CE151

Introduction to Programming

Part 2

Variables, Expressions and Conditional Statements

Identifiers 1

An ***identifier*** is a name used in a program to refer to a variable, class or function. Identifiers may contain one or more letters (upper- or lower-case), digits and/or underscore characters, but must not begin with a digit. It is recommended that in normal circumstances they should begin with a letter.

Valid identifiers include:

`printf`, `x`, `xx`, `x1`, `x2`, `main`, `x1`, `myIdent`, `my_ident`, `_y`, `_5`

The following are not valid identifiers:

`1x` (begins with digit), `x!`, `my-ident` (contain invalid characters)

Identifiers 2

To make programs readable identifiers should normally be given meaningful names.

Note that the Python language is case-significant so `id1` and `Id1` would be two different identifiers.

It is conventional to use lower-case letters for single-word or short identifiers. When an identifier comprises more than one word two conventions are commonly used: separation by an underscore or capitalisation of the first letter of all words other than the first, e.g. `first_name` or `secondName`.

Identifiers 3

Some words are reserved for use as key words in the Python language and cannot be used as identifiers. These are:

`and`, **`as`**, **`assert`**, **`break`**, **`class`**, **`continue`**, **`def`**, **`del`**, **`elif`**, **`else`**, **`except`**, **`finally`**, **`for`**, **`from`**, **`global`**, **`if`**, **`import`**, **`in`**, **`is`**, **`lambda`**, **`nonlocal`**, **`not`**, **`or`**, **`pass`**, **`raise`**, **`return`**, **`try`**, **`while`**, **`with`**, **`yield`**, **`True`**, **`False`**, **`None`**

In addition it is recommended that you avoid using the names of built-in functions or types such as **`print`**, **`input`**, **`type`**, **`int`**, **`float`**, **`set`**, **`list`** and **`str`** as names of variables. Although it is permissible to do so (e.g. we could write **`print = 3`**) problems are likely to occur. In general if any word is highlighted in colour when you type it in an IDLE window, you should not use it as an identifier.

Variables

A ***variable*** is used in procedural languages for the storage of data. It is essentially a name given to a memory location. (The name variable is not formally used in Python, but it is used in most programming languages, and most text books on Python use the term.) In many languages variables must be ***declared*** before they can be used but this is not the case in Python. We can simply create variables and give them initial values:

```
age = 35
name = "Mike"
height = 1.85
```

Types 1

Several different kinds of value can be stored in a variable. On the previous slide we saw an integer, a real number and a string of characters. In many languages (such as C and Java) the type is associated with the variable, so having stored an integer in the variable called `age` we could not later use the same variable to store a different type of data. However, in Python the type is simply associated with the data object.

A consequence of this is that type-checking in Python is performed at runtime, whereas in languages such as Java the compiler performs type checking.

Types 2

Python 3 has 11 built-in types; many others are defined in libraries distributed with the language, and programmers can define their own new types.

The built-in types are `int`, `float`, `complex`, `bool`, `str`, `list`, `tuple`, `range`, `dict`, `set` and `frozenset`.

The first four of the above are classed as the numeric types. Values of type `int` are integers whereas values of type `float` are real numbers. 8 and 8.0 are distinct; the former has type `int` whereas the latter has type `float`.

The only values of type `bool` are `True` and `False`: these are represented by 1 and 0 so we can manipulate them as if they were numbers. (We shall look at the other types later.)

Floating-Point Types 1

Real numbers are stored in computers using a ***floating-point*** notation; hence the type for real numbers is called `float`. (This name was first used in C and has subsequently been used in all languages based on C, including C++, C# and Java.)

Most real numbers cannot be represented precisely. In the same way as we cannot represent one-third as a finitely-written decimal number there are many numbers that cannot be represented precisely in Python. Since the floating-point notation is based on binary fractions the only numbers that can be represented precisely are multiples of one half, one quarter, one eighth and so on.

Floating-Point Types 2

As a consequence of the remarks on the previous slide numbers such as one tenth, which we can write as 0.1 using the decimal notation, cannot be expressed precisely in floating-point notation, so small rounding errors can sometimes occur when we perform real-number arithmetic.

The `float` type typically provides a precision of about 17 significant digits, with a range of about 10^{-308} to 10^{308}.(Anyone familiar with any of the languages of the C family should observe that this is the usual range and precision of the `double` type in those languages, so there is no need for a separate `double` type in Python.)

Assignment 1

Assignment is be used to store a value in a variable. The simplest form of assignment is a statement such as

```
age = 20
```

which will store the value 20 in the variable `age` (overwriting any previous value that was stored there, hence the phrase destructive assignment is sometimes used).

The left-hand side of an assignment must be some expression that denotes a memory location; there are other options as well as the name of a variable – we will encounter some of these later.

Assignment 2

The right-hand side an assignment may be an arbitrary expression, so we could use statements such as

```
age = 20+5
```

or

```
age = age+1
```

The latter assignment will add 1 to the current value of the variable `age` and store the result in the same variable; hence after performing this assignment the value stored in `age` will be one more than it was previously.

Assignment 3

We can assign the same value to several variables in the same statement:

```
yourAge = myAge = 21
```

It is also possible to assign different values to more than one variable in the same statement:

```
yourAge, myAge = 21, 50
```

This approach tends to make programs more difficult to read so it should normally be avoided. However one situation where it is very useful is if we want to swap the values in two variables:

```
a, b = b, a
```

Operators and Expressions 1

In addition to the `+` operator that we saw on slide 11, there are six more arithmetic operators in the Python language: `-`, `*`, `/`, `//`, `%` and `**`.

The `+` and `-` operators, as you would expect, are used to perform addition and subtraction. The operator for multiplication is `*`, since the only character on a keyboard that looks like a × sign is the letter x, and `x` is a valid identifier.

These three operators produce results of type `int` when both of their operands are of type `int`, but results of type `float` if one or more operand has type `float` so `6.5*2` would give the result 13.0, not 13, and `5.5-4.5` would give 1.0.

Operators and Expressions 2

The `/` operator is used for division. The result of dividing one integer by another could be an integer (e.g. 6/3 is 2) or a real number (e.g. 5/4 is 1.25). Since the programmer may need to know the type of the result of an expression such as `x/y` where `x` and `y` have values that have been supplied by the user, it is necessary that the operator always returns values of the same type; hence in Python 3 the value of a division expression is always of type float. (This is not the case in Python 2.)

The operator `**` is used for calculating powers: `3**4` denotes 3^4.

Operators and Expressions 3

Sometimes we want a division operation with two integers as operands to return an integer result, ignoring any remainder. (For example if we want to convert 376 pence into pounds and pence we need to divide 376 by 100 to generate 3, with a remainder of 76.)

To do this in Python 3 we need to use a different division operator, `//`. This always returns an integer result so the expressions `6//3`, `7//3` and `8//3` all have the value 2. In all cases the answer is rounded downwards so `9//-4` would give -3, not -2 (-2.25 gets rounded down, not towards 0).

The `%` operator gives the remainder that would be obtained had an integer division been performed so `6%3` has the value 0, `7%3` has the value 1 and `8%3` has the value 2.

Operators and Expressions 4

The expression on the right-hand side of the assignment statement

```
a = 3+b*4
```

is potentially ambiguous – it might mean add 3 to the result of multiplying the value of `b` by 4, or it might mean add together 3 and the value of `b` then multiply the result by 4. We can avoid such ambiguity by using parentheses – to obtain the first behaviour we could write

```
a = 3+(b*4)
```

and to obtain the second we could write

```
a = (3+b)*4
```

Operators and Expressions 5

The use of parentheses to avoid ambiguity has advantages and disadvantages: it makes simple expressions easier to understand but can make complex expressions with many parentheses hard to read.

Hence the normal practice is to use parentheses only when they are needed. In order to determine whether they are needed we need to know exactly what

```
a = 3+b*4
```

means. To do this we need to understand the ***precedence rules*** for operators in Python.

Operator Precedence 1

Each of the many operators in Python has a precedence level.

A partial list of operators and their precedence is

`(…)  […]  {…}`	(*tuple/list/dictionary creation*)
`[…]  .  (…)`	(*subscripting/slicing, attribute reference, function call*)
`**`	(***see notes on slide 20 and 22***)
`+  -`	(*unary, e.g.* `-5`)
`*  /  //  %`	
`+  -`	(*binary, i.e. addition/subtraction, e.g.* `x-5`)
`<  <=  >  >=  !=  == is  is not  in  not in`	
`not`	
`and`	
`or`	

(The complete list will be presented later in term.)

Operator Precedence 2

In the table on the previous slide operators at the top have the highest precedence and operators on the same line have equal precedence. Hence we see that addition and subtraction have lower precedence than the other arithmetic operations.

Note that the unary versions of `+` and `-` are for use in writing expressions such as `-7` or `-x` (the unary `+` is rarely used) and the addition and subtraction operations are the binary versions (they have two operands).

Also note that `is not` and `not in` are single operators, each with a space between two words.

Operator Precedence 3

The precedence level for the `**` operator is anomalous; if its right-hand operand is one of the three unary operators shown on the next line it (`**`) has lower precedence than that operand. This is because the only sensible interpretation of `x**-2` is `x**(-2)`. On the other hand, mathematicians would interpret the expression `-x**2` as `-(x**2)` and indeed Python treats it in this way.

Operator Precedence 4

Since multiplication has higher precedence than addition the previously-seen assignment

```
a = 3+b*4
```

means

```
a = 3+(b*4)
```

The higher precedence results in the multiplication being performed first.

However, the precedence table alone does not tell us the meaning of an assignment statement such as

```
a = 3-b+c
```

where the operators have equal priority.

Operator Associativity

When an expression contains adjacent operators with equal precedence we need to consider the ***associativity*** of the operators. With the exception of `**` and the unary operators, all Python operators associate from left to right.

This indicates that the leftmost occurrences have the highest precedence so the meaning of the expression `3-b+c` is `(3-b)+c`.

The power operator, on the other hand, associates from right to left so the meaning of `a**2**c` is `a**(2**c)`.

The unary operators have to associate from right to left; the only possible sensible meaning for `--5` is `-(-5)`.

Program Structure and Layout 1

A Python program is simply a sequence of ***statements***. Each of these is normally written on a separate line.

It is however, possible, to place more than one simple statement on a single line; in this case they must be separated by semicolons, e.g.

```
x = 3; y = 4
```

On some rare occasions it may be necessary to split a very long statement into more than one line so that it fits into a window or display area. In this case we use a \ character to indicate that the statement continues on the next line, e.g.

```
x = (3+4+5+6+7+8+9+10+11+12+13)*\
     (14+15+16+17+18+19+20)
```

Program Structure and Layout 2

When continuing a statement on a new line after a \ it is normal practice to indent the continuation line(s), and tools such as the IDLE development environment will do this automatically. However, this is the only situation in Python where the choice of whether to use indentation and how much, is left to the programmer.

Several non-simple types of statement occupy more than one line. In this case subsequent lines ***must*** be indented and the end of the sequence of indented lines indicates the end of the statement. Hence the rules for layout of programs in Python are much stricter than those for languages such as C and Java.

Comparison Operators 1

It is often necessary to compare numbers in programs. Python provides six operators to perform such comparisons. These produce results of type `bool`.

To check whether two numbers (or other data items) are equal we use the operator `==`. It is important to distinguish this from `=` which is the assignment operator.

For example the value of expression `x==3` will be `True` if the value stored in the variable `x` is 3, and `False` otherwise.

The operator `!=` is the "not equal" operator; `x!=3` will have the value `True` if and only if the value of `x` is not equal to 3.

Comparison Operators 2

The `<` and `>` operators are used to compare the size of numbers; `x<3` will have the value `True` if and only if the value of `x` is less than 3.

Mathematicians use the symbols ≤ and ≥ to mean "less than or equal to" and "greater than or equal to". Since these symbols are not available on the keyboard the Python language (like most others) uses `<=` and `>=` for these comparisons.

Note that when writing the operators `==`, `!=`, `<=` and `>=` there must be no spaces between the pair of characters.

Comparison Operators 3

Consider an examination where a mark of less than 50 denotes failure, a mark of 70 or more denotes a distinction and marks in the middle range denote a pass.

The condition for a standard pass would be written by mathematicians as $50 \leq mark < 70$. In Python we can simply write this condition in the form `50<=mark<70` (although this is not the case in most programming languages). An expression of this form is treated as if it had been written as `50<=mark and mark<70`. Hence the expression is not equivalent to `(50<=mark)<70` (which is what we would expect if we immediately applied the precedence rules).

Conditional Statements 1

Conditional statements are used when we wish to perform some action only if some condition is true.

The simplest form of conditional statement is

```
if e : S
```

where `e` is an expression and `S` is a statement, for example

```
if mark>=50 : print("Well done - you passed")
```

The program on the following slide will input an exam mark and output an appropriate message.

Conditional Statements 2

```
"""
marks1.py
"""

# input mark and convert to number

markString = input("Enter your mark: ")
mark = int(markString)

# output appropriate message

if mark<50 : print("You failed")
if 50<=mark<70 : print("Well done - you passed")
if mark>=70 : print("You got a distinction!")
```

Conditional Statements 3

Observe that the program on the previous slide contains some lines beginning with the `#` character – these are ***comments*** and are ignored by the interpreter. Comments should be used to say what each part of the program does so that if the programmer, or another programmer, wishes to modify it in the future he or she will find it easier to understand.

The `input` function is used to input one line of text into a string; this is used in an assignment statement to store the string in a variable. The string should contain a number (unless the user has typed an invalid mark) – we have to use the `int` function to extract a value of type `int` from the string.

Conditional Statements 4

We often wish to perform more than one statement if a condition is satisfied. To do this we have to write each statement on a separate line and the sequence of statements (known as a ***suite***) must be indented.

```
if mark<50 :
  print("You failed")
  if mark>=35 : print("You may resit")
if 50<=mark<70 : print("You passed")
if mark>=70 : print("You got a distinction!")
```

The amount of indentation is arbitrary but each line in the suite must be indented by the same amount. The usual practice is to use 4 spaces, but since these slides have limited width all examples will use 2 spaces.

Conditional Statements 5

The program on slide 29 performs in some cases some unnecessary comparisons. If the student has failed he or she cannot have also passed or got a distinction so if we have printed the failure message there is no need to perform any further checks.

We can rewrite the code using an alternative form of conditional statement

```
if e : S1
else : S2
```

In a statement of this form the statement `S1` will be executed if the expression `e` is true but `S2` will be executed if `e` is false. `S1` or `S2` or both may be indented suites of statements.

Conditional Statements 6

```
"""
marks2.py
"""

# input mark and convert to number

markString = input("Enter your mark: ")
mark = int(markString)

# output appropriate message

if mark<50 :
  print("You failed")
  if mark>=35 : print("You may resit")
else:
  if mark<70 : print("Well done - you passed")
  else : print("You got a distinction!")
```

Conditional Statements 7

Note that the indentation on the previous slide determines that the first `else` statement matches with the first `if`, so the last two lines of code will be reached if the mark is not less than 50; there is no `else` statement associated with the `if mark>=35` condition.

Note that all `else` statements must match an `if` statement at the same level of indentation; we cannot write an `else` statement not immediately preceded by an `if` statement.

Since we know that the `else` statement is reached only if the mark is not less than 50 it is no longer necessary to check this inside the suite, so we were able to replace `50<=mark<70` by `mark<70`.

Conditional Statements 8

If we use the style of coding on slide 33 and there are several possible ranges to check we would have to use an extra level of indentation for each `if` statement nested inside an `else` statement; this could result in code extending beyond the right margin of a window, making the program difficult to read.

It is better to use a more concise version that replaces an `else` statement whose entire suite is an `if` statement (possibly with another `else` statement) with an `elif` statement.

Conditional Statements 9

Instead of writing

```
if e1 : S1
else :
   if e2 : S2
   [ else S3 ]
```

we may use

```
if e1 : S1
elif e2 : S2
[ else S3 ]
```

(The square brackets are not part of the syntax – they just indicate that the last `else` statement is optional.)

On the next slide we present a version of our marks program making use of this.

Conditional Statements 10

```
"""
marks3.py
"""

# input mark and convert to number

markString = input("Enter your mark: ")
mark = int(markString)

# output appropriate message

if mark<50 :
  print("You failed")
  if mark>=35 : print("You may resit")
elif mark<70 : print("Well done - you passed")
else : print("You got a distinction!")
```

Logical Operators 1

We often need to check whether more than one condition is satisfied; for example to obtain a distinction a student might have to achieve an overall mark of 70 and also obtain at least 60 in the project module. To perform two separate comparisons and combine their results we need to use a ***logical operator***.

The expression to determine whether a student has got a distinction is now `mark>=70 and projMark>=60`. Observe from the precedence table that the `and` operator has lower precedence than the comparison operators so the operations are performed in the correct order.

Logical Operators 2

The value of an expression of the form ***e1*** **`and`** ***e2*** will be **`True`** only if both of the expressions ***e1*** and ***e2*** evaluate to **`True`**. Observe that if ***e1*** evaluates to **`False`** the value cannot be **`True`**, so there is no need to evaluate ***e2***, and indeed it will not be evaluated.

The value of an expression of the form ***e1*** **`or`** ***e2*** will be **`True`** if either or both of the expressions ***e1*** and ***e2*** evaluate to **`True`**. Observe that if ***e1*** evaluates to **`True`** the value must be **`True`**, so again there is no need to evaluate ***e2***.

Logical Operators 3

The remaining logical operator is the `not` operator. This is a unary operator (it has only one operand). An expression `not e` will be `True` if and only if the expression `e` evaluates to `False`.

Note that the `not` operator has higher precedence than both `and` and `or` so `not e1 and e2` would mean `(not e1) and e2`.

Also note that the `and` operator has higher precedence than `or` so `e1 or e2 and e3` would mean `e1 or (e2 and e3)`.

We finally present a version of our marks program that uses the new rules for obtaining a distinction.

Logical Operators 4

```
"""
marks4.py
"""
markString = input("Enter your mark: ")
mark = int(markString)

projString = input("Enter your project mark: ")
projMark = int(projString)

if mark<50 :
  print("You failed")
  if mark>=35 : print("You may resit")
elif mark>=70 and projMark>=60 :
  print ("You got a distinction!")
else : print("Well done - you passed")
```