Variables and Expressions in Python

FEB 11TH 2013

Variables in Python: The "sticky note" model

- Variables are "sticky notes" attached to objects.
- What happens during an assignment statement?

```
x = 10
```

  - A memory cell (4 or 8 bytes large) is created and the value 10 is placed in it.
  - The label "x" is attached ("stuck") to this memory cell.

Multiple "sticky notes" at the same location

What happens when we execute the following code?

```
x = 10
y = x
x = x + 1
```

1. x is a "sticky note" attached to a memory cell containing 10.
2. Then the label y is also stuck to this very location.
3. When x = x + 1 is executed, remember the memory cell containing 10 remains unchanged and the "sticky note" x is moved to the cell with 11.
4. Therefore y continues to have value 10.

Naming Variables

- Variable names need to start with a letter (upper or lower case) or an underscore (i.e., _).
- Following the first character, any sequence of letters, digits, and underscores is allowed.
- Python has a small number of *keywords,* that cannot be used as variable names:

and	del	from	not	while	as	elif	global
or	with	assert	else	if	pass	yield	break
except	import	print	class	exec	in	raise	continue
finally	is	return	def	for	lambda	try	

Naming Variables

- Case matters. The variables count and Count are different.
- Do not use lower case el (“l”), upper case oh (“O”), or upper case eye (“I”) as single letter variable names. These are hard to distinguish from numerals 0 and 1 in some fonts.
- Use meaningful names: e.g., factorBound, myUpperLimit, sequenceLength, etc.
- Watch out for spelling errors in variable names.

Scope of a Variable

- In Python there is no explicit variable declaration.
- In many languages (C, Java, etc.) variables have to be declared before they can be used.
- In programs in these languages, a variable comes into existence when it gets declared.
- In Python, a variable comes into existence when it is first assigned a value.
- The variable lives until the end of the program or until it is explicitly deleted using the del operator (this operator will become useful later).
- The scope of a variable is the portion of the program that the variable is in existence for.

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