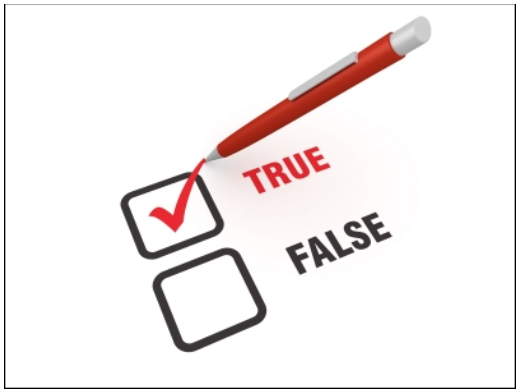
Booleans, Logical Expressions, and Predicates



CS111 Computer Programming

Department of Computer Science
Wellesley College

Concepts in this slide: Real-life examples for decision making with Boolean values.

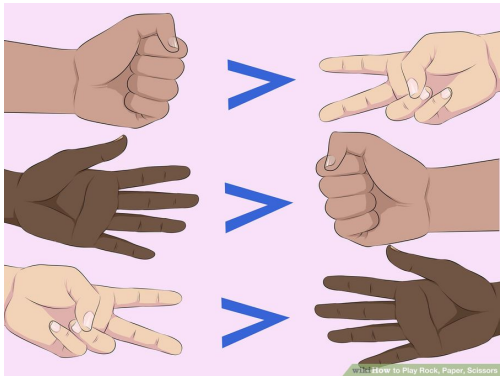
Making Decisions

If it's raining then bring umbrella and wear boots.

If timer is up, then do not cross

But first, is the condition true or false?

Concepts in this slide: Real-life examples for decision making with Boolean values.



Rock beats scissors: True

Paper beats rock: True

Scissors beats paper: True

The timer is up: False

It is raining: True

New values: Booleans

Concepts in this slide: New type: bool, and new values: True and False.

Python has two values of **`bool`** type, written **`True`** and **`False`**. The values must be capitalized.

These are called logical values or Boolean values, named after 19th century mathematician George Boole.

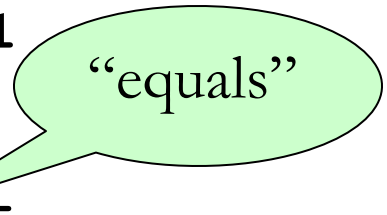
```
In [1]: True
Out[1]: True

In [2]: type(True)
Out[2]: bool

In [3]: true
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-74d9a83219ca> in <module>()
----> 1 true

NameError: name 'true' is not defined
```

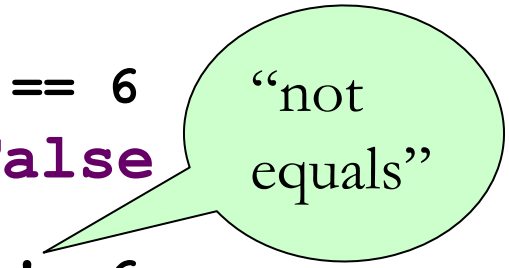
Relational Operators

Concepts in this slide:
New operators: relational.
They are: >, <, ==, !=, >=, <=

Booleans most naturally arise in the context of relational operators that compare two values.

```
In [1]: 3 < 5
Out[1]: True

In [2]: 3 < 2
Out[2]: False

In [3]: 3 > 2
Out[3]: True

In [4]: 5 <= 1
Out[4]: False

In [5]: 5 >= 1
Out[5]: True

In [6]: 5 == 5
Out[6]: True

In [7]: 5 == 6
Out[7]: False

In [8]: 5 != 6
Out[8]: True
```





Note **==** is pronounced "equals" and **!=** is pronounced "not equals". This is why we distinguish the pronunciation of the single equal sign **=** as "gets", which is assignment and nothing to do with mathematical equality!

Relational Operators [cont.]

Concepts in this slide: Relational expressions with string values.

The relational operators can also be used to compare strings (in dictionary order, meaning, something is smaller if it is earlier in the dictionary):

```
In [1]: 'bat' < 'cat'
Out[1]: True

In [2]: 'bat' < 'ant'
Out[2]: False

In [3]: 'bat' == 'bat'
Out[3]: True

In [4]: 'bat' < 'bath'
Out[4]: True

In [5]: 'Cat' < 'bat'
Out[5]: True
```

Important
If you want to compare two strings, always use the relational operators, no need to try to compare every element of the string. Python does that automatically for you.

In Python (and most other programming languages) uppercase letters come before lowercase letters in string ordering. **See Digging Deeper section about the reason.**

Logical Operators in plain English

Concepts in this slide:
New operators: logical.
They are `and, or, not`.

a:	the cake has pineapple	False
b:	the cake is chocolate	True
c:	the cake has walnuts	True
d:	the cake is square	False

Not

not a: the cake does not have pineapple True/False?

And

a and b: the cake has pineapple & the cake is chocolate True/False?

b and c: the cake is chocolate & the cake has walnuts True/False?

Or (slightly different from English…)

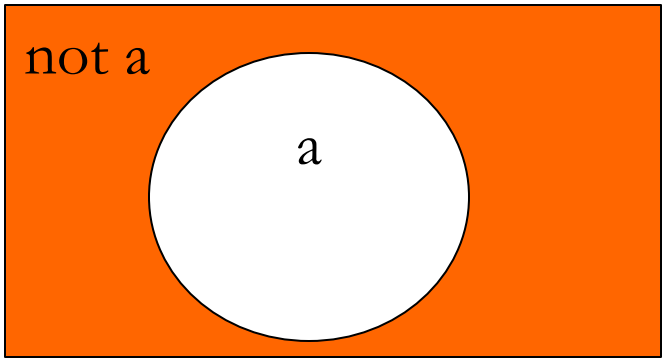
a or b: the cake has pineapple or the cake is chocolate True/False?

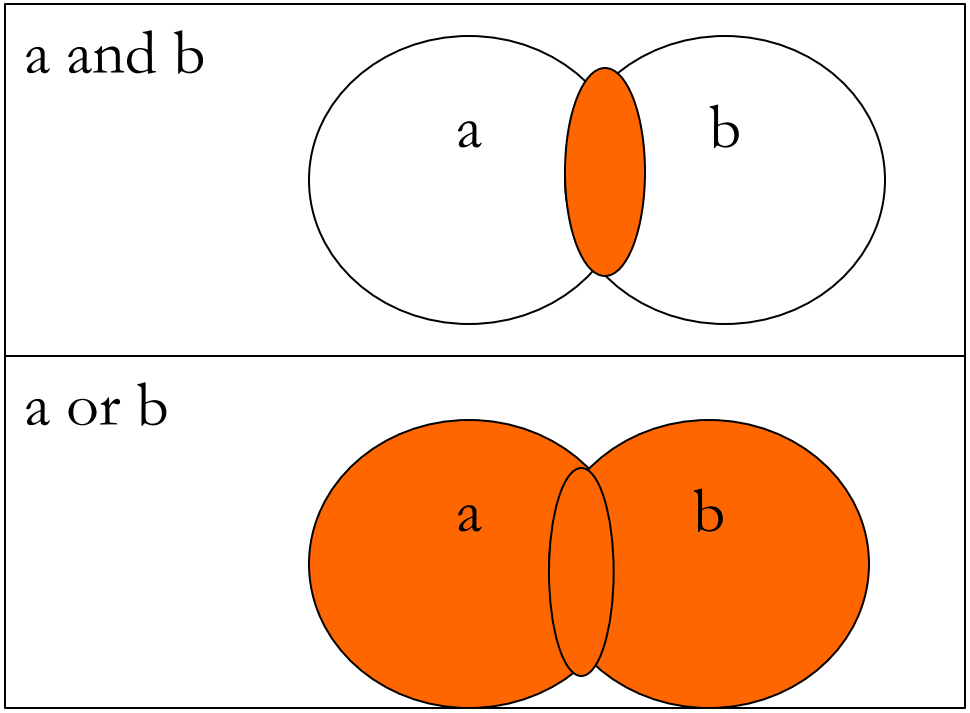
b or c: the cake has chocolate or the cake has walnuts True/False?

a or d: the cake has pineapple or the cake is square True/False?

Logical Operators in Venn Diagrams

Concepts in this slide: Venn diagrams, visual representation of logical operations.





Logical operators are used in everyday speech (see Slide 6), but also consistently in Math and CS.

Logical Operators: not, and, or

Concepts in this slide: Logical operators work with Boolean values or relational expressions.

`not` *exp* evaluates to the opposite of the truth value of *exp*

exp1 `and` *exp2* evaluates to `True` iff both *exp1* and *exp2* evaluate to `True`.

exp1 `or` *exp2* evaluates to `True` iff at least one of *exp1* or *exp2* evaluates to `True`.

```
In [1]: not (3 > 5)
Out[1]: True

In [2]: not (3 == 3)
Out[2]: False

In [3]: (3 < 5) and ('bat' < 'ant')
Out[3]: False
In [4]: (3 < 5) and ('bat' < 'cat')
Out[4]: True

In [5]: (3 > 5) or ('bat' < 'cat')
Out[5]: True
In [6]: (3 > 5) or ('bat' < 'ant')
Out[6]: False
```

Concepts in this slide: and / or expressions produce different Boolean values.

Truth Tables: and

exp1	exp2	exp1 and exp2
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

Truth Tables: or

exp1	exp2	exp1 or exp2
True	True	**True**
True	False	**True**
False	True	**True**
False	False	**False**

Combining logical operators

What cake do I like?

(cake is chocolate) **or** (cake has pineapple) **and** (cake is square)

and takes precedence over **or** (like * over +)

((cake is chocolate) **or** (cake has pineapple)) **and** (cake is square)

Parentheses take precedence

Digging Deeper

Short-circuit evaluation of and and or

In *exp1* **and** *exp2* or *exp1* **or** *exp2*, the expression *exp2* is not evaluated if the answer is determined by *exp1*.

```
In[14]: ((1/0) > 0) and (2 > 3)
---------------------------------------------------------------
ZeroDivisionError                Traceback (most recent call last)
<ipython-input-17-5e0d829f2dca> in <module>()
----> 1 ((1/0) > 0) and (2 > 3)

ZeroDivisionError: integer division or modulo by zero

In[15]: (2 > 3) and ((1/0) > 0)
Out[15]: False

In[16]: (2 < 3) or ((1/0) > 0)
Out[16]: True
```

Predicates

Concepts in this slide: Definition and examples of predicates.

A predicate is a function that returns a Boolean value.

```
def isDarth (name):
    """determines if name is Darth Vader"""
    return name == 'Darth Vader'

def isDivisibleBy(num, factor):
    """determines whether num is divisible by factor"""
    return (num % factor) == 0

def isEven(n):
    """determines whether n is even"""
    return isDivisibleBy(n, 2)

def sameLength(s1, s2):
    """determines whether strings s1 and s2 have the
    same length"""
    return len(s1) == len(s2)
```

Note: The triple-quoted strings are the function docstrings.

More Predicates

Concepts in this slide: Examples of predicates with complex logical expressions.

```
def isBetween(n, lo, hi):
    """determines if n is between lo and hi"""
    return (lo <= n) and (n <= hi)

def isSeason(s):
    """determines if s is one of the four seasons"""
    return (s == 'Winter' or s == 'Spring'
            or s == 'Summer' or s == 'Autumn')

def isSmallPrime(n):
    """determines if n is a prime integer less than 100"""
    return (isinstance(n, int)
            and (n > 1) and (n < 100)
            and (n == 2 or n == 3 or n == 5 or n == 7
                 or not (isDivisibleBy(n,2)
                         or isDivisibleBy(n,3)
                         or isDivisibleBy(n,5)
                         or isDivisibleBy(n,7))))
```

Preview: Some useful string operations

We will cover strings and other "sequence" types like tuples and lists in a few lectures, but here are some useful operations that come handy when writing predicates.

The square bracket **[]** operator can be used to index (access) an element of a string.

```
In [1]: name = 'Esmeralda'
In [2]: name[0]
Out[2]: 'E'

In [3]: name[1]
Out[3]: 's'

In [4]: name.lower()
Out[4]: 'esmeralda'

In [5]: name
Out[5]: 'Esmeralda'
```

To notice:

- The index of the first character is **0** not 1, as you would expect. That is a quirk of many programming languages.
- The method **lower** returns a new string that is the lowercased version of the original one, which doesn't change. This behavior is different from cs1graphics objects.

`in` and `not in` test for substrings

s1 `in` s2 tests if string **s1** is a substring of string **s2**

```
In [1]: 'i' in 'generation'
Out[1]: True

In [2]: 'u' in 'generation'
Out[2]: False

In [3]: 'era' in 'generation'
Out[3]: True
```

```
In [4]: 'get' in 'generation'
Out[4]: False

In [5]: 'nerati' in 'generation'
Out[5]: True
```

What other English words are in the string `'generation'`?

s1 `not in` s2 is the same as **`not` s1 `in` s2**

```
In [6]: 'era' not in 'generation'
Out[6]: False

In [7]: 'get' not in 'generation'
Out[7]: True
```

Your Turn: Write these predicates

Exercise 1: Write the predicate **`isVowel`** that behaves as shown below:

```
In [6]: isVowel('E')
Out[6]: True

In [7]: isVowel('b')
Out[7]: False
```

Exercise 2: Use the predicate **`isVowel`** that you wrote above to write a new predicate **`startsWithVowel`** that behaves like shown:

```
In [8]: startsWithVowel('Esmeralda')
Out[8]: True

In [9]: startsWithVowel('bravery')
Out[9]: False
```

Continuation Characters in Long Expressions

If you want to write expressions without parens that span multiple lines, you **must** use the backslash continuation character to end each line (and this character cannot be followed by any other character except newline). These multiline expressions **cannot** contain embedded comments, such as **`# Is n an integer?`**

```
def isSeason(s):
    """determines if s is one of the four seasons"""
    return s == 'Winter' or s == 'Spring' \
        or s == 'Summer' or s == 'Autumn'


def isSmallPrime(n):
    """determines if n is a prime integer less than 100"""
    return isinstance(n, int) \
           and (n > 1) and (n < 100) \
           and (n == 2 or n == 3 or n == 5 or n == 7
                or not (isDivisibleBy(n,2)
                 or isDivisibleBy(n,3)
                 or isDivisibleBy(n,5)
                 or isDivisibleBy(n,7)))
```

No continuation characters needed in parenthesized expressions

Digging Deeper

Parentheses Instead of Continuation Characters

You can avoid continuation characters by wrapping the expression in explicit parentheses, like the big blue parentheses below:

```
def isSeason(s):
    """determines if s is one of the four seasons"""
    return (s == 'Winter' or s == 'Spring'
            or s == 'Summer' or s == 'Autumn')

def isSmallPrime(n):
    """determines if n is a prime integer less than 100"""
    return (isinstance(n, int)
              and (n > 1) and (n < 100)
              and (n == 2 or n == 3 or n == 5 or n == 7
                  or not (isDivisibleBy(n,2)
                          or isDivisibleBy(n,3)
                          or isDivisibleBy(n,5)
                          or isDivisibleBy(n,7)))
```

ASCII Table: uppercase vs. lowercase

Digging Deeper

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

In computer programs, all data is stored as numbers (binary numbers made of 0 and 1s. Take CS 240 to learn more). ASCII is a standard that specifies the mapping between keyboard characters and numbers. When you compare "A" and "a", you are comparing the underlying numbers 65 and 97.

Test your knowledge

1. What is the result of relational expressions? What is the result of logical expressions? What makes them different?
2. How does the comparison of string values work? Can you provide an example to illustrate?
3. Operators like **>** , **or** are called binary operators, while **not** is called a unary operator. Can you give an educated guess for the why?
4. [MATH] Relational operators are used in Math to describe intervals of numbers. Draw a picture showing the interval 10 to 20 (excluding 20). How would you write this in Python? What about the intervals of numbers less than 5 but greater than 15. Drawing the picture helps visualize relations.
5. Write the Truth Table for the expression **not** (exp1 **and** exp2)
6. Is there any difference between a predicate and a function?
7. What is the result of the expression `'$' > '%'` . How would you explain that to someone?
8. In the expression `3 < 5 and 'bat' < 'cat'` (notice there are no parens), does **and** have priority over **<**? Explain.