Fundamentals of Computer Programming

Lecture slides - Control Statements

Intro

- This lesson covers
  - Boolean expressions
  - Decision making using *if* statements
  - Using membership testing
  - Logical operators within Boolean expressions
  - Ternary Operators
  - Iteration using *while* and *for*

First Steps to Programming

- Up until now we have been learning about *expressions*, *variables*, *data-types*, *basic I/O* and calling *functions*
- But programming is actually more about implementing **algorithms**
- It is the algorithms within a program define the “logic” needed to complete tasks
- The code we have written so far has had very little “logic”, it has mainly been linear sequences of operations
- All 3GL type languages provide very similar constructs to support what are known as **control statements** that allow us to implement algorithms

Boolean Expressions

- All algorithms are implemented using *sequence*, *selection* and *iteration*
  - ***sequence*** relates to the execution of one instruction after another
  - ***selection*** relates to deciding which sections of code should be executed
  - ***iteration*** relates to performing actions repeatedly, within some sort of loop

- Both *selection* and *iteration* often base their logic (i.e. which code to execute, or how many times to loop) on the result of a **Boolean expression**
- *Boolean expressions* compare values using *relational* operators
- Unlike *arithmetic* operators such as '`+`' or '`*`', the result is always **Boolean** (`True` or `False`)

Python's Relational (comparison) Operators

Algebraic operator	Python operator	Sample condition	Meaning
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Note: these have lower *precedence* than arithmetic operators, == and != are lowest of all.

Selection: The “if” Statement

- The **if** statement uses a *condition* to decide whether to execute a statement (or a group of statements)
- Each **if** statement consists of the keyword `if`, the *condition* to test, and a colon (:) followed by an *indented* code block
- Each **code block** must contain one or more statements, e.g.

```
if number1 > number2:
    print(number1, "is greater than", number2)

print("This is NOT part of the if block")
```

The if...else Statement

- The **if** statement can include an optional **else** clause
- When present, the `else` keyword should be followed by a colon (:) and an *indented* code block
- The code following the `else` is executed if the given *condition* evaluates to `False`, i.e. when the code associated with the **if** doesn't execute

```
if number1 > number2:
    print(number1, "is greater than", number2)
else:
    print(number1, "is less or equal to", number2)
```

More on the ‘if’ Statement

- If there is more than one *condition* to be checked, `if` statements can be *chained* together, this is done using the **elif** keyword, e.g.

```
if number1 > number2:
    print(number1, "is greater than", number2)
elif number1 == number2:
    print(number1, "is equal to", number2)
else:
    print(number1, "is less than", number2)
```

- Any number of `elif` blocks can appear between the first `if` and the final `else` (which is optional). Note: Python does not support C style *switch/case* type statements.

Logical Operators

- It is sometimes necessary to base a decision on more than one particular *condition*, e.g. is a person aged between 18 and 65?
- *Boolean expressions* can include **Logical Operators** that allow us to test for multiple *conditions* within a single expression
- Logical operators allow us to logically *combine* the result of several other expressions in order to get a single result
- Logical operators in Python are represented by the keywords '`and`', '`or`' and '`not`'
- Do not to confuse *logical* operators, with the *bitwise* operators (`&` and `|`)

Logical Operator Examples

- '**and**' returns `True` if *both* operands evaluate to `True`

```
if age >=18 and age <=65:
    print("You are within typical working age")
```

- '**or**' returns `True` if *either* operand evaluates to `True`

```
if age <18 or age >65:
    print("You probably do not work at the moment")
```

- '**not**' returns `True` if the operand evaluates to `False` (and vice-versa)

```
if not male:
    print("You are probably female")
```

Logical Operator Examples

- Logical operators can be further combined, e.g.

```
if age >=18 and age <=65 or male:
    print("You are of working age, or a male of any age")
```

- Be careful of operator *precedence*, these logical operators have a lower priority than almost all other operators, `not` is the highest, followed by `and`, then `or`.
- *Hint*: always use parentheses `()` when using logical operators, even if they just add clarity, e.g.

```
if (age >=18 and age <=65) or male:
    print("You are working age, or a male of any age")
```

Relational Operator Chaining

- Python allows relational operators to be *chained*, e.g.

```
if 99 < x <= 1000:
    print("x is between 100 and 1000")
```

- Chaining relational operators is equivalent to using the logical '**and**' operator, e.g. the above is the same as -

```
if 99 < x and x <= 1000:
    print("x is between 100 and 1000")
```

- Longer chains are possible, but this can lead to expressions that are hard to understand, e.g.

```
if 99 < x <= y > 200:
    print("x is between 100 and y, which is more than 200")
```

Membership Testing

- As well as using relational operators to specify *conditions*, Python provides **membership test** operators
- These are useful when attempting to find out whether a compound type (such as a *string* or *list*) contains a specific value
- A *membership test* is performed using the `in` or `not in` operators, e.g.

```
if "Eric" in names:       # assuming 'names' is a list of names
    print("Eric is in the list of names")


if word not in sentence:  # assuming sentence is a string
    print("The word", word, "is not in the sentence")
```

- When testing a string, `in` will return `True` if the given value appears anywhere as a sub-string, e.g. `"Jo" in "I am John"` will return `True`

Using Non-Boolean Expressions

- If an expression used as a *condition* results in a *numerical* type result, then Python bases the decision on whether the value evaluates to `0` or <>`0`
- The value of `0` is equivalent to `False`, and <>`0` is equivalent to `True`, e.g.

```
x = 1
if x:
    print("This will ALWAYS execute")
```

- Also, an empty *sequence* (such as a *string* or *list*) is equivalent to `False`, and a non-empty one is equivalent to `True`, e.g.

```
if answer:
    print("You entered an answer of", answer)
else:
    print("You did not enter an answer")
```

Ternary Operator

- A **ternary** operator is used to *return* a value based on a *condition*
- It looks similar to an **if...else** control statement, but is more concise
- Rather than execute blocks of code a ternary operator evaluates and returns one of two possible values, the syntax is as follows -

```
[true_value] if [condition_expr] else [false_value]
```

- The first value is returned if the *condition expression* evaluates to `True`, and the second value is returned if it evaluates to `False`
- Notice how the first value appears before the '`if`' keyword and the second value appears after the '`else`' keyword

Ternary Operator : example

- The following ternary operator assigns the highest of either value ‘a’ or ‘b’

```
highest = a if a > b else b
```

- This is equivalent to the following **if** control statement -

```
if a > b:
    highest = a
else:
    highest = b
```

- The ternary operator version is clearly more concise, and can be embedded into other statements, e.g.

```
print("a is the highest" if a > b else "b is the highest")
```

Ternary Operator vs ‘if’ Control Statements

- A **ternary operator** returns a value, whereas an **if** control statement executes instructions within a code block
- A **ternary operator** must include the **else** keyword, allowing exactly *two* values to be specified
- A **ternary operator** can be embedded into other statements, and appear on a single line, whereas **if** control statements are multi-line statements
- **Ternary operators** can only be used as an alternative to **if** control statements in specific circumstances, i.e.
    - One of exactly two values needs returning
    - The returned value is to be assigned or used within another statement or expression

Iteration: The “while” Statement

- Within Python *iteration* can be implemented using several techniques
- The **while** keyword allows us to implement *loops*
- A loop is basically a block of code that executes over and over again
- A while statement repeats the loop *while* a *condition* is true
- As with the “if” statement, we typically specify the *condition* using a *Boolean expression*,
- As long as the boolean expression evaluates to `True`, the code within the block repeatedly executes

An Example of a ‘while’ Loop

```
x = 10
print("counting down from",x)
while x > 0:
    print(x)
    x = x - 1
print("countdown loop is complete!")
```

- The `x>0` is the *Boolean expression* used as the looping *condition*, notice the following ‘`:`’ symbol
- The two subsequent statements are *indented*, and thus are part of the while’s code block, i.e. they are executed *while* the *condition* holds `True`

Iteration: The “for” Statement

- As well as using the “while” statement, *iteration* can be achieved within Python by using the **for** statement
- Rather than iterate *while* a *condition* holds `True`, a “for” loop *iterates* over a sequence of values, such as the elements within a list, e.g.

```
names = ["Terry", "John", "Michael", "Eric", "Terry", "Graham" ]
for n in names:
    print(n)
```

- In this example, the variable `n` takes the value of each list element in-turn
- When executed, the above *for loop* will *iterate* a total of six times

Using the range() Function

- The **range()** function is often used with the “for” statement to allow iteration over a range of numbers
- The range function generates an *arithmetic progression*, e.g. to print the values between 0 to 9 we could use the following,

```
for next in range(10):
    print(next)
```

- Notice that the resultant values exclude the upper range value
- We can also specify a range that does not start at 0, e.g.

```
for next in range(20,40): # print value 20 to 39
    print(next)
```

More About range()

- As well as a *lower* and *upper* limit, the **range()** function allows a ‘step’ to be specified (as a third parameter)

```
for next in range(10,20,3):
    print(next)
```

- This will output: `10 13 16 19`
- The ‘step’ value can also be negative, e.g.

```
for next in range(-2,-10,-2):
    print(next)
```

- This will output: `-2 -4 -6 -8`

Breaking a Loop

- It is sometimes desirable to terminate a loop prior the usual end condition
- This is known as *breaking out* of a loop, and uses the keyword `break`
- When a `break` is encountered the enclosing loop finishes immediately, e.g.

```
# search a list of "names", for a specific "name"
for n in names:
    if n == name:
        print("found",name)
        break
```

- In the example, the loop will terminate (break) as soon as the current name in the list, matches that of the `name` variable

"break" and "else"

- There is sometimes a need to execute specific code only when a loop terminates normally, i.e. it does not terminate due to a `break` statement
- Python supports this by allowing an `else` statement to be associated with a `for` or `while` statement, e.g.

```
for n in names:
    if n == name:
        print("found",name)
        break
else:
    print("Did not find",name)
```

- In this example, if the loop terminates normally (not via the break) then the message `"Did not find …"` will be displayed

Continuing a Loop

- It is sometimes desirable to force the next iteration of a loop to begin immediately, i.e. prior to all the code within the block executing
- This can be done using the keyword `continue`
- When a `continue` is encountered the next iteration of the loops begins

```
for next in names:
    if "a" in next:
        count += 1
        continue     # start the next iteration immediately
    print("The name", next, "does not contain the letter 'a'")

print("the number of names containing the letter 'a' is", count)
```

Nested Control Statements

- Up until the last few slides, most examples have contained control statements that have had a single level of indentation
- When implementing certain algorithms however it is often necessary to include control statements within code blocks of other control flow statements
- This is known as **nesting**, where blocks of code contain control statements that themselves have blocks of code
- The `break` and `continue` examples contained one *level* of nesting, but it is not unusual to see two or even three *levels* of nesting

Nested Loop Example

- The following is known as a *nested loop*, e.g.

```
num1 = 1
while num1 <= 10:
    for num2 in range(1,11):
        print(num1, " x ", num2, "=", num1 * num2)
    num1 += 1
```

- The amount of indentation identifies the *owning* control statement, e.g. in the above example `num1 += 1` belongs to the “while” loop (not the “for” loop)
- Whereas, the call to the `print()` function belongs to the “for” loop
- Beware: too many levels of nesting makes code harder to read and debug, deeply nested loops can also greatly affect performance

Summary

- Algorithms are implemented using *sequence*, *selection* and *iteration*
- The '**`if`**' control-statement and **ternary operator** support *selection*
- The '**`while`**' and '**`for`**' control-statements allow us to implement *iteration*
- Decisions within '`if`' and '`while`' statements are made by evaluating **Boolean expressions**, which often contain *relational* (comparison) operators, such as <, <=, ==, !=, etc
- Results within Boolean expressions can be combined using **Logical operators** such as '`and`', '`or`' and '`not`'
- The *code-blocks* associated with control-statements can be **nested**