M30299 – Programming
Lecture 13 – Booleans and Further Loops

Matthew Poole & Nadim Bakhshov
moodle.port.ac.uk

School of Computing
University of Portsmouth

2020/21

Introduction to lecture

- Last week we introduced control structures by looking at `if` statements and `for` loops.
- In the previous lecture, we introduced the `while` loop, and saw some simple examples of its use.
- If statements and `while` loops depend on Boolean expressions to make decisions.
- In this lecture:
  - we'll look more closely at the Boolean data type; and then
  - look at some more examples of `while` loops, and consider issues to bear in mind when designing loops.

A review of the Boolean data type

- Recall some of the operators that we've used in `if` statements and `while` loops which give a Boolean result:

```
>>> month = "December"
>>> year = 2020
>>> year < 2025
True
>>> year != 2020
False
>>> month == "February"
False
>>> month <= "February"
True
```

Boolean operators

- In the practicals we introduced the operators "`and`" and "`or`" for combining Boolean expressions. There is also a "`not`" operator.
- These are the operators of the Boolean data type, and can be described using **truth tables**:

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

a	not a
True	False
False	True

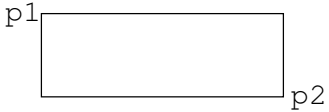
- Let's see some examples of their use ...

Boolean operators

```
>>> month = "December"
>>> year = 2020
>>> month == "December" and year > 2000
True
>>> month == "December" and year == 2000
False
>>> month == "December" or year == 2000
True
>>> month == "November" or year == 2000
False
>>> not (month == "November")
True
```

Boolean operators

- Suppose we have a rectangle (e.g. in a graphics window), specified by top-left point p1 and bottom-right point p2:



- We can determine if a point p is inside the rectangle (or on an edge) as follows:

```
if p.getX() >= p1.getX() and p.getX() <= p2.getX() and \
   p.getY() >= p1.getY() and p.getY() <= p2.getY():
    print("It's inside or on an edge")
else:
    print("It's outside")
```

Boolean operators

- A common mistake is to write code like:

```
x > 10 and < 20      instead of      x > 10 and x < 20
```

- The code on the left gives a syntax error at the <. Why?
- A similar mistake is to write:

```
r == "y" or "Y"      instead of      r == "y" or r == "Y"
```

- The code on the left will always be True, since Python interprets any non-empty string (such as `"Y"`) as True.
- Like numerical operators, Boolean operators obey precedence rules: `not` is highest, then `and`, then `or`.
- Parentheses can be used to enforce execution order.

Boolean operators

- Sometimes, good use of Boolean operators can simplify `if` statements and `while` loops (e.g. by removing nested `if`'s).
- As an example, recall the `isLeapYear` function that used nested `if-else` statements to test whether a year is a leap year.
- It returned True for leap years and False for non-leap years.
- (Years divisible by 4 are leap years except those divisible by 100; however, years divisible by 400 are also leap years.)
- This function can be written much more easily as follows:

```
def isLeapYear(y):
    return (y % 4 == 0 and y % 100 != 0) or y % 400 == 0
```

Other `while` loop patterns

- Let's look at some other "patterns" of `while` loops that you are likely to see and use very often.
- These patterns can be illustrated using a typical programming problem: **validating** the user's input.
- Our programs should be able to deal properly with invalid input, and thus provide good user-interfaces.
- (They should also be **robust**, which means that it should be difficult for the user to make them crash.)
- We will use an example of obtaining a number between 1 and 10 from the user, and present a few solutions to this problem.

Validating user input

- The simplest way to obtain a valid number from the user is:

```
number = any invalid value
while number is invalid:
    number = int(input(...))
```

- `number` is initialised to an invalid value to force the 1st execution of the loop body; the loop exits once a valid value is entered.
- Using this technique, we can write:

```
def getOneToTen():
    number = 0
    while number < 1 or number > 10:
        number = int(input("Enter a number: "))
    return number
```

Validating user input

- This function operates as follows:

```
>>> getOneToTen()
Enter a number: -5
Enter a number: 13
Enter a number: 6
6
```

- The main problem is that the user is given no feedback when they enter an invalid value.
- This is easily solved by placing an `if` statement within the body of the loop...

Validating user input

```
def getOneToTen():
    number = 0
    while number < 1 or number > 10:
        number = int(input("Enter a number: "))
        if number < 1 or number > 10:
            print("That's not between 1 and 10")
    return number
```

- Now, we get:

```
Enter a number: 13
That's not between 1 and 10
Enter a number: 6
6
```

Using a Boolean continuation variable

- However, it seems to suffer from lack of style! ... the condition "`number < 1 or number > 10`" is repeated.
- One way to avoid this would be to introduce a Boolean variable to use as the loop condition:

```
def getOneToTen():
    invalidNumber = True
    while invalidNumber:
        number = int(input("Enter a number: "))
        invalidNumber = number < 1 or number > 10
        if invalidNumber:
            print("That's not between 1 and 10")
    return number
```

The break statement

- Another solution is to use a new feature in Python — break.
- The break keyword is used only within loops. Its effect is to immediately exit from the enclosing loop.
- Using a break, we can simplify the loop condition to True, and break out of the loop when a valid number is entered:

```
def getOneToTen():
    while True:
        number = int(input("Enter a number: "))
        if number >= 1 and number <= 10:
            break
        print("That's not between 1 and 10")
    return number
```

Design guidelines for while loops

- The decision as to whether to use:
  - `while True` at the beginning of a loop; and/or
  - a `break` within a loop

  should be made based on readability of the code.
- Having a "proper" (i.e. not just `True`) loop condition is usually preferred since it makes loops more readable.
- I suggest using `while True` and/or `break` only when the alternatives would involve writing repetitive code.
- Note also that in the above code, I could simply have put the code '`return number`' where the `break` is. Why didn't I?