Making Decisions

Decision Making

- What we have learnt so far lets us write programs that run sequentially, one line after the other.
- This is fine for just doing calculations…
- ...but any interesting program is going to want to make decisions about those calculations!

Boolean Expressions

- We previously alluded to a *boolean* data type.
- A boolean takes one of two values: `True` or `False`.
    - Represents whether a logical statement is true or false.
    - Remember - Python is case sensitive. The T and F are capitalised!
- We can form boolean expressions using various operators.
- Comparison operators take two expressions of comparable data types, and gives a boolean output:
    - `x == y`
        - Equality - are two values equal?
        - Works as you would expect on all data types.
        - Do not confuse this with a single equals symbol!
        - x = y assigns the value of variable y to variable x.
    - `x != y`
        - Inequality - are two values not equal?
        - Opposite of equality.
    - `x < y`
        - Less than - is x less than y?
        - Works as you would expect for numeric values.
        - For strings, can be used to give a dictionary ordering.
        - Be careful with your datatypes! Consider:
            - `12 < 2` is False, because integer order.
            - `“12” < “2”` is True, because alphabetical order.
    - `x > y`
        - Greater than - is x greater than y?
    - `x <= y`
        - Less than or equal to.
    - `x >= y`
        - Greater than or equal to.
        - Make sure to get the two symbols in the correct order - the equals comes second!
- Logical operators take one or two boolean expressions and give a boolean output:
    - You may well recognise some of these if you have ever studied boolean logic!
    - `not x`
        - Inverts the value.
        - True becomes False.
        - False becomes True.
    - `x and y`

        - True if x and y are both True.
        - False otherwise.
    - `x or y`
        - True if one or both of x and y are True.
        - False otherwise.
    - Can summarise using truth tables.
- Boolean expressions built from these operators can be used to determine whether or not something is the case.
- Examples:
    - `name == “Alex”`
        - True if the variable name contains the string “Alex”.
    - `age >= 18 or loggedin`
        - True if the value of age is 18 or higher, or if the boolean variable loggedin is True.
        - There is no need for loggedin == True, since loggedin is already a boolean. (Although this would still be valid code.)
- Exercise:
    - Say that an item of food is described by the following variables:
        - `colour` - Colour of food, as a string.
        - `category` - Category of food, as a string.
        - `age` - How many days old the food is, as an integer.
    - For example, an apple I bought recently might have:
        - `colour=”red”`
        - `type=”fruit”`
        - `age=2`
    - Write a boolean expression that is True for all food items that are not vegetables, are either brown or yellow, and are less than a week old; and False for everything else.
    - Test your expression by outputting it for different combinations of values of the variables.
    - Answer:
        - `type != “vegetable” and (colour == “brown” or colour == “yellow”) and age < 7`

Simple If Statements

- We can use boolean expressions to make decisions about whether some code should be run.
- We do this using the `if` statement.
- In its simplest form:
    - `if` *`boolean-expression`* `:` *`statement`*
- What this does:
    - The boolean expression is evaluated.
    - If it is True, then the statement is run.
    - If it is False, then the statement is not run.
- For example:
    - `if age < 18 : print(“You are not old enough!”)`

    - The message will only be displayed if age is less than 18.
- Exercise:
    - Ask the user for the secret word.
        - (Recall the `input` function allows user input to be taken.)
    - If they enter the correct secret word, show them a message.
    - Answer:

        -
          ```
          word = input(“Enter the secret word: “)
          if word == “swordfish” : print(“Correct!”)
          ```

<u>Block If Statements</u>

- Usually, we will want to run more than one statement after a successful `if`.
- In Python, we can form blocks of code using *indentation*.
    - Indented code after a statement ending with a colon - such as `if` - forms a block.
    - Everything indented to the same level is part of the block.
    - The block ends once the indentation ends.
    - You can use however much indentation you want, but the standard in Python 3 is *four spaces*. The important thing is to be consistent.
- By creating a block, we can run several lines of code on the condition that a boolean expression is true.
- Example:

    -
      ```
      word = input(“Enter the secret word: “)
      if word == “swordfish”:
          print(“Correct!”)
          print(“Well done for knowing the secret word.”)
      print(“Goodbye”)
      ```
    - If the correct word is entered, several lines are outputted.
    - “Goodbye” is outputted regardless, since it is not part of the code block.

<u>Else</u>

- Rather than only carrying out a task if an expression is true, and doing nothing otherwise, we may want a choice of possible actions.
- An `if` statement can be followed by an `else` statement, as follows:

    -
      ```
      if boolean-expression:
          ... do stuff ...
      else:
          ... do other stuff ...
      ```
- The else block is run if the boolean expression is false. i.e. One of the two blocks is run.
- Exercise:
    - Modify the previous example where a secret word was asked for.
    - If the correct word is entered, output several lines of messages, using a code block.
    - If the wrong word is entered instead, output a warning message, using `else` and a code block.
    - Answer:

        -
          ```
          word = input(“Enter the secret word: “)
          if word == “swordfish”:
              print(“Correct!”)
              print(“Well done for knowing the secret
          word.”)
          else:
              print(“That’s not the right word!”)
          ```

Elif

- What if what we want to decide is more complicated than a simple yes or no answer?
- An `elif` statement, with a block, can be used to mean ‘else if’:

    -
      ```
      if boolean-expression:
          ... do stuff ...
      elif boolean-expression:
          ... do other stuff ...
      elif boolean-expression:
          ... do some other stuff ...
      else:
          ... do some other other stuff ...
      ```
- We can have as many `elif`s as we need, but they must come after an `if`.
- There can only be one `else`, and it must come last.
- The interpreter tries each condition in order, until it reaches one that is True.
- Example:

    -
      ```
      shape = input(“Enter a shape: “)
      if shape == “triangle”:
          print(“Has three sides.”)
      elif shape == “square” or shape == “rectangle”:
          print(“Has four sides.”)
      elif shape == “pentagon”:
          print(“Has five sides.”)
      elif shape == “hexagon”:
          print(“Has six sides.”)
      else:
          print(“I don’t know that shape!”)
      ```

Nested Blocks

- It is possible to use multiple levels of indentation to *nest* blocks inside other blocks.
- Example:

    -
      ```
      age = int(input(“Enter your age: “))
      if age >= 18:
          word = input(“Enter the secret word: “)
          if word == “swordfish”:
              print(“Welcome!”)
          else:
              print(“That’s not the secret word!”)
      ```

```
        else:
            print("You are not old enough!")
```

  - We can see from how the indentation lines up which code blocks match with which `if`s and `else`s.
- There are other statements we will see later which use blocks - the same principles about indentation and nesting apply there too.

Exercise

- Based on what you have learnt about decision making, write a simple quiz program.
- The program should ask the user some questions, and the user should input their responses.
- The program should tell the user whether they answered correctly.
- The program could also keep a score of how many questions were answered correctly, to output at the end.
- Try various types of questions to make sure you fully understand how boolean expressions, `if`, `else`, and `elif` work!

Short Circuit Evaluation

- Before we move on, we shall consider a special feature of boolean expressions, present in Python and many other languages.
- This is a slightly more advanced concept, so do not worry if you do not entirely follow it! It is simple useful to know it exists.
- Consider boolean expressions using logical `and` from earlier.
  - `x and y`
- You would expect this to be evaluated as follows:
  - Evaluate the boolean expression `x`.
  - Evaluate the boolean expression `y`.
  - Finally, evaluate the boolean expression `x and y`.
- But what if `x` is False?
  - No matter what `y` evaluates to, the whole expression `x and y` can only be False.
  - So why bother calculating `y` at all?
  - The interpreter will, in fact, not evaluating the second expression if the first is False, because there is no need!
- A similar scenario exists for `x or y`: if `x` is True, then no matter what `y` is, the whole expressions must be true, so there is no need to calculate `y`.
- This phenomenon is known as *short circuit evaluation* - where the latter calculation is *short-circuited* if the former calculation decides the answer immediately.
- We can use this to write more succinct code.
- Consider a program that needs to divide one number another and compare the answer to a value.

  -
    ```
    if a / b == 4:
        print("something")
    ```
  - This is unsafe code! If `b == 0`, then the program will crash, as division by zero is impossible.
  - We might prevent this by first checking the value of `b`:

-

    -
      ```
      if b != 0:
          if a / b == 4:
              print(“something”)
      ```
    - This code is safe, but required several levels of nested indentation.
    - We could make it better using short circuit evaluation:

    -
      ```
      if b != 0 and a / b == 4:
          print(“something”)
      ```
    - This code does what we want, and is completely safe.
        - If `b` is not zero, then the first expression is True, so we must still test the second expression - which is safe since `b` is not zero.
        - If b is zero, then the first expression is False, so the interpreter can short-circuit the second expression - `a / b` never gets calculated, which is just as well as it would crash the program.
- When used knowingly and correctly, short circuit evaluation can be a useful and powerful tool to de-clutter your code.
- However, watch out for it appearing where you don’t expect it, causing pieces of code not to be evaluated when you thought they would be!

Next lesson: Simple Loops