COM4013 Introduction to Software Development

- Week 3
- More on Operators and Conditions
- 'while' Loops

- *Umar Arif – u.arif@leedstrinity.ac.uk*

Expressions

- An expression is a calculation in Python which is essentially a combination of variable, values, operators that results in a value.

```
length * width    # area of a square
celsius * 9.0 / 5.0 + 32
```

- Expressions on their own like this rarely make useful statements, we normally use expressions as part of a more complex statement:

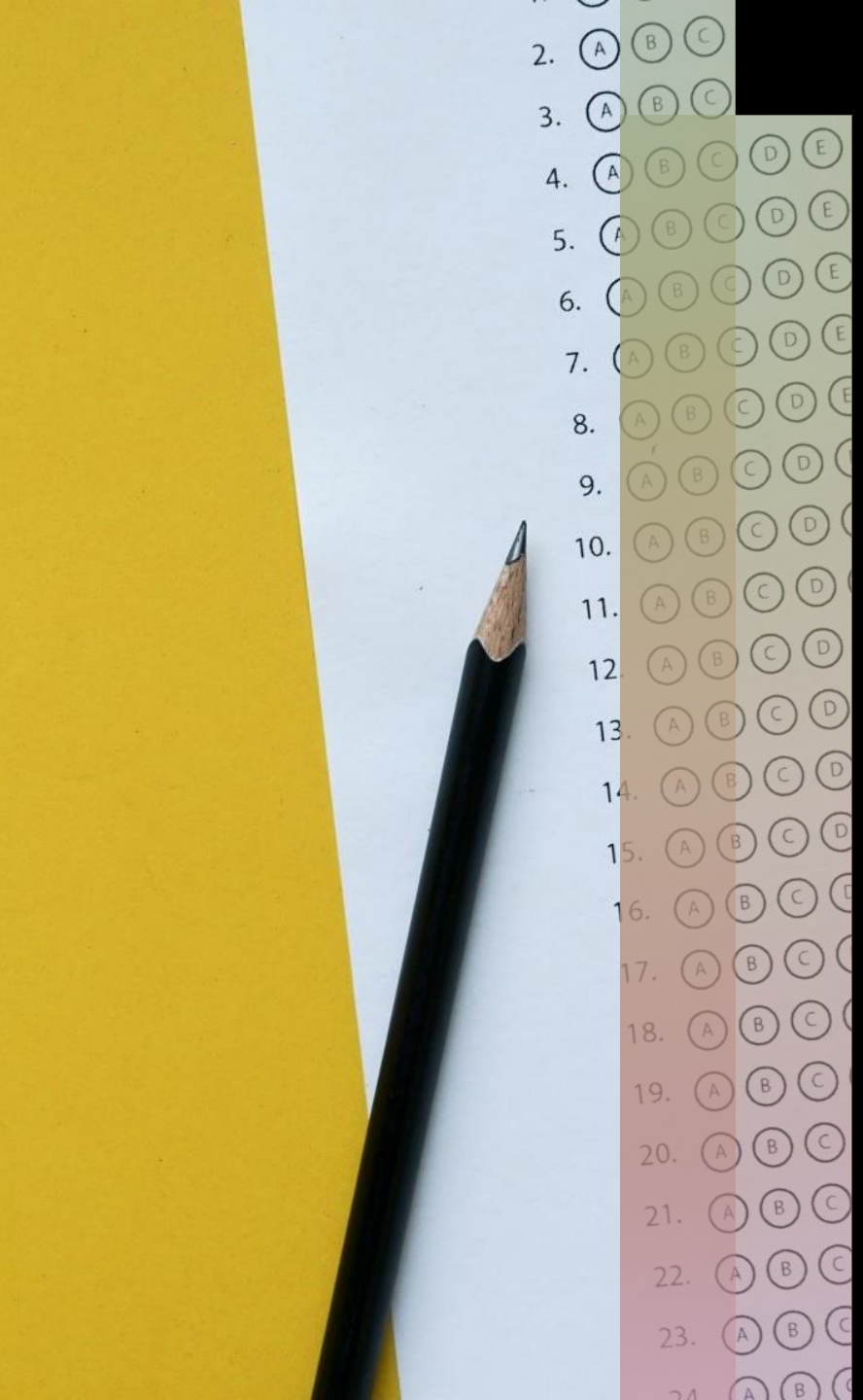
```
areaOfWall = wallLength * wallHeight
print(f"Area of the wall is {areaOfWall}")
```

- Python will evaluate expressions that you write (work out their value), but sometime the result depends on the order of calculations, for example:

  What is the value of: `8 / 2 + 2`

Operator Precedence

- The expression 8 / 2 + 2 could mean:
  - Do the division first: (8 / 2) + 2 = 4 + 2 = 6
  - Or do the addition first: 8 / (2 + 2) = 8 / 4 = 2
- The order in which the operators apply changes the result
- Most programming languages have an operator precedence, a fixed ordering for operators. The order for arithmetic is:
  - 1st: Brackets (i.e., do any bracketed part first)
  - 2nd: Division, multiplication, & modulus (remainder)
  - 3rd: Addition, subtraction
- So, the example above should result in 6, because division is before addition
- You may have been taught this using a term like 'BODMAS'
  - However, such terms can be confusing – see the next slide

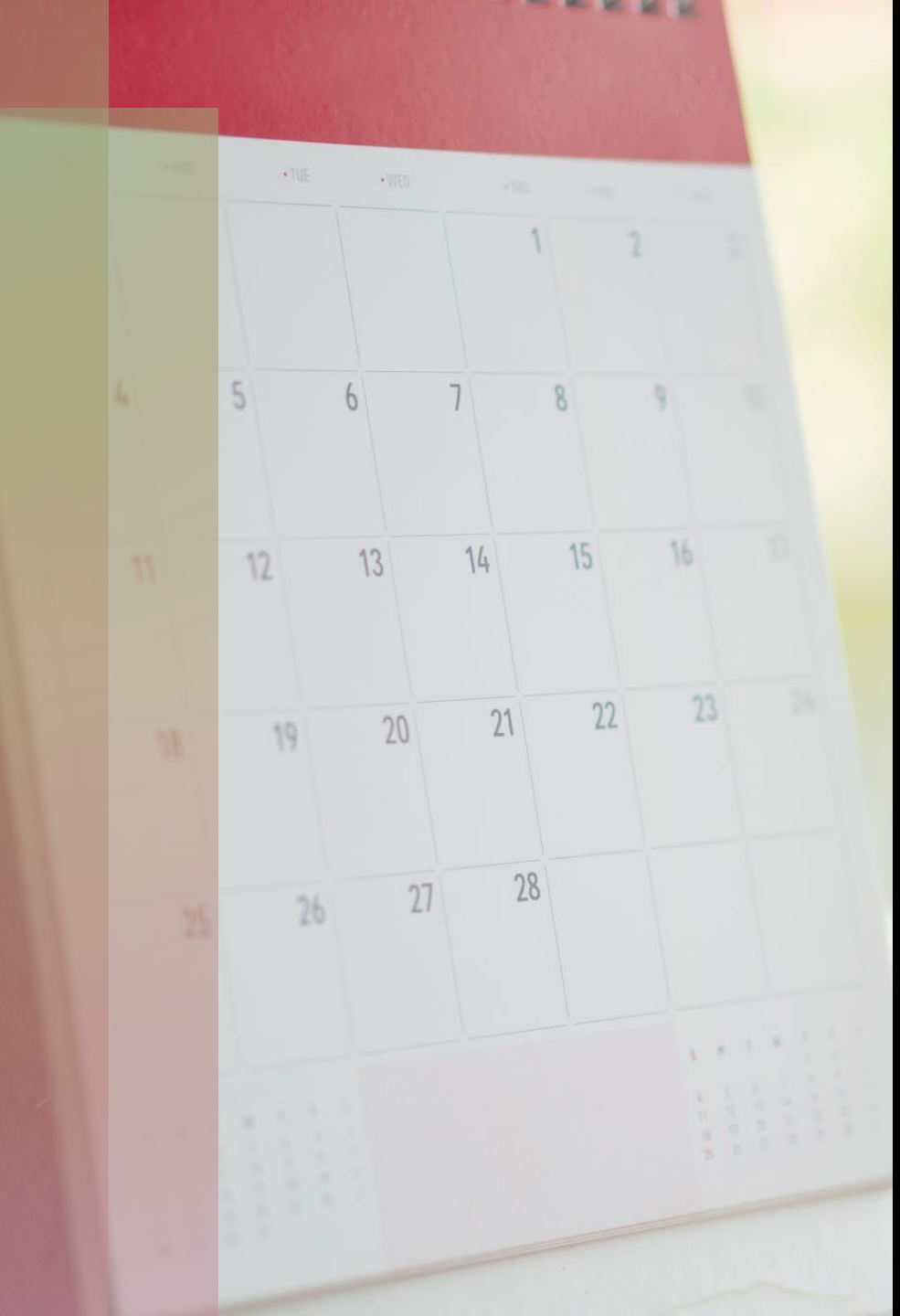


Equal Precedence

- Notice that multiplication and division have equal precedence.
  - So, what is the value of: 8 / 2 * 2
    - It could be: (8 / 2) * 2 = 4 * 2 = 8
    - Or: 8 / (2 * 2) = 8 / 4 = 2
- In this case, most programming languages will start with the left operator (i.e., the division in the example above)
  - However, there are some exceptions. So, you should use brackets when mixing operators of equal precedence to be clear:

    result1 = (8 / 2) * 2

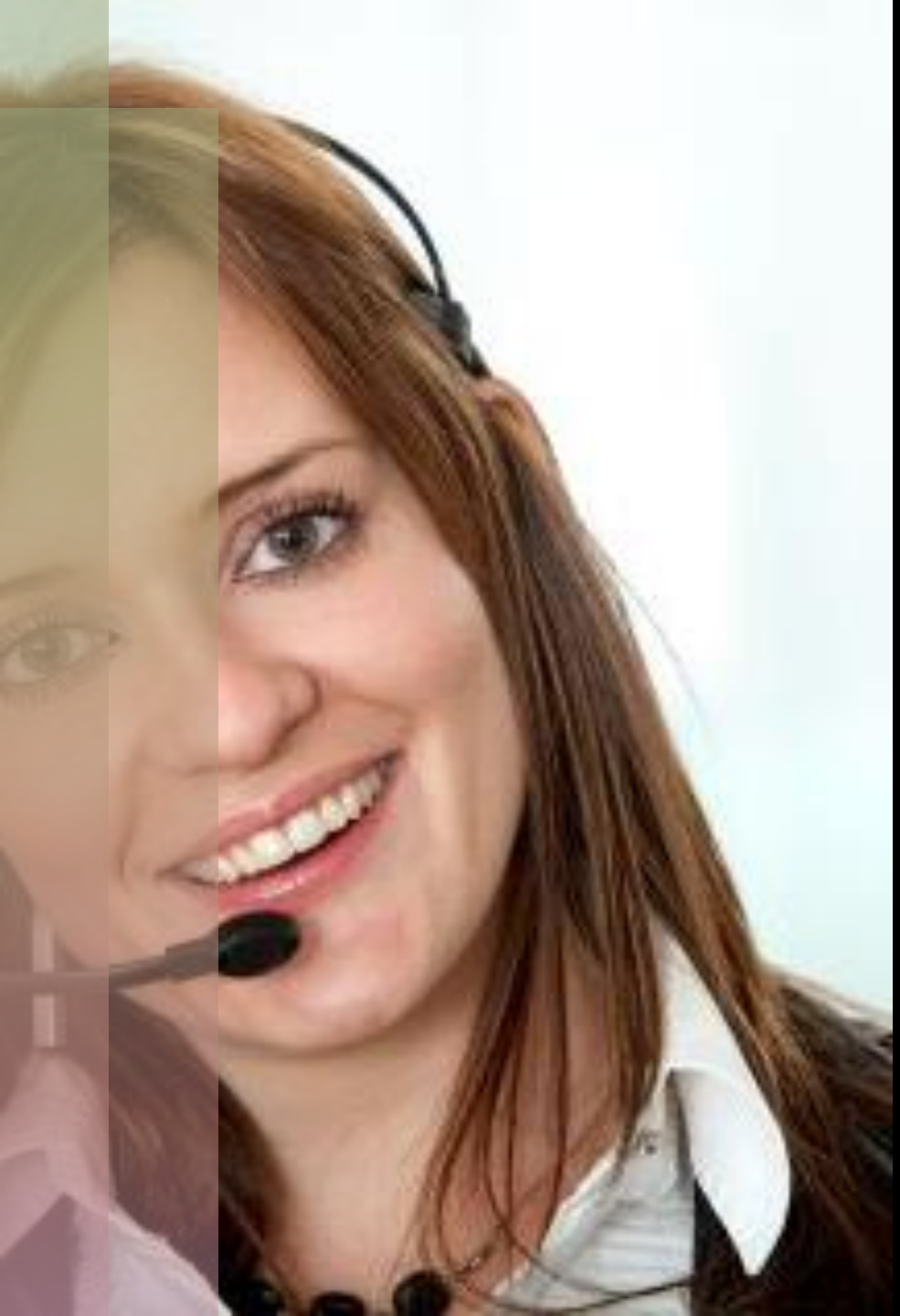
    result2 = (9 – 3) + 4
- In fact, it's a good idea to use brackets whenever you are not sure



Operator Precedence and Brackets

Here's a full precedence list for all the operators you will meet in this part of the module (we haven't covered some of these yet!)

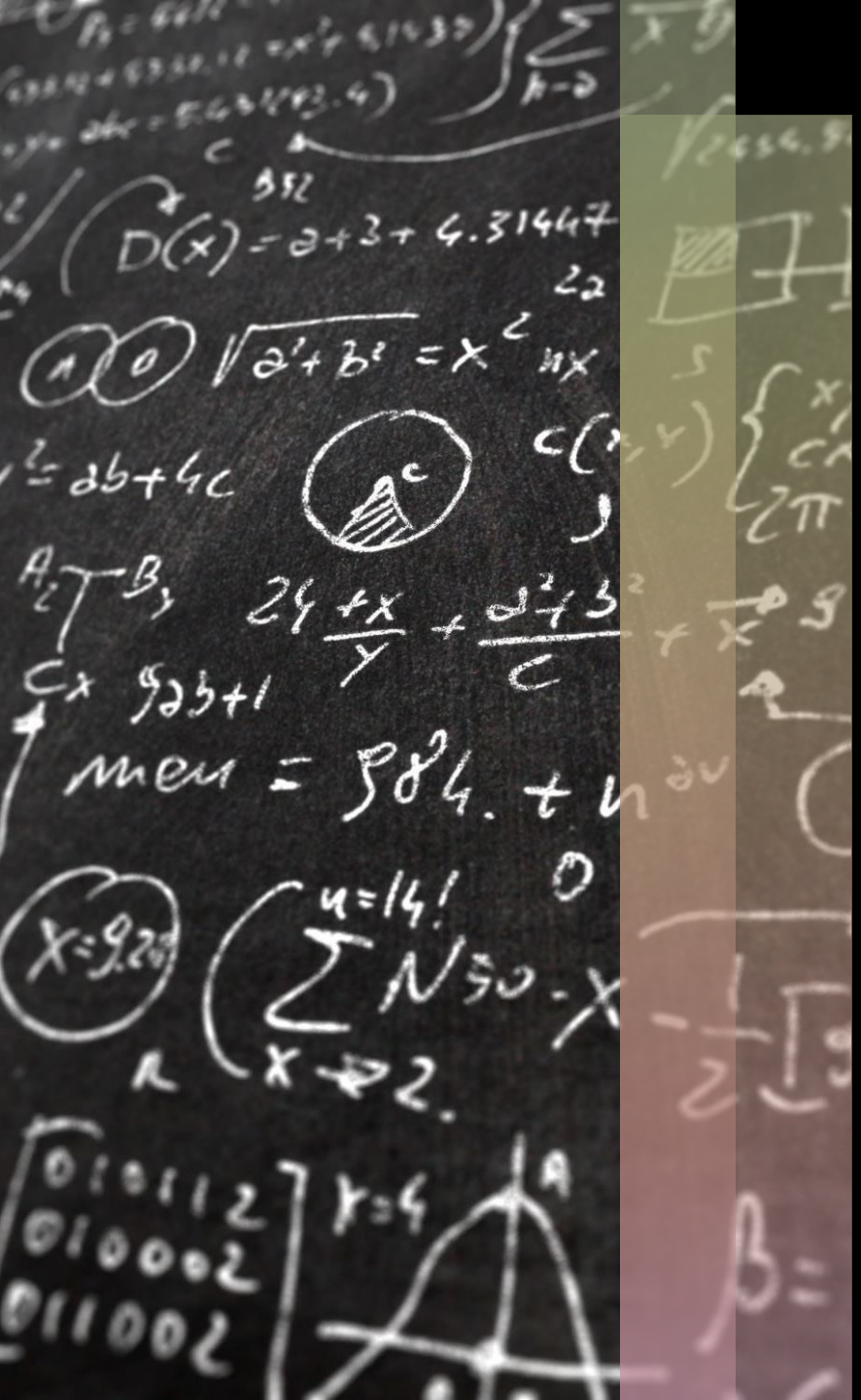
1	()	Brackets
2	not, +, -	Logical Not, making a value positive or negative
3	* / %	Multiply, divide, modulus (remainder)
4	+ -	Addition, subtraction
5	< > <= >=	Greater & less-than comparisons
6	== !=	Equal & not-equal comparisons
7	and	Logical And
8	or	Logical Or
9	= += -= *= /= etc.	Assignment and assignment operators



Operator Precedence and Brackets

You may be used to syntax like this:

- `!isRunning` # using a `!` to suggest the value of a Boolean being False
- In Python we have a `not` operator, for logical negation.
- But we can use both `not` or `!=` in our conditions
  - `x = True`
  - `x = not x  # x has been assigned to False`
  - `y = not x  # y has been initialised to True`
- Also `++` (increment) and `--` (decrement). To increase the value of an integer by 1 does not exist in Python...
  - If we want to increment the value of an integer by 1 (for counting purposes) we can do the following:
    - `x += 1  # This means x = x + 1`
    - `x -= 1  # This is an example of decrementing by 1`



Combined Conditions: And, Or

Imagine we had the user's age and wanted to check if they were in their 20's. Both conditions must be true:

```
age >= 20       and       age <= 29
```

Logical operators in Python allow us to combine conditions like this:

```
and                 or
```

You may be used to `&&` and `||` syntax. Forget these when using python, they do not exist (`and` + `or` are more readable).

So, the condition to check if the user is in their 20's is:

```
age >= 20 and age <= 29
```

Which we might use in an if statement like this:

```
if (age >= 20 an age <= 29):
    print("You are in your twenties")
```

Precedence of And, Or

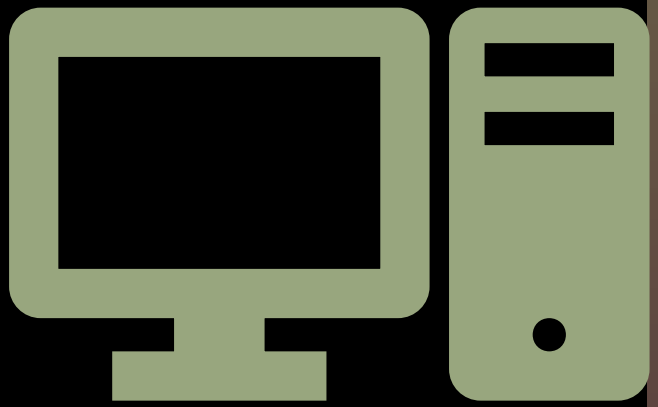
- A more complex example. We want to test if the user is called “Jim” but is not in their twenties. You might try this:

```
if (age < 20 or age > 29 and name == "Jim")
```

- But in the operator precedence, and comes before or, so the statement above actually says:

```
if (age < 20 or (age > 29 and name == "Jim"))
```

- Which really tests:
  - If the user is younger than 20 (with any name)
  - Or the user is older than 29 and called Jim



- Again, brackets are important. Use them when mixing and and or operators

```
if ((age < 20 or age > 29) and name == "Jim")
```

Reminder: ‘if’ and ‘else’ Statements

- Last week we used if-else statements to choose between two alternative blocks of code
- Here’s an example where we output a basic greeting for most people, but a special greeting for “Jim”:

```
if (name == "Jim")
    print( "Hey Jim, you owe me some money..." )
else:
    print( f"Hello {name}" )
```

Common Mistake: Unnecessary Condition

- Novice programmers sometimes try to write this:

```
if (name == "Jim"):
    print( "Hey Jim, you owe me some money..." )
else (name not "Jim"):
    print( f"Hello {name}" )
```

- There’s no need for the name != "Jim" expression
- The final else already catches all cases that didn’t match the earlier condition
- There isn't really a need to use parentheses here either, however for explanation purposes it's easier (both work in Python).

Chaining ‘if’ and ‘else’ Statements

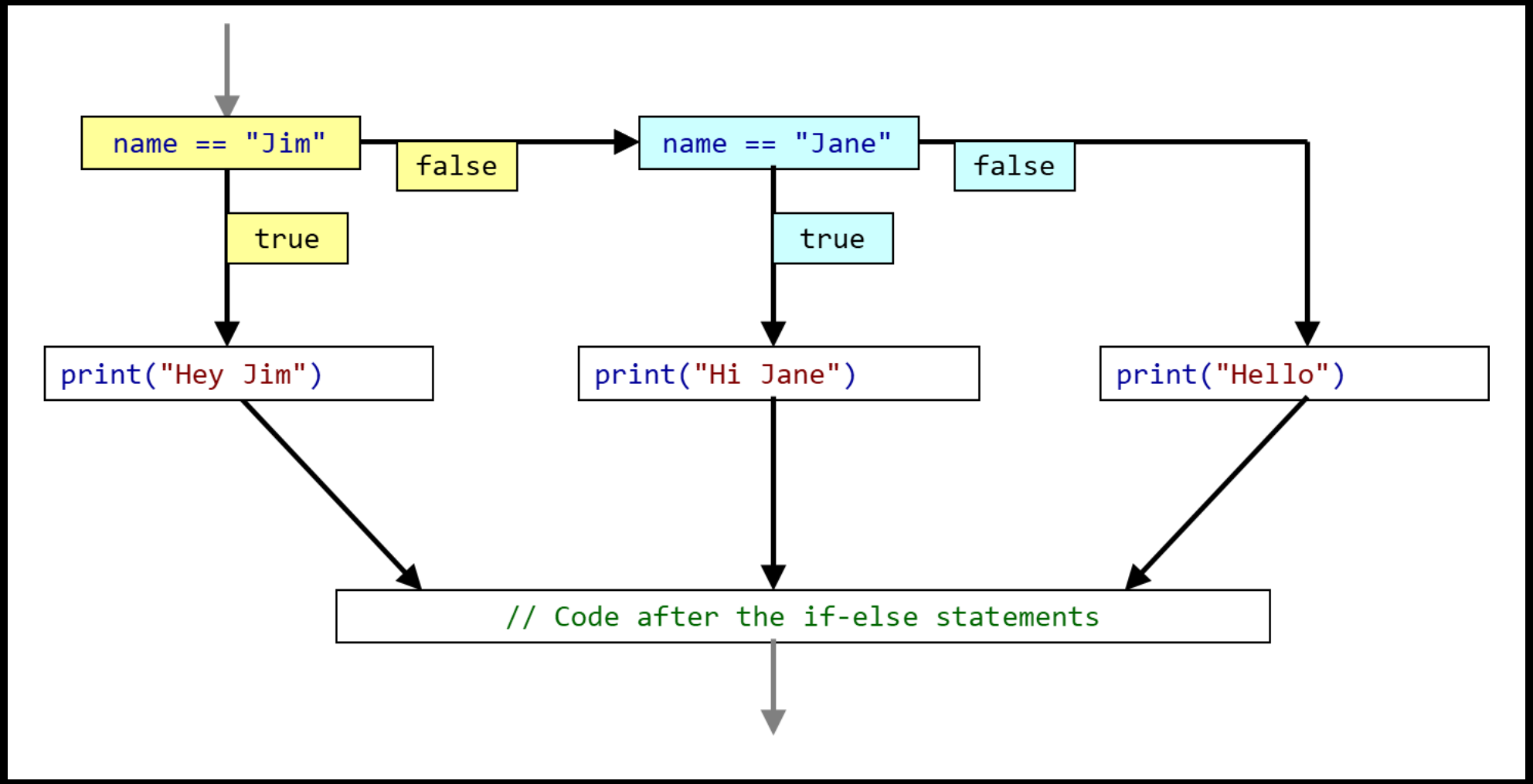
However, you can do something like the last slide to choose between more than two alternative blocks of code:

```
if ( name == "Jim" ):
    print( "Hey Jim, you owe me some money..." )
elif ( name == "Jane" ):
    print( "Hi Jane, have you got my phone?" )
else:
    # Final else catches everyone who isn’t called Jim or Jane
    print( "Hello {name}" )
```

But still the last else doesn’t need a condition, it catches everything else

Multiple ‘if-else’ Statements Visualised

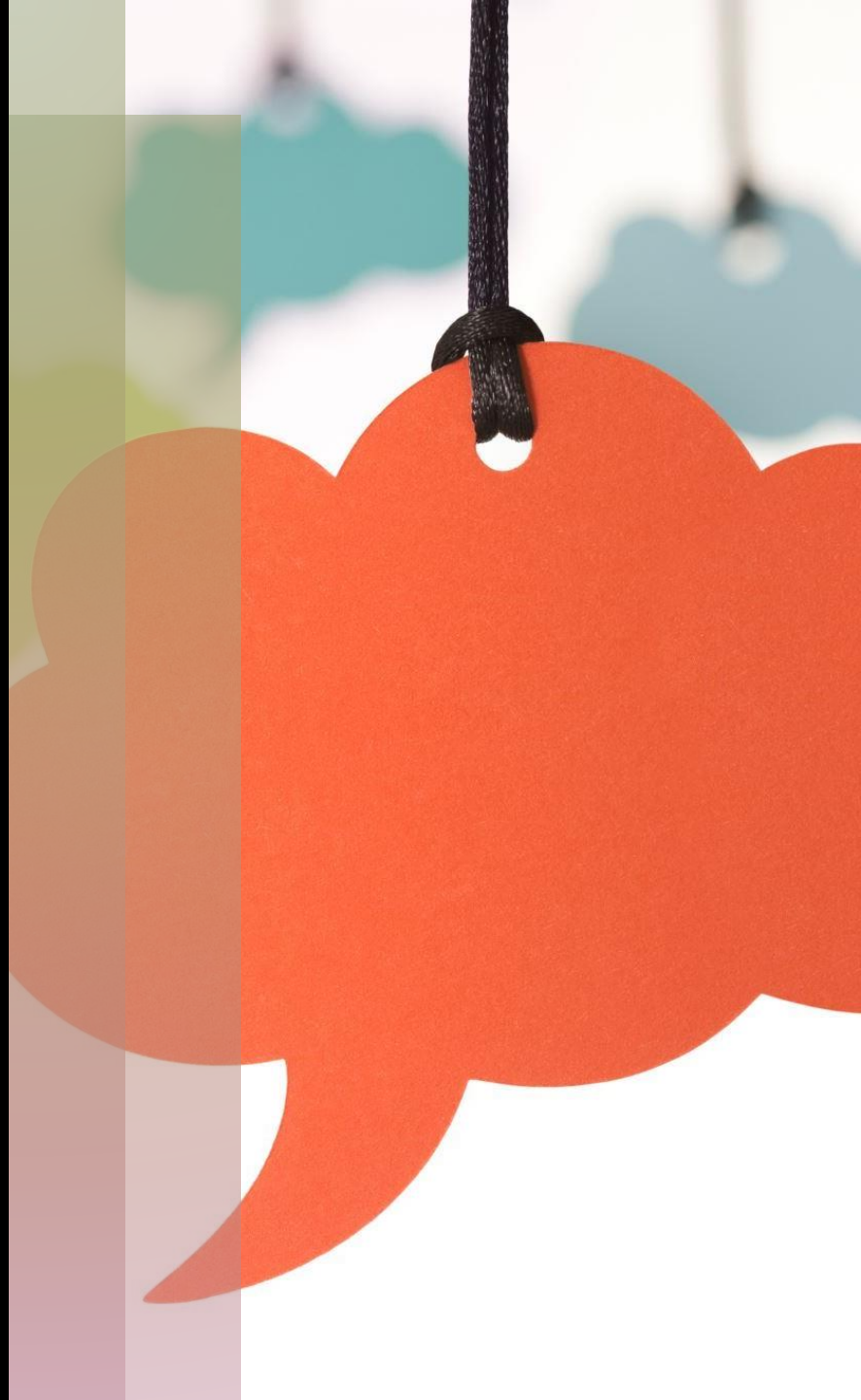
Flow of the `if-else` statement example:



Nested ‘if’ and ‘else’ Statements

- We can put if statements inside other if statements
  - Putting blocks of code inside other blocks is called nesting
  - Use this kind of code for more complex programs

```
if (name == "Jim"):
    print( "Hey Jim, you owe me some money..." )
        # Only for Jim
        if (day == "Tuesday" or day == "Thursday"):
            print( "Playing football later?" )
else:
    print( f"Hello {name}" )
```

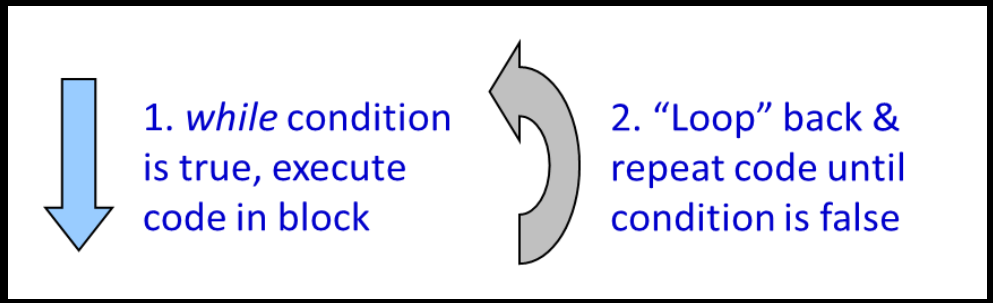


Loops: the 'while' Statement

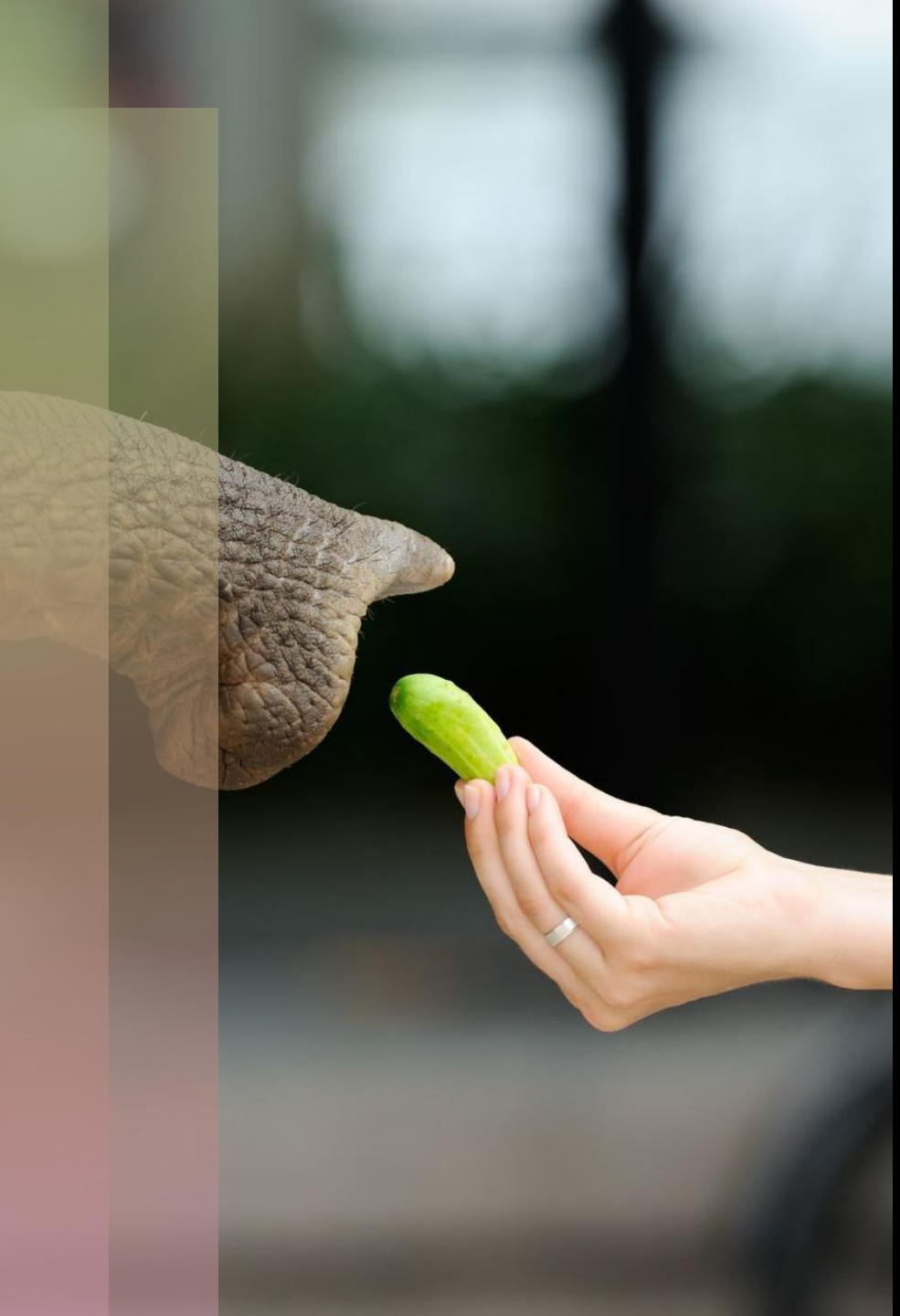
- We often wish to repeat the same piece of code multiple times. (For us we will be doing it mainly for input)
- The while statement is one way to do this in Python
- It has a similar form to an if statement.

```
while (condition):
    code to repeat

continue here when finished
```



- The code inside the block is repeated **while** the condition is true
  - Repeating of a block of code is called **looping**
- When the condition becomes false, the program finishes looping
  - And continues with whatever code follows the while block



Form of a ‘while’ Loop

- Here’s an example of a while loop that counts from 1 to 10:

```
counter = 1
while (counter <= 10):
  print( counter )
  counter += 1
```

- The variable counter starts at 1
- While the counter is less than or equal to 10 the code in the block is executed repeatedly
- The code in the block displays counter and increases it by 1
- So, after repeating ten times, counter will go beyond 10, which makes the condition become false and so the loop ends

Flow of a ‘while’ Loop

```
counter = 1

while (counter <= 10):
    print( counter )
    counter += 1
```

```
while (counter <= 10):
    print( counter )
    counter += 1
```

```
while (counter <= 10)
```

- counter starts at 1
- (1 <= 10) condition is true so the code in the block executes
- 1 is displayed and counter increases to 2
- Loop round and repeat code
- 2 <= 10 is true so the code executes again
- 2 is displayed and counter increases to 3
- …Repeat over and over until counter reaches 11
- 11 <= 10 is false so the loop finishes, program will continue after the while block
- So, this while loop will display the numbers 1 to 10

User Input in a ‘while’ Loop

- Here’s another example of a while loop, a guessing game:

```
userGuess = input("Guess the word: ")          # User’s first guess

while (userGuess != "tomato"):                 # Loops while the user is wrong
    userGuess = input("Wrong, guess again: ")  # User has another guess (and
                                               # another, and another etc.)
print("Correct")                               # Finally guesses "tomato" correctly
```

- Notice that if the user guesses right first time it will not execute the code inside the loop at all – it will skip directly to "Correct"

Summary

In today's lecture we have covered:

- A range of new operators:
  - Shorthand operators for changing variables: `+=` `-=` `*=` `/=` etc.
  - Logical operators for combined conditions: `and` `or`
- Operator precedence – which operators are calculated first
- More detail about `if-else` statements
- The `while` loop

Experienced programmers will know there are some other forms of loop, we will cover them next week