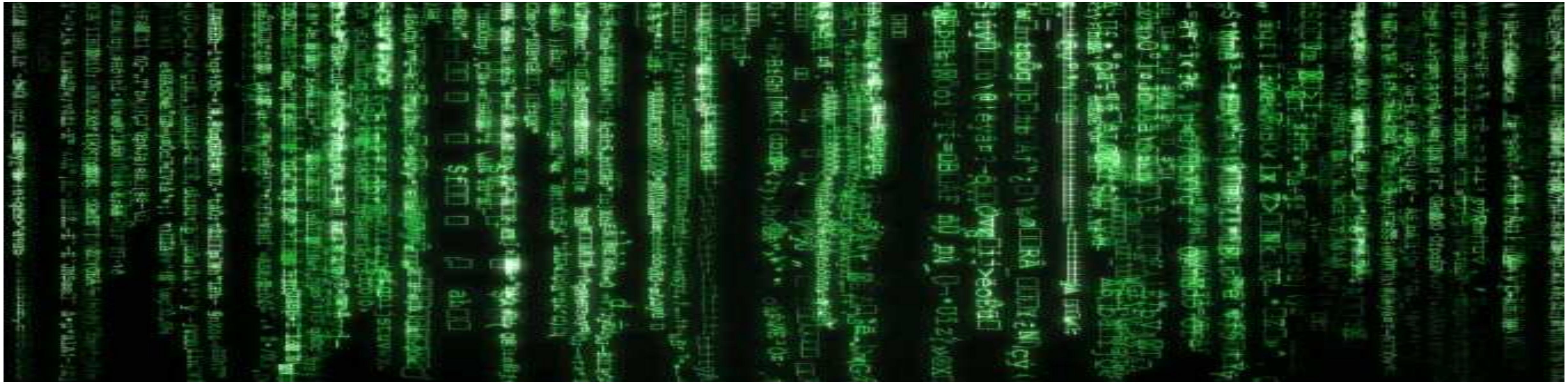


INF 110 Discovering Informatics

More Expressions

NORTHERN ARIZONA UNIVERSITY

Logical Operations

Just like comparison operations, logical operations evaluate to Boolean values.

The most basic examples of logical operations are “and”, “or” and “not”.

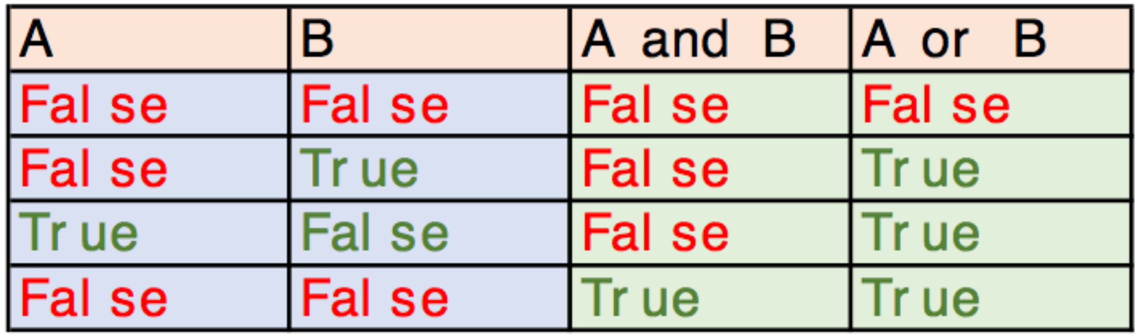
In “and”, if x and y are **True**, then if one is **False**, then the statement is **False**.

“or” means one or both values are True.

- “True or True” evaluates to **True**, and so does “True or False”

- `x and y`
- `x or y`
- `not x`

- `True and True -> True`
- `True or True -> True`



A	B	A and B	A or B
Fal se	Fal se	Fal se	Fal se
Fal se	Tr ue	Fal se	Tr ue
Tr ue	Fal se	Fal se	Tr ue
Fal se	Fal se	Tr ue	Tr ue

Logical Operations

“not” is pretty simple. It’s the opposite of whatever is given.

- Not False is **True**, and Not True is **False.**

We can also combine logical operations into chains.

- i.e., “A and not (B or C)”
- This means A has to be **True**, but neither B nor C can be **True**.

- `x and y`
- `x or y`
- `not x`

- `True and True -> True`
- `True or True -> True`

A	B	A and B	A or B
False	False	False	False
False	True	False	True
True	False	False	True
False	False	True	True

If statements

These are used to conditionally execute blocks of code.
If the condition is satisfied, then we execute the code (otherwise, we just skip it)

```
if <expr>:
   statement1
   statement2
   …
elif <expr>:
   statement1
   statement2
   …
else:
   statement1
   statement2
   …
```

Here is an “if”, “elif”, “else” block.
This is saying:

- “if the first expression is true, run the first block of code.”
- Otherwise, if the first condition is false and the second is true, run the second block of code.
- Or if no other condition is true, just run the last block of code.

If statements

These are used to conditionally execute blocks of code.
If the condition is satisfied, then we execute the code (otherwise, we just skip it)

```
if <expr>:
   statement1
   statement2
   …
elif <expr>:
   statement1
   statement2
   …
else:
   statement1
   statement2
   …
```

We can have any combination of “ifs” and “elifs”.

But, we can only have one “else”, because this is kind of a “catch-all” block.

Therefore, a conditional can have:

- 1 or more “if” blocks
- 0 or more “elif” blocks
- 0 or 1 “else” block.

Live Code How's the Weather?

Task: Write an if statement that provides a statement about the weather based on the temperature

Learning Outcomes

- Designing if statements
- Solving problems with comparison operators
- Solving problems with logical operators

You can practice writing the provided code out, or try to write your own (not an assignment)

Function Composition

Here we're going to organize our code so that the output of one function serves as the input to another function.

The comments here give a good idea of what's going on.

```python
# First we associate the string with a variable
x = "Plum"

# Second we find the length of x
x_length = len(x)

# Third we convert it to binary
bin_string = bin(x_length)

# Fourth, we print the result
print(bin_string)
```

Copy and paste this code into a Jupyter cell to check it out.

Function Composition

By using function composition, we can combine all of those steps into a single line:

```python
print(bin(len("Hello")))
```

This code is functionally identical to the previous code.
But, there are no intermediate values -> we’re just chaining the functions together

Python evaluates these statements “inside out” -> from the innermost parentheses first

Function Composition

```
number = input("Enter␣a␣number:␣")
base = input("Enter␣its␣base:␣")

print(int(number, int(base)))
```

Let’s assign two variables based on ***user input***.
The input command will print a prompt to the screen and wait for the user to provide their input.

So when the first line is run, it will display “Enter a number: “ and wait for you to say something.

When you respond with 11, it will then assign 11 to the variable “number”.

The next line converts the value for “number” into a representation that is in the specified base.
- If you enter “2”, it will convert the number to base 2.
- So for 11, it will print 3 because that is 11 in binary representation.

Subexpressions

A logically complete expression that is part of a compound expression.

```
print(bin(len(x) + 3))
```

- What are the subexpressions?
- What wouldn't be a subexpression?

Subexpressions

A logically complete expression that is part of a compound expression.

```
print(bin(len(x) + 3))
```

- What are the subexpressions? **len(x)**; **len(x) + 3**
- What wouldn't be a subexpression? **+ 3**

Because by itself, Python would not be able to evaluate it!

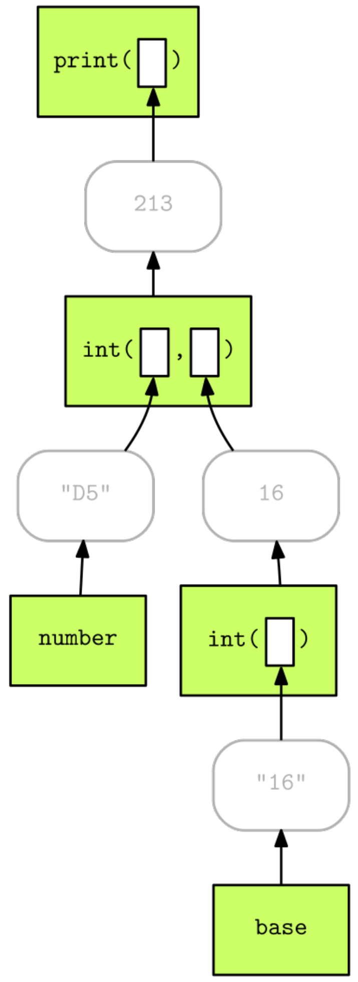
Expression Tree Evaluation

Before the expression can be evaluated, Python first needs to set up an ordering of which expressions to evaluate first.

1. Calculate the length of x
2. Add 3
3. Convert to binary
4. Print.

To plan this ordering, Python uses an expression tree.

From the bottom, it iteratively evaluates and combines the expressions into the final result.



Method Chaining

Let’s revisit the difference between functions and methods.

This is a very fine distinction, and people often use the terms interchangeably.

Methods are functions that operate on an object.
Methods require the dot operator.
Functions can be called independently, so you don’t need the dot operator.

Method chaining is like function chaining: they are evaluated left to right.

Method Chaining

This is the more naïve approach to method chaining – they’re all on separate lines.

Here, we create a variable x, and set it to equal a string “flagstaff” with spaces surrounding it.

Then, we use the strip method to cut off the whitespace, then the capitalize method, then print.

```python
x = "␣␣␣flagstaff␣␣␣␣"

# Remove the whitespace
x = x.strip()

# Capitalize
x = x.capitalize()

print(x)
```

Method Chaining

The same thing can be accomplished using method chaining:

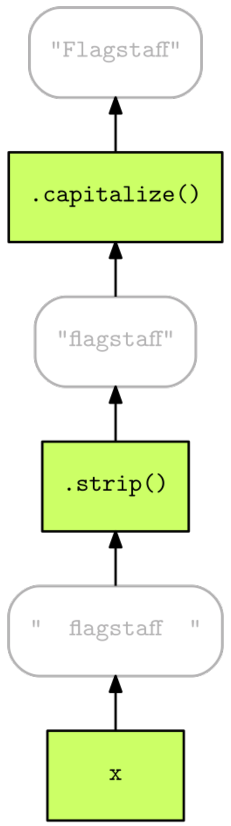
```
x = "␣␣␣flagstaff␣␣␣␣"

print(x.strip().capitalize())
```

Note the dot operator.

No intermediate values!

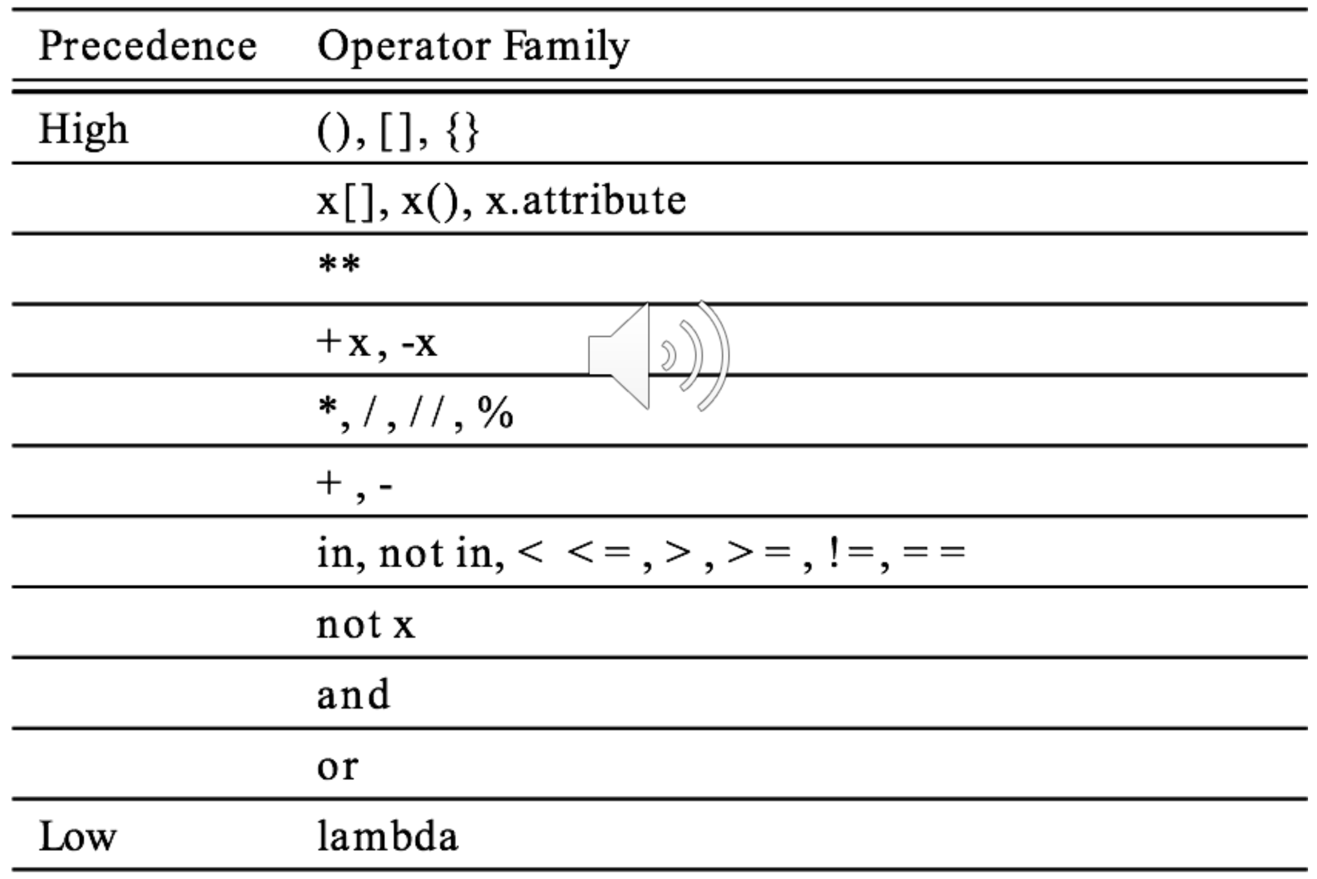
Keep in mind- it’s evaluated left to right!!!!

Method Chaining Expression Tree



Operator Precedence

Table 0.0.1: *Operator Precedence*



Precedence	Operator Family
High	(), [], {}
	x[], x(), x.attribute
	**
	+x, -x
	*, /, //, %
	+, -
	in, not in, < <=, >, >=, !=, ==
	not x
	and
	or
Low	lambda

How Python builds it’s expression trees.

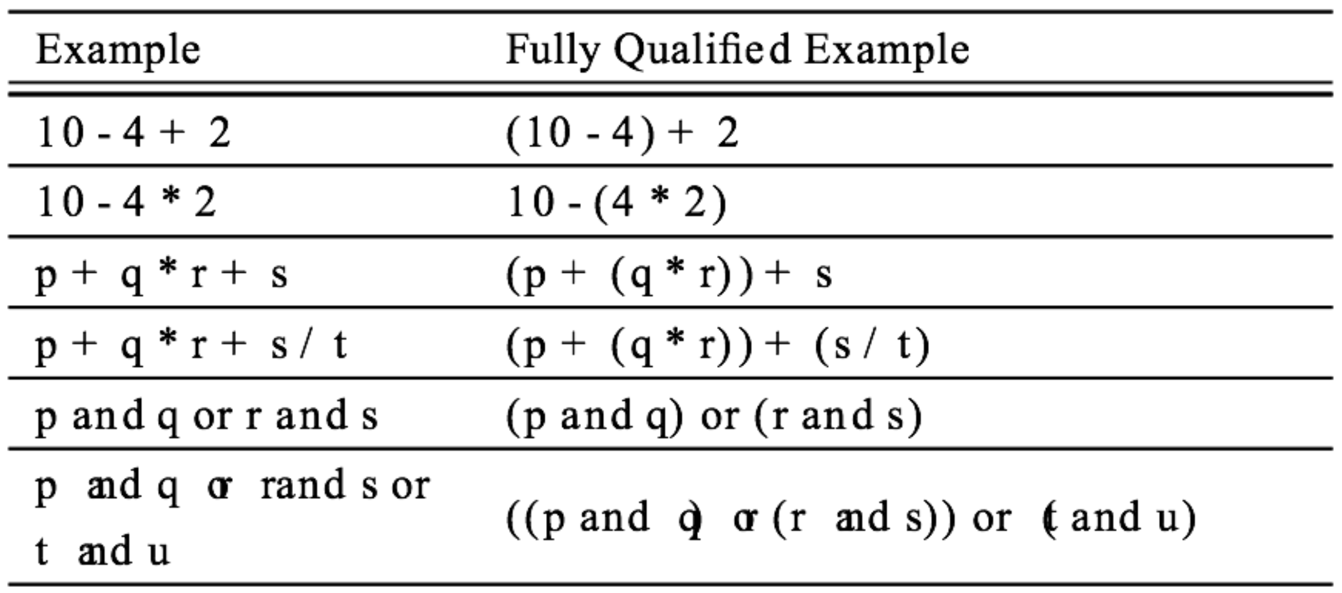
Operator Precedence

You could force your own orders of operations by using parentheses.

For instance: “A+B*C/D” can be “(A+B)*(C/D)”

Operator Precedence

Table 0.0.1: *Operator Precedence Examples*

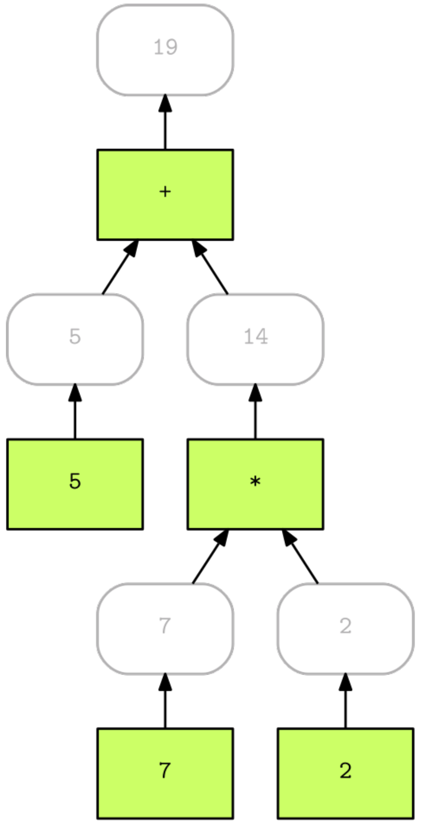


Example	Fully Qualified Example
10 - 4 + 2	(10 - 4) + 2
10 - 4 * 2	10 - (4 * 2)
p + q * r + s	(p + (q * r)) + s
p + q * r + s / t	(p + (q * r)) + (s / t)
p and q or r and s	(p and q) or (r and s)
p and q or r and s or t and u	((p and q) or (r and s)) or (t and u)

If you write what's on the left, Python will evaluate it according to what's on the right.

The best way (IMO) is to use your own parentheses to make sure Python is doing what you want it to do!

Operator Precedence



Augmented Assignment Operators

- `x += y` means `x = x + y`
- `x -= y` means `x = x - y`
- `x *= y` means `x = x * y`
- `x /= y` means `x = x / y`

These are used to shorten your code into something more readable and writeable.

They are “modify and reassign” operations.

You don’t have to use them, though.

Augmented Assignment Operators

```
number_of_widgets = 7

# We can add one the long way..
number_of_widgets = number_widgets + 1

# Or the short way..
number_of_widgets += 1
```

You can see here how augmented assignment operators make code a little simpler.

Live Code Vinyl Record Sales?

Task: What is the **percent difference** in record sales for these two years?

14.32 million in 2017

13.1 million in 2016

Learning Outcomes

- Solving problems with math operators
- Using data to make inferences

end