COMP1021
Introduction to Computer Science

More on Operators

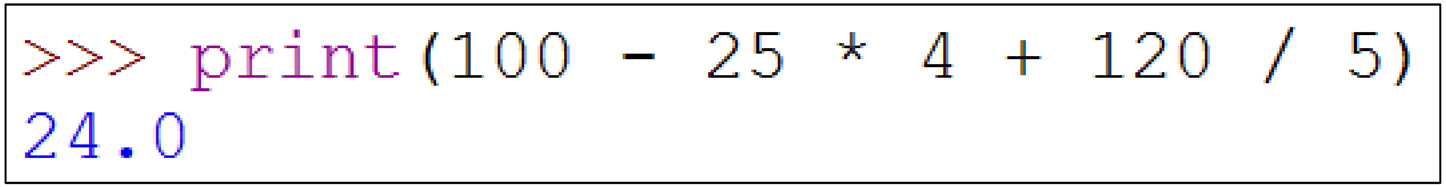
David Rossiter and Gibson Lam

Outcomes

- After completing this presentation, you are expected to be able to:
    1. Explain the use of the various kinds of Python operators
    2. Write code to represent `True` or `False` using numbers, lists, tuples or strings
    3. Apply operator precedence in expressions

Python Operators

- We already know we can do common maths things in Python, i.e. `+ - / *`



```
>>> print(100 - 25 * 4 + 120 / 5)
24.0
```

- These things are called *operators*
- This presentation gives you summaries of different types of operators (you have already used most of them in the course)
- We will also look at some other things that you need to know about operators

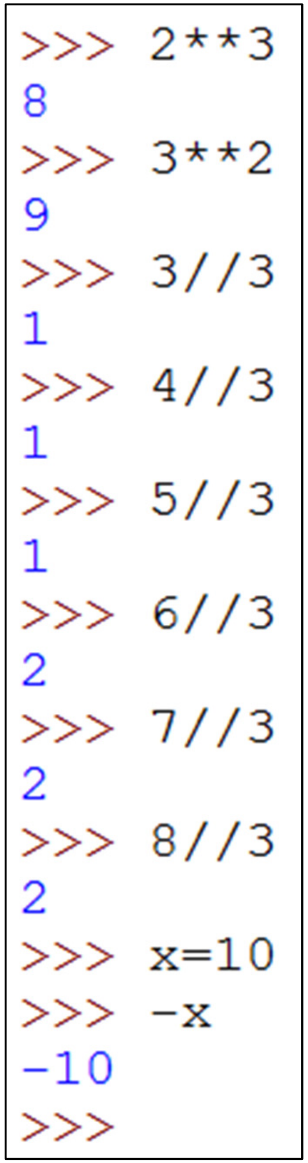
Arithmetic Operators

- Basic operators: `+  -  /  *  %`
- ‘Advanced’ operators:

  `**` means ‘to the power of’

  `//` means ‘do division, return the integer result’

  `-x` means the same as ‘`-1 * x`’



```
>>> 2**3
8
>>> 3**2
9
>>> 3//3
1
>>> 4//3
1
>>> 5//3
1
>>> 6//3
2
>>> 7//3
2
>>> 8//3
2
>>> x=10
>>> -x
-10
>>>
```

Comparison Operators

- For comparing between two values:

  `a <  b` returns `True` if `a` is less than `b`

  `a <= b` returns `True` if `a` is less than or equal to `b`

  `a >  b` returns `True` if `a` is greater than `b`

  `a >= b` returns `True` if `a` is greater than or equal to `b`

  `a == b` returns `True` if `a` is equal to `b`

  `a != b` returns `True` if `a` is not equal to `b`

- All of them return `False` otherwise

Logical Operators

- Logical operators work with boolean values, i.e. `True` or `False`

`a and b` returns `True` if both `a` **and** `b` are `True`, and `False` otherwise

`a or b` returns `True` if either `a` **or** `b` is `True`, and `False` otherwise

`not a` returns `True` if `a` is `False`, and `False` otherwise

- It is easier to understand what they do by looking at the table on the next slide

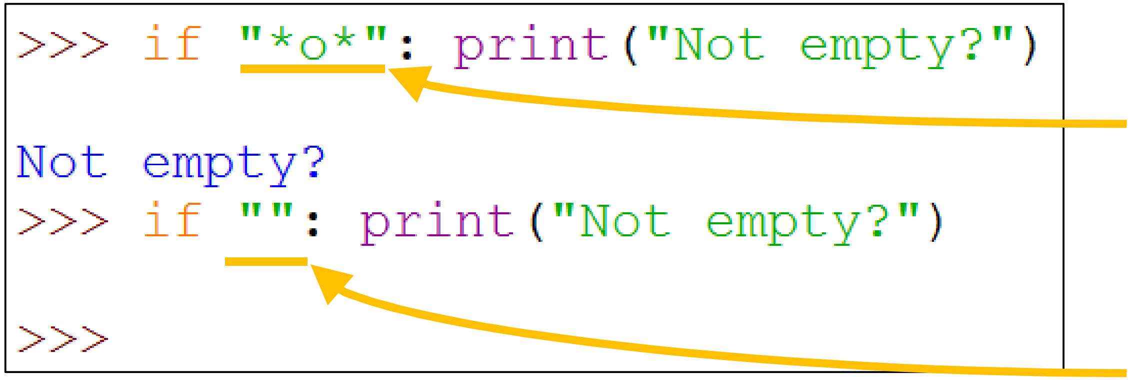
Summary Table of Logical Operators

- Here is a summary of the input and output of logical operators:

`a`	**`b`**	**`a and b`**	**`a or b`**	**`not a`**
`True`	`True`	`True`	`True`	`False`
`True`	`False`	`False`	`True`	`False`
`False`	`True`	`False`	`True`	`True`
`False`	`False`	`False`	`False`	`True`

Using Other Things as True/False

- In most programming languages including Python you can:
  - Use any number other than 0 to represent `True`
  - Use 0 to represent `False`
- You can also use an empty list, tuple or string to represent `False` and any non-empty one to `True`



```
>>> if "*o*": print("Not empty?")

Not empty?
>>> if "": print("Not empty?")

>>>
```

Python sees this as `True`

Nothing is printed because Python sees this as `False`

Using the Equal Sign

- You use the equal sign to put things into a variable, i.e. `age = 25`
- Sometimes you may want to do something like this (adding one to the variable `count`):

```
count = count + 1
```

- When you are doing something to the **same** variable Python allows you to use a shortcut, like this:

```
count += 1
```

Using Shortcuts with the Equal Sign

- You can use the equal sign together with almost all arithmetic operators, for example:

```
calories = calories + 800   ⇨   calories += 800

pigs = pigs * 5             ⇨   pigs *= 5

cakes = cakes / students    ⇨   cakes /= students

marks = marks - 20          ⇨   marks -= 20

hello = hello + "!"         ⇨   hello += "!"
```

This works for strings too, not just numerical values

Operators for Lists, Tuples and Strings

- These operators are used by lists, tuples and strings:

  `x + y` concatenates (put together) two lists, tuples or strings

  `x * n` concatenates `n` copies of `x`

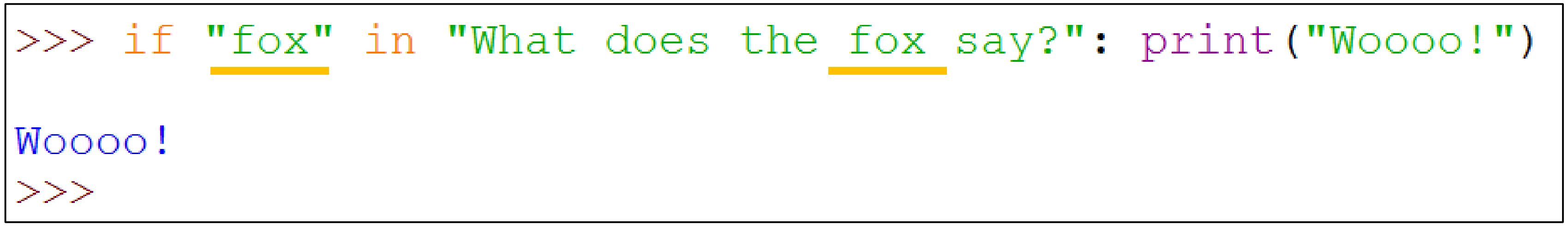
  `a in x` returns `True` if `a` is in collection `x` and `False` otherwise

  `a not in x` returns `False` if `a` is in collection `x` and `True` otherwise

- The `in` and `not in` operators also work with checking the existence of keys in dictionaries

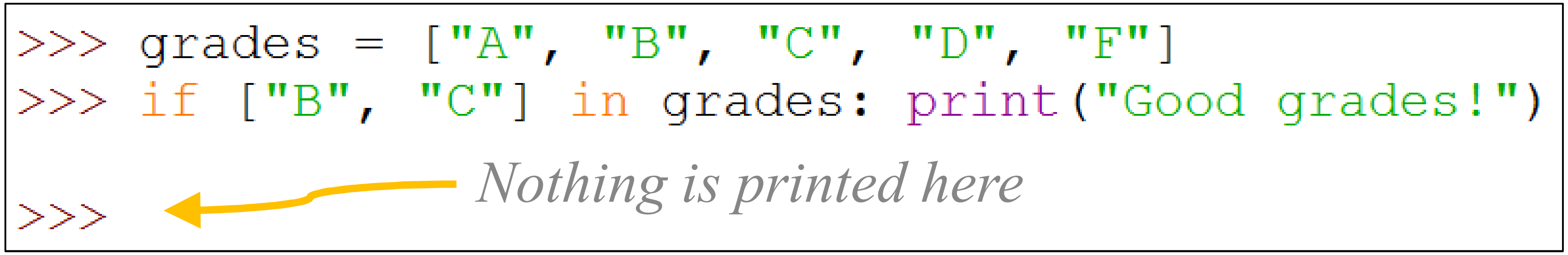
Using ‘in’ for Substrings

- Using the `in` operator you can test for substrings inside any string, like this:



```
>>> if "fox" in "What does the fox say?": print("Woooo!")

Woooo!
>>>
```

- However, you cannot do the same thing for ‘sub-list’ or ‘sub-tuple’, as shown below:



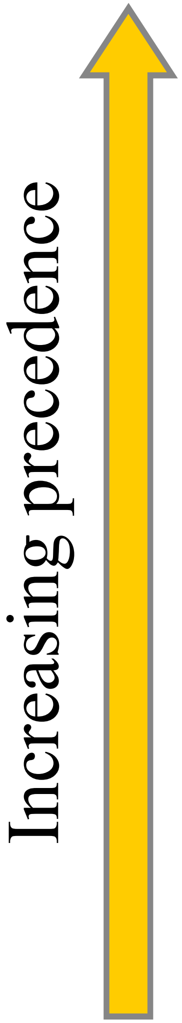
```
>>> grades = ["A", "B", "C", "D", "F"]
>>> if ["B", "C"] in grades: print("Good grades!")

>>>
```

Nothing is printed here

Operator Precedence

- If we ask Python to calculate `2 + 3 * 4` what will the result be?
  - You might think the answer is `5 * 4` is 20
  - You are wrong!
  - This is because `*` has *precedence* over +
  - So `3 * 4` will be calculated first, then the result (12) will be added to 2, so the answer is 14
- If you always use brackets, e.g. `2 + (3 * 4)`, then you don’t need to worry about precedence, but you need to understand what happens when there aren’t any brackets

The Precedence Table



Increasing precedence

- Highest precedence -

`( )`

`**`

`-x, +x`

`*, /, %, //`

`+, -`

`<, >, <=, >=, !=, ==`

`in, not in`

logical `not`

logical `and`

logical `or`

- Lowest precedence -

So if you use parentheses, it overrides everything else

Precedence Example 1

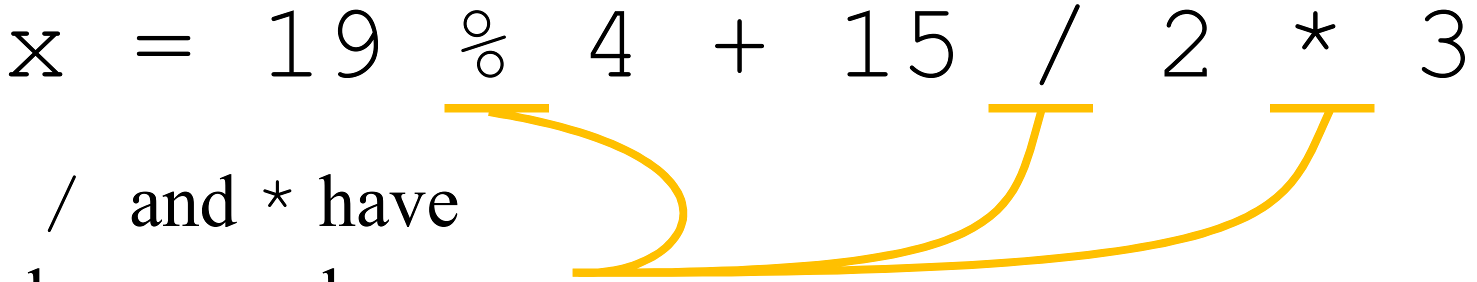
```
x = 17 / 2 * 3 + 2
```

- / and * have higher precedence than +, so they are handled first
- / and * have equal precedence, so the one on the left (/) is evaluated first

- So the answer is:

```
=((17/2) * 3) + 2
= 27.5
```

Precedence Example 2



```
x = 19 % 4 + 15 / 2 * 3
```

- %, / and * have higher precedence than +, so they are handled first
- %, / and * have equal precedence, so the one on the left is evaluated first, which is %, then /, then *

- So the answer is:

```
=(19%4) + ((15/2)*3)
= 25.5
```

Precedence Example 3

```
x = 17 / 2 % 2 * 3**3
```

- `**` has a higher precedence than the others, so it is handled first
- `/`, `%`, and `*` have equal precedence, so the one on the left (`/`) is evaluated first, then `%`, then `*`

- So the answer is:

```
=((17/2)%2)*(3**3)

=((17/2)%2)*   27

= 13.5
```