Expressions in Python

FEB 8TH 2012

Well-formed expressions

- **Examples:**
  - 1 - 2 * 4 * * 3 – 24
  - len(str(bin(2222/10)))
  - (currentNumber < max) and (currentNumber >= secondMax)
  - not False or True and not True
  - 56 +++++ 32 --- 25
  - 250/0
  - len(str(bin(2222)/10))

- **Examples of "ill-formed" expresions:**
  - (23 + abs(-9)
  - "33 + "25"
  - 3(12 + 4)

Well-formed expressions

- Python has a bunch of rules for determining whether an expression has correct structure (similar to grammar rules in a language that determine whether a sentence has correct structure).

- These rules, by themselves, do not guarantee that the expression is meaningful (see the last two well-formed expression examples from the previous slide).

- These rules are what you would expect:
  - A constant or variable by itself is a well-formed expression.
  - A unary operator (e.g., -, not) should be followed by a well-formed expression.
  - A binary operator should be preceded by and followed by well-formed expressions.
  - If you put parentheses around a well-formed expression, it will be well-formed.
  - If f is a function name and X, Y, Z, etc. are well-formed expressions, then f(), f(X), f(X, Y), f(X, Y, Z), etc. are all well-formed expressions.

Evaluating expressions

- Syntax rules defining well-formed expressions tell us which expressions are structurally correct, but do not tell us how to evaluate expressions.

- Here are examples of expressions in which there is some ambiguity.

- **Examples:**

```
1 - 2 * 4 * * 3 – 24
not False or True and not True
```

- Python has rules on *order of evaluation* and *operator precedence* to help resolve such ambiguities.

Python’s algorithm for evaluating expressions

1. Evaluate expressions inside inner-most parentheses first.
2. Evaluate sub-expressions involving operators with higher precedence first.
3. Sub-expressions involving operators of the same precedence are evaluated left to right.

- Rule (1) implies that parentheses can be used to override the other rules.

Operator precedence

Operator	Meaning
f (...)	function application
**	exponentiation
-E	change sign
*, /, //, %	multiplication, division, remainder
+, -	addition, subtraction
<, >, <=, >=, ==, !=	comparison
not	logical negation
and	logical conjunction
or	logical disjunction

Examples

1. not False or True and not True
   1. not False is evaluated first: True or True and not True
   2. Not True is evaluated next: True or True and False
   3. True and False is evaluated next: True or False
   4. True or False is evaluated next: True
2. 1 - 2 * 4 * * 3 – 24
   1. 4 * * 3 is evaluated first: 1 – 2 * 64 – 24
   2. 2 * 64 is evaluated next: 1 – 128 – 24
   3. 1 – 128 is evaluated next: -127 – 24
   4. -127 – 24 is now evaluated: -151

and and or are “short-circuit” operators

- In evaluating boolean operators and and or Python tries to get away with the minimum evaluation needed to figure out the value of the expression.

- A and B:
  - A is evaluated first.
  - If A is False then the expression evaluates to False, *without B being evaluated.*
  - If A is True then B is evaluated and the expression evaluates to the value of B.

Try evaluating these example expressions

- 100/0
- False and (100/0)
- (100/0) and False
- True and (100/0)
- (100/0) and True

`and` and `or` are "short-circuit" operators

- A `or` B:
  - A is evaluated first.
  - If A is `True` then the expression evaluates to `True`, *without B being evaluated.*
  - If A is `False` then B is evaluated and the expression evaluates to the value of B.

Python associates boolean values to everything

- Every object (e.g., "6", 9.98, "") has an associated boolean value.
- Use the **bool** function to find out the boolean value of an object.
- **Examples:** Try evaluating

bool("a")	bool(0)	x = 6
bool("")	bool(1)	bool(x)

What is True? And what is False?

True	False
The constant `True`	`The constant False`
1, numbers other than 0	0
Non-empty strings	Empty strings

Later when we study *Lists*, *Dictionaries*, etc., we will see that empty instances of these types of objects are also considered False.

A new version of the intToBinary program

```
while n:
    suffix = str(n%2) + suffix
    n = n/2
```

The boolean expression after the while can just be n instead of n > 0.

Some silly examples

- 10 < 20 and 50
- “hello” and “” or 70 < 20
- not not not 20