COMP 10280
Programming I (Conversion)

John Dunnion

School of Computer Science
University College Dublin

COMP 10280 Programming I (Conversion)/Lecture 8

Outline

Conditional statement (1)

```
if num > 0:
    print('Number is positive.')
elif num == 0:
    print('Number is equal to 0')
else:
    print('Number is negative.')
```

Conditional statement (2)

```
if num == 0:
    print('Number is equal to 0')
elif num > 0:
    print('Number is positive.')
else:
    print('Number is negative.')
```

Conditional statement (3)

```
if num > 0:
    print('Number is positive.')
elif num == 0:
    print('Number is equal to 0')
elif num < 0:
    print('Number is negative.')
```

Boolean conditions (1)

- We have already seen the three Boolean operators: `and`, `or` and `not`
- These can be used to create complex Boolean conditions

```
if num_hours < 0 or num_hours > 168:
    print('Number of hours worked per week shoul
      positive and be a maximum of 168!')
```

Boolean oconditions (2)

- Consider the following:

```
if num > 20:
    if num % 2 == 0:
        print('Number is even and greater than 20'
```

- and

```
if num > 20 and num % 2 == 0:
    print('Number is even and greater than 20')
```

Boolean oconditions (3)

- Consider the following:

```
if num > 20 and num % 2 == 0:
    print('Number is even and greater than 20')
```

- and

```
if num > 20 or num % 2 == 0:
    print('Number is even and greater than 20')
```

Boolean oconditions (4)

- Consider the following:

```
if num_hours < 0 or num_hours > 168:
    print('Number of hours worked per week '
    'should be positive and be a maximum of 168!')
```

- and

```
if num_hours < 0 and num_hours > 168:
    print('Number of hours worked per week '
    'should be positive and be a maximum of 168!')
```

Short-circuiting the evaluation of Boolean conditions (1)

- In Python, Boolean expressions are evaluated from left to right
- If the Python interpreter works out that it knows the result of a Boolean expression, ie there is no point in evaluating the rest of an expression, it stops the evaluation and does not carry out the computations in the rest of the expression
- When the evaluation of a Boolean expression stops because the overall value is already known, it is called short-circuiting the evaluation.

Short-circuiting the evaluation of Boolean conditions (2)

```
>>> a = 30
>>> b = 5
>>> a >= 20 and a / b > 4
True
>>> a = 10
>>> b = 2
>>> a >= 20 and a / b > 4
False
```

Short-circuiting the evaluation of Boolean conditions (3)

```
>>> a = 10
>>> b = 0
>>> a >= 20 and a / b > 4
False
>>> a = 25
>>> b = 0
>>> a >= 20 and a / b > 4

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    a >= 20 and a / b > 4
ZeroDivisionError: division by zero
```

Short-circuiting the evaluation of Boolean conditions (4)

- The fourth calculation failed because Python was evaluating `a / b` and `b` was 0 which caused a runtime error
- The third calculation did not fail because the first part of the expression, `a >= 20`, evaluated to `False` and thus the expression `a / b` wasn't evaluated because of the short-circuit rule
- The Boolean condition in this example can be constructed to strategically place a guard expression just before the sub-expression that might cause an error

Short-circuiting the evaluation of Boolean conditions (5)

- Consider the following:

```
>>> a = 30
>>> b = 5
>>> a >= 20 and b != 0 and a / b > 4
True
>>> a = 10
>>> b = 2
>>> a >= 20 and b != 0 and a / b > 4
False
```

Short-circuiting the evaluation of Boolean conditions (6)

```
>>> a = 30
>>> b = 0
>>> a >= 20 and b != 0 and a / b > 4
False
>>> a = 30
>>> b = 0
>>> a >= 20 and a / b > 4 and b != 0

Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    a >= 20 and a / b > 4 and b != 0
ZeroDivisionError: division by zero
```

Short-circuiting the evaluation of Boolean conditions (7)

- In the second calculation, `a >= 20` is `False` and the evaluation stops at the first `and`
- In the third calculation, `b != 0` is `False` and the evaluation stops at the second `and`
- We say that the expression `b != 0` acts as a guard to ensure that we only execute `a / b` if `b` is non-zero
- In the fourth calculation, the expression `b != 0` is *after* the expression `a / b`, and so the expression fails with an error

Chained expressions

- In Python, comparisons can be chained
- `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all if `x < y` is found to be false)
- Formally, if `a`, `b`, `c`, `...`, `y`, `z` are expressions and `op1`, `op2`, `...`, `opN` are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b and b op2 c and ... and y opN z`, except that each expression is evaluated at most once.
- Note that `a op1 b op2 c` does not imply any kind of comparison between `a` and `c`, so that `x < y > z` is perfectly legal
- Note that such chained expressions will not work in most other languages!

Leap years (1)

- (From Wikipedia!)
- A leap year (also known as an intercalary year or a bissextile year) is a year containing one additional day (or, in the case of lunisolar calendars, a month) added to keep the calendar year synchronized with the astronomical or seasonal year
- Because seasons and astronomical events do not repeat in a whole number of days, calendars that have the same number of days in each year drift over time with respect to the event that the year is supposed to track
- By inserting (also called intercalating) an additional day or month into the year, the drift can be corrected
- A year that is not a leap year is called a common year

Leap years (2)

Algorithm (from Wikipedia)
The following pseudocode determines whether a year is a leap year or a common year in the Gregorian calendar:

if (year is not exactly divisible by 4) then (it is a common year)
else
if (year is not exactly divisible by 100) then (it is a leap year)
else
if (year is not exactly divisible by 400) then (it is a common year)
else (it is a leap year)

Leap years (3)

Prompt the user for a year
Read the year
if *year* $\geq$ *0* ***then***
    if *(year mod 4 = 0* ***and*** *year mod 100 != 0)*
        or *year mod 400 = 0* ***then***
        Print(“Year is a leap year”)
    else
        Print(“Year is not a leap year”)
else
    Tell the user that the year must be $\geq$ *0*
Program finishes

Leap years (4)

```
# Ask the user to enter a year
year = int(input('Enter a year: '))

print('Year entered:', year)

if year >= 0:
    if (year % 4 == 0 and year % 100 != 0)
                or year % 400 == 0:
        print(year, 'is a leap year.')
    else:
        print(year, 'is not a leap year.')
else:
    print('Year must be greater than 0.')

print('Finished!')
```

Strings and sequence types

- Strings are one of several sequence types in Python
- Strings share a number of operations with all sequence types

Length of a string

- The length of a string can be found using the `len` function

```
>>> x = 'abcdef'
>>> len(x)
6
>>> len('abcd')
4
>>>
```

Indexing into a string (1)

- Individual characters can be extracted from a string by indexing
- In Python, all indexing starts at 0

```
>>> x = 'abcdef'
>>> x[1]
'b'
>>> x[0]
'a'
>>> 'abcd'[2]
'c'
>>> x[6]

Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    x[6]
IndexError: string index out of range
```

Indexing into a string (2)

- Since Python uses 0 to access the first element of a string, the last element of a string of length 4, for example, is accessed using the index 3
- Negative numbers are used to index from the end of a string

```
>>> x = 'abcdef'
>>> x[-1]
'f'
>>> x[-6]
'a'
>>> x[-7]

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    x[-7]
IndexError: string index out of range
>>>
```

Slicing a string (1)

- Extracting a substring from a string is called slicing
- If `s` is a string, the expression `s[start:end]` denotes the substring of `s` that starts at index `start` and ends at index `end - 1`
- The slice ends at index `end - 1` so that expressions such as `s[0:len(s)]` have the value you might expect

```
>>> x = 'abcdef'
>>> x[3:5]
'de'
>>> x[0:len(x)]
'abcdef'
>>> x[3:3]
''

>>>
```

Slicing a string (2)

- If the value before the colon is omitted, it defaults to 0
- If the value after the colon is omitted, it defaults to the length of the string
- If both values are omitted, both defaults apply
- Thus `s[:]` is the same as `s[0:len(s)]`

```
>>> x = 'abcdef'
>>> x[:4]
'abcd'
>>> x[3:]
'def'
>>> x[:]
'abcdef'
>>>
```