Python programming

P4

Zain Merchant

Python

Python was conceived by Guido van Rossum in the late 1980s. Python 2.0 was released in 2000 and Python 3.0 in 2008. Python is a multi-paradigm programming language, meaning that it fully supports both object-oriented programming and structured programming. Many other paradigms, including logic programming, are supported using extensions.

The Python programs in this book have been prepared using Python 3 (see www.python.org for a free download) and Python's Integrated DeveLopment Environment (IDLE).

Key characteristics of Python include the following.

- Every statement must be on a separate line.
- Indentation is significant. This is known as the 'off-side rule'.
- Keywords are written in lower case.
- Python is case sensitive: the identifier **Number1** is seen as different from **number1** or **NUMBER1**.
- Everything in Python is an object .
- Code makes extensive use of a concept called 'slicing'.
- Programs are interpreted

You can type a statement into the Python Shell and the Python interpreter will run it immediately.

Programming basics

Declaration of variables

Python handles variables differently to most programming languages. It tags values. This is why Python does not have variable declarations.

In pseudocode, variable declarations are written as:

```
DECLARE <identifier> : <dataType>
```

For example, you may declare the following variables:

```
DECLARE Number1 : INTEGER  // this declares Number1 to store a whole number
DECLARE YourName : STRING // this declares YourName to store a
                          // sequence of characters
DECLARE N1, N2, N3 : INTEGER // declares 3 integer variables
DECLARE Name1, Name2 : STRING // declares 2 string variables
```

Syntax definitions	Python does not have variable declarations	

Code examples	`# Number1 of type Integer`
	`# YourName of type String`
	`# N1, N2, N3 of type integer;`
	`# Name1, Name2 of type string`
		There are no declarations, but comments should be made at the beginning of a module

Declaration and assignment of constants

Sometimes we use a value in a solution that never changes, for example, the value of the mathematical constant pi (π). Instead of using the actual value in program statements, it is good practice and helps readability, if we give a constant value a name and declare it at the beginning of the program.

In pseudocode, constant declarations are written as:

```
CONSTANT <identifier> = <value>
```

For example:

```
CONSTANT Pi = 3.14
```

Syntax definitions	<identifier> = <value>	
Code examples	`PI = 3.14`	Python convention is to write constant identifiers using uppercase only. The values can be changed, although you should treat constants as not changeable.

Assignment variables

Once we have declared a variable, we can assign a value to it

In pseudocode, assignment statements are written as:

<identifier> ← <expression>

For example:

```
A ← 34

B←B+ 1
```

Syntax definitions	<identifier> = <expression>	

Code examples	`A = 34`
	`B = B + 1`
		The assignment operator is =

Arithmetic operators

Assignments don't just give initial values to variables. We also use an assignment when we need to store the result of a calculation.

Operation	**Pseudocode**	**Python**
Addition	+	+
Subtraction	-	-
Multiplication	*	*
Division	/	/
Exponent	^	**
Integer division	DIV	//
Modulus	MOD	%

When more than one operator appears in an expression, the order of evaluation depends on the mathematical **rules of precedence**: parentheses, exponentiation, multiplication, division, addition, subtraction.

Outputting information to the screen

In pseudocode, output statements are written as:

OUTPUT <string> OUTPUT <identifier(s)>

When outputting text and data to the console screen, we can list a mixture of output strings and variable values in the print list.

Syntax definitions	`print(<printlist>)` `print(<printlist>, end ='')`	Print list items are separated by commas (,). To avoid moving onto the next line after the output, use end ='')
Code examples	`print("Hello ", YourName,". Your number is ", Number1)` `print("Hello ", end= '')`	The prompt is provided as a parameter to the input function. Single quotes are also accepted. All input is taken to be a string; if you want to use the input as a number the input string has to be converted using a function
	`print ("Hello {0}. Your number is {1}".format(YourName, Number1))`	
	`A = input("Enter a number: ")`	

Comments

It is good programming practice to add comments to explain code where necessary.

Code examples	`# this is a comment`
	`# this is another comment`	

Data types

Description of data	Pseudocode	Python
Whole signed numbers	INTERGER	int
Signed numbers with a decimal point	REAL	float
A single character	CHAR Use single (') quotation marks to delimit a character	Not available
A sequence of characters (a string)	STRING Use double (") quotation marks to delimit a string.	str (stored as ASCII but Unicode strings are also available) Use single ('), double (") or triple (''' or """) quotation marks to delimit a string.
Logical values: True (represented as 1) and False (represented as 0)	BOOLEAN	bool possible values: True False

In Python, a single character is represented as a string of length 1.

Date has various internal representations but is output in conventional format.

Description of data	Pseudocode	Python
Date value	DATE	Use the datetime class

Boolean expressions

We covered logic statements. These were statements that included a condition. Conditions are also known as Boolean expressions and evaluate to either True or False. True and False are known as Boolean values.

Simple Boolean expressions involve comparison operators. Complex Boolean expressions also involve Boolean operators.

Operation	Pseudocode	Python
Equal	=	==
Not equal	<>	!=
Greater than	>	>
Less than	<	<
Greater than or equal to	>=	>=
Less than or equal to	<=	<=

Operation	Pseudocode	Python
AND (logical conjunction)	AND	and
OR (logical inclusion)	OR	or
NOT (logical negation)	NOT	not

Selection

IF...THEN statements

In pseudocode the IF...THEN construct is written as:

```
IF <Boolean expression>
    THEN
        <statement(s)>
ENDIF
```

Pseudocode example:

```
IF x < 0
    THEN
        OUTPUT "Negative"
ENDIF
```

Syntax definitions	`if<Boolean expression>:`
	`    <statement(s)>`
		Note that the THEN keyword is replaced by a colon (:). Indentation is used to show which statements form part of the conditional statement.

Code examples	`if x < 0:`
	`  print("Negative")`	

IF...THEN...ELSE statements

In pseudocode, the IF...THEN...ELSE construct is written as:

```
IF <Boolean expression>
    THEN
        <statement(s)>
    ELSE
        <statement(s)>
ENDIF
```

Pseudocode example:

```
IF x < 0
    THEN
        OUTPUT "Negative"
    ELSE
        OUTPUT "Positive"
ENDIF
```

Syntax definitions	`if <Boolean expression>:`
	`  <statement(s)>`
	`else:`
	`  <statement(s)>`
		Indentation is used to show which statements form part of the conditional statement;
		the else keyword must line up with the corresponding if keyword.

Code examples	`if x < 0:`
	`    print("Negative")`
	`else:`
	`    print("Positive")`	

Nested IF statements

In pseudocode, the nested IF statement is written as:

```
IF <Boolean expression>
    THEN
        <statement(s)>
    ELSE
        IF <Boolean expression>
            THEN
                <statement(s)>
            ELSE
                <statement(s)>
        ENDIF
ENDIF
```

Pseudocode example:

```
IF x < 0
    THEN
        OUTPUT "Negative"
    ELSE
        IF x = 0
            THEN
                OUTPUT "Zero"
            ELSE
                OUTPUT "Positive"
        ENDIF
ENDIF
```

Syntax definitions	`if <Boolean expression>:`
	`    <statement(s)>`
	`elif <Boolean expression>:`
	`    <statement(s)>`
	`else:`
	`    <statement(s)>`
		Note the keyword elif (an abbreviation of else if). This keyword must line up with the corresponding if. There can be as many elif parts to this construct as required.

Code examples	`if x < 0:`
	`    print("Negative")`
	`elif x == 0:`
	`    print("Zero")`
	`else:`
	`    print("Positive")`	

CASE statements

An alternative selection construct is the CASE statement. Each considered CASE condition can be:

- a single value
- single values separated by commas
- a range.

In pseudocode, the CASE statement is written as:

```
CASE OF <expression>

          <value1> : <statement(s)>
          <value2>,<value3> : <statement(s)>
           <value4> TO <value5> : <statement(s)>
          .
          .
          OTHERWISE <statement(s)>

ENDCASE
```

The value of <expression> determines which statements are executed. There can be as many separate cases as required. The OTHERWISE clause is optional and useful for error trapping.

In pseudocode, an example CASE statement is:

```
CASE OF Grade
        "A" : OUTPUT "Top grade”
        "F", "U" : OUTPUT “Fail"
        "B".."E" : OUTPUT "Pass"
        OTHERWISE OUTPUT "Invalid grade"
ENDCASE
```

Syntax definitions	Python does not have a CASE statement. You need to use nested If statements instead.

Code examples	`if Grade == "A":`
	`  print("Top grade")`
	`elif Grade == "F" or Grade == "U":`
	`  print("Fail")`
	`elif Grade in ("B", "C", "D", "E"):`
	`  print("Pass")`
	`else:`
	`  print("Invalid grade")`

Iteration

Count-controlled (FOR) loops

In pseudocode, a count-controlled loop is written as:

```
FOR <control variable> ← s TO e STEP i // STEP is optional

        <statement(s)>

NEXT <control variable>
```

The control variable starts with value s, increments by value i each time round the loop and finishes when the control variable reaches the value e.

In pseudcode, examples are:

```
FOR x ← 1 TO 5
   OUTPUT x
NEXT x

FOR x = 2 TO 14 STEP 3
   OUTPUT x
NEXT x

FOR x = 5 TO 1 STEP -3
   OUTPUT x
NEXT x
```

Syntax definitions	`for <control variable> in range(s, e, i):`
	`    <statement(s)>`
		The values s, e and i must be of type integer. The loop finishes when the control variable is just below e. The values for s and i can be omitted and they default to 0 and 1, respectively.

Code examples	`for x in range(5):`
	`    print(x, end=' ')`
		The start value of x is 0 and it increases by 1 on each iteration. Output: 0 1 2 3 4

	`for x in range(2, 14, 3):`
	`    print(x, end=' ')`
		Output:2 5 8 11

	`for x in range(5, 1, -1):`
	`    print(x, end=' ')`
		The start value of x is 5 and it decreases by 1 on each iteration. Output:5 4 3 2

	`for x in ["a", "b", "c"]:`
	`    print(x, end='')`
		The control variable takes the value of each of the group elements in turn. Output: abc

Post-condition loops

A post-condition loop, as the name suggests, executes the statements within the loop at least once. When the condition is encountered, it is evaluated. As long as the condition evaluates to False, the statements within the loop are executed again. When the condition evaluates to True, execution will go to the next statement after the loop.

When coding a post-condition loop, you must ensure that there is a statement within the loop that will at some point change the end condition to True. Otherwise the loop will execute forever.

In pseudocode, the post-condition loop is written as:

```
REPEAT

    <statement(s)>

UNTIL <condition>
```

Pseudocode example:

```
REPEAT

    INPUT "Enter Y or N: " Answer

UNTIL Answer = "Y"
```

Post-condition loops are not available in Python. Use a pre-condition loop instead.

Pre-condition loops

Pre-condition loops, as the name suggests, evaluate the condition before the statements within the loop are executed. Pre-condition loops will execute the statements within the loop as long as the condition evaluates to True. When the condition evaluates to False, execution will go to the next statement after the loop. Note that any variable used in the condition must not be undefined when the loop structure is first encountered.

When coding a pre-condition loop, you must ensure that there is a statement within the loop that will at some point change the value of the controlling condition. Otherwise the loop will execute forever.

In pseudocode the pre-condition loop is written as:

```
WHILE <condition> DO

    <statement(s)>

ENDWHILE
```

Pseudocode example,

```
Answer ← “"

WHILE Answer <> "Y" DO

    INPUT "Enter Y or N: " Answer

ENDWHILE
```

Syntax definitions	`while <condition>:`
	`    <statement(s)>`
		Note that statements within the loop must be indented by a set number of spaces. The first statement after the loop must be indented less.

Code examples	`Answer = ''`
	`while Answer != 'Y':`
	`    Answer = input("Enter Y or N: ")`	

Which loop structure to use

If you know how many times around the loop you need to go when the program execution gets to the loop statements, use a count-controlled loop. If the termination of the loop depends on some condition determined by what happens within the loop, then use a conditional loop. A pre-condition loop has the added benefit that the loop may not be entered at all, if the condition does not require it.

Built-in functions

String manipulation functions

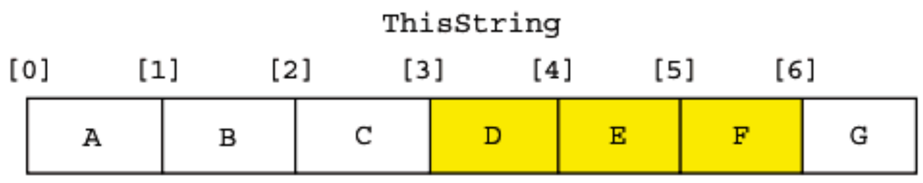
Description	Pseudocode	Python
Access a single character using its position P in a string ThisString	ThisString[P] Counts from 1	ThisString[P] Counts from 0
Returns the character whose ASCII value is i	CHR(i : INTEGER) RETURNS CHAR	chr(i)
Returns the ASCII value of character ch	ASC(ch) RETURNS INTEGER	ord(ch)
Returns the integer value representing the length of string S	LENGTH(S : STRING) RETURNS INTEGER	len(S)
Returns leftmost L characters from S	LEFT(S : STRING, L : INTEGER) RETURNS STRING	S[0:L]
Returns rightmost L characters from S	RIGHT(S: STRING, L : INTEGER) RETURNS STRING	S[-L:]
Returns a string of length L starting at position P from S	MID(S : STRING, P : INTEGER, L : INTEGER) RETURNS STRING	S[P : P + L]
Returns the character value representing the lower case equivalent of Ch	LCASE(Ch : CHAR) RETURNS CHAR	Ch.lower()

Description	Pseudocode	Python
Returns the character value representing the upper case equivalent of Ch	UCASE(Ch : CHAR) RETURNS CHAR	Ch.upper()
Returns a string formed by converting all alphabetic characters of S to upper case	TO _ UPPER(S : STRING) RETURNS STRING	S.upper()
Returns a string formed by converting all alphabetic characters of S to lower case	TO _ LOWER(S : STRING) RETURNS STRING	S.lower()
Concatenate (join) two strings	S1 & S2	s = S1 + S2

Slicing in Python

In Python a subsequence of any sequence type (e.g. lists and strings) can be created using 'slicing'. For example, to get a substring of length L from position P in string S we write S[P : P + L].

Figure shows a representation of ThisString. If we want to return a slice of length 3 starting at position 3, we use ThisString[3 : 6] to give 'DEF'. Position is counted from 0 and the position at the upper bound of the slice is not included in the substring.



If you imagine the numbering of each element to start at the left-hand end, then it is easier to see how the left element (the lower bound) is included, but the right element (the upper bound) is excluded. Table below shows some other useful slices in Python.

Expression	Result	Explanation
ThisString[2:]	CDEFG	If you do not state the upper bound, the slice includes all characters to the end of the string.
ThisString[:2]	AB	If you do not state the lower bound, the slice includes all characters from the beginning of the string.
ThisString[-2:]	FG	A negative lower bound means that it takes the slice starting from the end of the string.
ThisString[:-2]	ABCDE	A negative upper bound means that it terminates the string at that position.

Zain Merchant

Truncating numbers

Instead of rounding, sometimes we just want the whole number part of a real number. This is known as 'truncation'.

Pseudocode	INT(x : REAL) RETURNS INTEGER	Returns the integer part of x.
Python	`int(x)`	If x is a floating-point number, the conversion truncates towards zero.

Converting string to a number

Sometimes a whole number may be held as a string. To use such a number in a calculation, we first need to convert it to an integer. For example, these functions return the integer value 5 from the string "5":

Pseudocode	STR_TO_NUM(x : STRING/CHAR) RETURNS INTEGER/REAL	returns a numeric representation of a string.
Python	`int(x)`	The returned value is an integer number.
	`float(x)`	The returned value is a floating-point number.

Random number generator

Random numbers are often required for simulations. Most programming languages have various random number generators available. As the random numbers are generated through a program, they are referred to as 'pseudo-random' numbers. A selection of the most useful random number generators are shown in the following code examples.

Pseudocode	RAND(x : INTEGER) RETURNS REAL	returns a real number in the range `0` to `x` (**not** inclusive of `x`).

Python	`from random import randint`
	`randint(1, 6)`
		This code produces a random number between 1 and 6

	`import random`
	`random.randint(1, 6)`	

Procedures

We used procedures as a means of giving a group of statements a name. When we want to program a procedure we need to define it before the main program. We call it in the main program when we want the statements in the procedure body to be executed.

In pseudocode, a procedure definition is written as:

```
PROCEDURE <procedureIdentifier> // this is the procedure header

    <statement(s)> // these statements are the procedure body

ENDPROCEDURE
```

This procedure is called using the pseudocode statement:

CALL <procedureIdentifier>

When programming a procedure, note where the definition is written and how the procedure is called from the main program.

Here is an example pseudocode procedure definition:

```
PROCEDURE InputOddNumber
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
ENDPROCEDURE
```

This procedure is called using the CALL statement:

```
CALL InputOddNumber
```

Syntax definitions	`def <identifier>():`
	`    <statement(s)>`

Code example	`def inputOddNumber():`
	`    number = 0`
	`    while number % 2 == 0:`
	`        number = int(input("Enter an odd number: "))`
	`    print("Valid number entered")`
	
	`# main program starts here`
	`inputOddNumber()`

Passing parameters to procedures

If a parameter is passed **by value**, at call time the argument can be an actual value. If the argument is a variable, then a copy of the current value of the variable is passed into the subroutine. The value of the variable in the calling program is not affected by what happens in the subroutine.

For procedures, a parameter can be passed **by reference**. At call time, the argument must be a variable. A pointer to the memory location (the address) of that variable is passed into the procedure. Any changes that are applied to the variable's contents will be effective outside the procedure in the calling program/module.

Note that neither of these methods of parameter passing applies to Python. In Python, the method is called pass by object reference. This is basically an object-oriented way of passing parameters. The important point is to understand how to program in Python to get the desired effect.

The full **procedure header** is written in pseudocode, in a very similar fashion to that for function headers, as:

```
PROCEDURE <ProcedureIdentifier> (<parameterList>)
```

The parameter list needs more information for a procedure definition. In pseudocode, a parameter in the list is represented in one of the following formats:

```
BYREF <identifier1> : <dataType>
BYVALUE <identifier2> : <dataType>
```

Passing parameters by value

```
PROCEDURE OutputSymbols(BYVALUE NumberOfSymbols : INTEGER, Symbol :
CHAR)
    DECLARE Count : INTEGER
    FOR Count ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT NewLine
ENDPROCEDURE
```

In Python, all parameters behave like local variables and their effect is as though they are passed by value.

Code example	see below

```python
def outputSymbols( numberOfSymbols, symbol):
    for count in range(numberOfSymbols):
        print(symbol, end='')
    print()


# main program starts here
outputSymbols(5, '*')
```

Passing parameters by reference

When parameters are passed by reference, when the values inside the subroutine change, this affects the values of the variables in the calling program.

Consider the pseudocode procedure AdjustValuesForNextRow below.

```
PROCEDURE AdjustValuesForNextRow(BYREF Spaces : INTEGER, Symbols : INTEGER)
Spaces ← Spaces - 1
Symbols ← Symbols + 2
ENDPROCEDURE
```

The pseudocode statement to call the procedure is:

```
CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
```

The values of the parameters Spaces and Symbols are changed within the procedure when this is called. The variables NumberOfSpaces and NumberOfSymbols in the program code after the call will store the updated values that were passed back from the procedure.

Python does not have a facility to pass parameters by reference. Instead the subroutine behaves as a function and returns multiple values. Note the order of the variables as they receive these values in the main part of the program.

Code example

```
def adjustValuesForNextRow(spaces, symbols):
    spaces = spaces - 1
    symbols = symbols + 2
    return spaces, symbols


# main program starts here
numberOfSpaces = int(input())
numberOfSymbols = int(input())
numberOfSpaces, numberOfSymbols =
adjustValuesForNextRow(numberOfSpaces, numberOfSymbols)
print(numberOfSpaces)
print(numberOfSymbols)
```

This way of treating a multiple of values as a unit is called a 'tuple'. You can find out more by reading the Python help files.

Functions

You can write your own functions. Any function you have written can be used in another program if you build up your own module library.

A function is used as part of an expression. When program execution gets to the statement that includes a function call as part of the expression, the function is executed. The **return value** from this function call is then used in the expression.

When writing your own function, ensure you always return a value as part of the statements that make up the function (the function body). You can have more than one RETURN statement if there are different paths through the function body.

In pseudocode, a function definition is written as:

```
FUNCTION <functionIdentifier> RETURNS <dataType> // function header
    <statement(s)> // function body
    RETURN <value>
ENDFUNCTION
```

We can write the example procedure from previous page as a function. In pseudocode, this is:

```
FUNCTION InputOddNumber RETURNS INTEGER
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered”
    RETURN Number
ENDFUNCTION
```

When programming a function, the definition is written in the same place as a procedure. The function is called from within an expression in the main program, or in a procedure. In Python different terminology is used for subroutines; procedures are called void function and functions are called fruitful function.

Void means ‘nothing’. Python uses this term to show that their procedure-type subroutine does not return a value. Python refers to both types of subroutines as functions. The fruitful function returns one or more values.

Syntax definitions

```
def <functionIdentifier>():
    <statement(s)>
    return <value>
```

Code example

```
def inputOddNumber():
    number = 0
    while number % 2 == 0:
        number = int(input("Enter an odd number: "))
    return number


# main program starts here
newNumber = inputOddNumber()
```

The variable Number in code above is not accessible in the main program. Python’s variables are local unless declared to be global.

A global variable is available in any part of the program code. It is good programming practice to declare a variable that is only used within a subroutine as a local variable.

In Python, every variable is local, unless it is overridden with a global declaration.

Passing parameters to function

he function header is written in pseudocode as:
FUNCTION <functionIdentifier> (<parameterList>) RETURNS <dataType>

where <parameterList> is a list of identifiers and their data types, separated by commas.
Here is an example pseudocode function definition that uses parameters:

```
FUNCTION SumRange(FirstValue : INTEGER, LastValue : INTEGER) RETURNS
INTEGER
    DECLARE Sum, ThisValue : INTEGER
    Sum ← 0
    FOR ThisValue ← FirstValue TO LastValue
        Sum ← Sum + ThisValue
    NEXT ThisValue
    RETURN Sum
ENDFUNCTION
```

Code example	`def sumRange(firstValue, lastValue):`
	`    sum = 0`
	`    for thisValue in range(firstValue, lastValue + 1):`
	`        sum = sum = thisValue`
	`    return sum`
	
	`# main program starts here`
	`newNumber = sumRange(1, 5)`
	`print(newNumber)`

Arrays

Creating 1D arrays

In Python number array elements from 0 (the lower bound). In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Pseudocode example:

```
DECLARE List1 : ARRAY[1:3] OF STRING // 3 elements in this list
DECLARE List2 : ARRAY[0:5] OF INTEGER // 6 elements in this list
DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
DECLARE List4 : ARRAY[0:25] OF STRING // 26 elements in this list
```

Syntax definitions	In Python, there are no arrays. The equivalent data structure is called a list. A list is an ordered sequence of items that do not have to be of the same data type.	

Code examples	`list1 = []`
	`list1.append("Babar Azam")`
	`list1.append("Virat Kohli")`
	`list1.append("Viv Richards")`
		As there are no declarations, the only way to generate a list is to initialise one. You can append elements to an existing list.

	`List2 = [0,0,0,0,0]`	You can enclose the elements in `[ ]`.
	`List3 = [0 for i in range(100)]`	You can use a loop.

Zain Merchant

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
<arrayIdentifier>[x]
```

Pseudocode example:

```
NList[25] = 0 // set 25th element to zero
AList[3] = "D" // set 3rd element to letter D
```

Code example	`NList[24] = 0`
	`AList[3] = "D"`

In Python, you can print the whole contents of a list using print(List).

Creating 2D arrays

In pseudocode, a 2D array declaration is written as:

DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>, <lBound2>:<uBound2>] OF <dataType>

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

DECLARE Board : ARRAY[1:6, 1:7] OF INTEGER

Syntax definitions	In Python, there are no arrays. The equivalent data structure is a list of lists.	

Code examples	
	```
	Board = [[0, 0, 0, 0, 0, 0, 0],
	          [0, 0, 0, 0, 0, 0, 0],
	          [0, 0, 0, 0, 0, 0, 0],
	          [0, 0, 0, 0, 0, 0, 0],
	          [0, 0, 0, 0, 0, 0, 0],
	          [0, 0, 0, 0, 0, 0, 0]]

	Board = [[0 for i in range(7)]for j in
	range(6)]
	```
		2D lists can be initialised in a similar way to 1D lists. Remember that elements are numbered from 0. These are alternative ways of initialising a 6 × 7 list. The rows are numbered 0 to 5 and the columns 0 to 6. The upper value of the range is not included.

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

```
<arrayIdentifier>[x, y]
```

Pseudocode example:

```
Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

The following code examples demonstrate how to access elements

Code examples	`Board[2][3] = 0`	Elements are numbered from 0 in Python, so [3] gives access to the fourth element.

Text Files

Writing to a text file

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE // open the file for writing
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename> // close file
```

The following code examples demonstrate how to open, write to and close a file called SampleFile.TXT. If the file already exists, it is overwritten as soon as the file handle is assigned by the 'open file' command.

Code examples	`FileHandle = open("SampleFile.TXT", "w")`
	`FileHandle.write("Line of text to store\n")`
	`FileHandle.close()`
		You specify the filename and mode ('w' for write) when you call the open function. The line of text to be written to the file must contain the newline character "\n" to move to the next line of the text file.

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ // open file for reading
READFILE <filename>, <stringVariable> // read a line of text from the file
CLOSEFILE <filename> // close file
```

The following code examples demonstrate how to open, read from and close a file called SampleFile.TXT

Code examples	`FileHandle = open("SampleFile.TXT", "r")`
	`LineOfText = FileHandle.readline()`
	`FileHandle.close ()`
		You specify the filename and mode ('r' for read) when you call the open function.

Appending to a text file

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename> // close file
```

The following code examples demonstrate how to open, append to and close a file called SampleFile.TXT.

Code examples	`FileHandle = open("SampleFile.TXT", "a")`
	`FileHandle.write(LineOfText)`
	`FileHandle.close()`
		You specify the filename and mode ('a' for append) when you call the open function.

The end-of-file (EOF) marker

The following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
        READFILE "Test.txt", TextString
        OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

The following code examples demonstrate how to read and then output the contents of a file in each of the three languages.

Code examples	`FileHandle = open("Test.txt", "r")`
	`LineOfText = FileHandle.readline()`
	`while len(LineOfText) > 0:`
	`    LineOfText = FileHandle.readline()`
	`    print(LineOfText)`
	`FileHandle.close()`
		There is no explicit EOF function. However, when a line of text has been read that only consists of the end-of-file marker, the line of text is of length 0.

Object-oriented programming

Designing classes and objects

When designing a class, we need to think about the attributes we want to store. We also need to think about the methods we need to access the data and assign values to the data of an object. A data type is a blueprint when declaring a variable of that data type. A class definition is a blueprint when declaring an object of that class. Creating a new object is known as 'instantiation'.

Any data that is held about an object must be accessible, otherwise there is no point in storing it. We therefore need methods to access each one of these attributes. These methods are usually referred to as **getters**. They get an attribute of the object.

When we first set up an object of a particular class, we use a constructor. A **constructor** instantiates the object and assigns initial values to the attributes.

Any attributes that might be updated after instantiation will need subroutines to update their values. These are referred to as **setters**. Some attributes get set only at instantiation. These don't need setters. This makes an object more robust, because you cannot change attributes that were not designed to be changed.

Python allows properties to be declared. A **property** combines the attribute with its associated setter and/or getter

Declaring a class

Attributes should always be declared as ‘Private’. This means they can only be accessed through the class methods. So that the methods can be called from the main program, they have to be declared as ‘Public’.

Code examples

```
class car:
   def __init__(self, n, e):
      self.__VehicleID = n
      self.__Registration = ""
      self.__Date0fRegistration = None
      self.__EngineSize = e
      self.__PurchasePrice = 0.00
   def SetPurchasePrice(self, p):
      self.__PurchasePrice = p
   def SetRegistration(self, r):
      self.__Registration = r
   def SetDateOfRegistration(self, d):
      self.__Date0fRegistration = d
   def GetVehicleIDID(self):
      return(self.__VehicleID)
   def GetRegistration(self):
      return(self.__Registration)
   def GetDate0fRegistration(self):
      return(self.__Date0fRegistration)
   def GetEngineSize(self):
      return(self.__EngineSize)
   def GetPurchasePrice(self):
      return(self.__PurchasePrice)
```

Two underscore characters are required before and after **init** to define the constructor.

Self is the first parameter in the parameter list for every method.

Two underscore characters before an attribute name signify it is private

Instantiating class

To use an object of a class type in a program the object must first be instantiated. This means the memory space must be reserved to store the attributes.

The following code instantiates an object ThisCar of class Car.

Code example

```
ThisCar = Car("ABC1234", 2500)
```

Using a method

To call a method in program code, the object identifier is followed by the method identifier and the parameter list.

The following code sets the purchase price for an object ThisCar of class Car.

Code example

```
ThisCar.SetPurchasePrice(12000)
ThisCar.PurchasePrice = 12000 # using properties
```

To call a method in program code, the object identifier is followed by the method identifier and the parameter list.

The following code sets the purchase price for an object ThisCar of class Car.

Code example

```
print(ThisCar.GetVehicleID())
print(ThisCar.VehicleID) # using properties
```

Inheritance

The advantage of OOP is that we can design a class (a base class or a superclass) and then derive further classes (subclasses) from this base class. This means that we write the code for the base class only once and the subclasses make use of the attributes and methods of the base class, as well as having their own attributes and methods. This is known as **inheritance**.

Declaring a base class and derived class (subclass)

```
import datetime
class LibraryItem:
    def __init__(self, t, a, i):    # initialiser method
        self.__Title = t
        self.__Author__Artist = a
        self.__ItemID = i
        self.__OnLoan = False
        self.__DueDate = datetime.date.today()
    def GetTitle(self):
        return (self.__Title)
# other Get methods go here
    def Borrowing(self):
        self.__OnLoan = True
        self.__DueDate = self.__DueDate +
datetime.timedelta(weeks=3)
    def Returning(self):
        self.__OnLoan - False
    def PrintDetails(self):
        print(self.__Title, ',', self.__Author__Artist, ',')
        print(self.__ItemID, ',', self.__OnLoan, ',',
self.__DueDate)

class Book(LibraryItem):
    def __init__(self, t, a, i):
        LibraryItem.__init__(self, t, a, i)
        self.__IsRequested = False
        self.__RequestedBy = 0
    def GetIsRequested(self):
        return (self.__IsRequested)
    def SetLiRequested(self):
        self.__IsRequested = True

class CD(LibraryItem):
    def __init__(self, t, a, i):
        LibraryItem.__init__(self, t, a, i)
        self.__Genre = ""
    def GetGenre(self):
        return (self.__Genre)
    def SetGenre(self, g):
        self.__Genre = g
```

Instantiating a subclass

Creating an object of a subclass is done in the same way as with any class

Code example	`ThisBook = Book(Title, Author, ItemID)`
	`ThisCD = CD(Title, Artist, ItemID)`

Using a method

Using an object created from a subclass is exactly the same as an object created from any class

Code example	`ThisCD.Borrowing()`
	`ThisCD.PrintDetails()`

Polymorphism

The constructor method of the base class is redefined in the subclasses. The constructor for the Book class calls the constructor of the LibraryItem class and also initialises the IsRequested attribute. The constructor for the CD class calls the constructor of the LibraryItem class and also initialises the Genre attribute.

The PrintDetails method is currently only defined in the base class. This means we can only get information on the attributes that are part of the base class. To include the additional attributes from the subclass, we need to declare the method again. Although the method in the subclass will have the same identifier as in the base class, the method will actually behave differently. This is known as **polymorphism**.

The way the programming languages re-define a method varies.

The code shown here includes a call to the base class method with the same name. You can completely re-write the method if required.

```
class Book(LibraryItem):
    def __init__(self, t, a, i):
        LibraryItem.__init__(self, t, a, i)
        self.__IsRequested = False
        self.__RequestedBy = 0
    def GetIsRequested(self):
        return (self.__IsRequested)
    def SetLiRequested(self):
        self.__IsRequested = True
    # define the print details method for Book
    def PrintDetails(self):
        print("Book Details")
        LibraryItem.PrintDetails()
        print(self.__IsRequested)
```

Garbage collection

Zain Merchant

When objects are created they occupy memory. When they are no longer needed, they should be made to release that memory, so it can be re-used. If objects do not let go of memory, we eventually end up with no free memory when we try and run a program. This is known as ‘memory leakage’.

Memory management involves a private heap containing all Python objects and data structures. The management of the Python heap is performed by the interpreter itself. The programmer does not need to do any housekeeping.

Containment (aggregation)

The base class is the most general class, subclasses derived from this base class are more specialised.

We have other kinds of relationships between classes. **Containment** means that one class contains other classes. For example, a car is made up of different parts and each part will be an object based on a class. The wheels are objects of a different class to the engine object. The engine is also made up of different parts. Together, all these parts make up one big object.

Zain Merchant

```
class Lesson:
    def __init__(self, t, d, r):
        self.LessonTitle = t
        self.DurationMinutes = d
        self.RequiresLab = r
    def OutputLessonDetails(self):
        print(self.LessonTitle)
        print("Total time in minutes:", self.DurationMinutes)


class Assessment:
    def __init__(self, t, m):
        self.AssessmentTitle = t
        self.MaxMarks = m
    def OutputAssessmentDetails(self):
        print(self.AssessmentTitle)
        print("Maximum marks:", self.MaxMarks)

class Course:
    def __init__(self, t, m):
        self.CourseTitle = t
        self.MaxStudents = m
        self.NumberOfLessons = 0
        self.CourseLesson = []
        self.CourseAssessment = Assessment
    def AddLesson(self, t, d, r):
        self.NumberOfLessons = self.NumberOfLessons + 1
        self.CourseLesson.append(Lesson(t, d, r))
    def AddAssessment(self, t, m):
        self.CourseAssessment = Assessment(t, m)
    def OutputCourseDetails(self):
        print(self.CourseTitle)
        print("Maximum number:", self. MaxStudents)
        for i in range(self.NumberOfLessons):
            print(self.CourseLesson[i].OutputLessonDetails())

MyCourse = Course("Computing", 10) # sets up a new course
MyCourse.AddAssessment("Programming", 100) # adds an assignment #
add 3 lessons
MyCourse.AddLesson("Problem Solving", 60, False)
MyCourse.AddLesson("Programming", 120, True)
MyCourse.AddLesson("Theory", 60, False)
# check it all works
MyCourse.OutputCourseDetails()
```

Records

Sometimes variables of different data types are a logical group, such as data about a person (name, date of birth, height, number of siblings, whether they are a full-time student).

Name is a STRING; date of birth is a DATE; height is a REAL; number of siblings is an INTEGER; whether they are a full-time student is a BOOLEAN.

We can declare a record type to suit our purposes. The record type is known as a user-defined type, because the programmer can decide which variables (fields) to include as a record.

In pseudocode a record type is declared as:

```
TYPE <TypeIdentifier>
    DECLARE <field identifier> : <data type>
    .
    .
ENDTYPE
```

We can now declare a variable of this record type:

```
DECLARE <variable identifier> : <record type>
```

And then access an individual field using the dot notation:

```
<variable identifier>.<field identifier>
```

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure:

```
TYPE CarRecord
DECLARE VehicleID :  STRING // unique identifier and record key
DECLARE Registration :  STRING
DECLARE DateOfRegistration :  DATE
DECLARE EngineSize :  INTEGER
DECLARE PurchasePrice :  CURRENCY
ENDTYPE
```

To declare a variable of that type we write:

```
DECLARE ThisCar : CarRecord
```

Note that we can declare arrays of records. If we want to store the details of 100 cars, we declare an array of type CarRecord

```
DECLARE Car : ARRAY[1:100] OF CarRecord
```

Code examples

```python
class CarRecord: # declaring a class without other methods
   def __init__(self):    # constructor
      self.VehicleID = ""
      self.Registration = ""
      self.DateOfRegistration = None
      self.EngineSize = 0
      self.PurchasePrice = 0.00
ThisCar = CarRecord() # instantiates a car record
ThisCar.EngineSize = 2500 # assign a value to a field
Car = [CarRecord() for i in range(100)] # make a list of 100 car records
Car[1].EngineSize = 2500 # assign value to a field of the 2nd car in list
```

Python does not have a record type. However, we can use a class definition with only a constructor to assign initial values.

Exception handling

Run-time errors can occur for many reasons. Some examples are division by zero, invalid array index or trying to open a non-existent file. Run-time errors are called ‘exceptions’. They can be handled (resolved) with an error subroutine (known as an ‘exception handler’), rather than let the program crash.

Using pseudocode, the error-handling structure is:

```
TRY
        <statementsA>
EXCEPT
        <statementsB>
ENDTRY
```

Any run-time error that occurs during the execution of <statementsA> is caught and handled by executing <statementsB>. There can be more than one EXCEPT block, each handling a different type of exception. Sometimes a FINALLY block follows the exception handlers. The statements in this block will be executed regardless of whether there was an exception or not.

Python distinguishes between different types of exception, such as:

- **IOError**: for example, a file cannot be opened
- **ImportError**: Python cannot find the module
- **ValueError**: an argument has an inappropriate value
- **KeyboardInterrupt**: the user presses Ctrl+C or Ctrl+Del
- **EOFError**: a file-read meets an end-of-file condition
- **ZeroDivisionError**: a division by zero has been attempted

Code example

```
NumberString = input("Enter an integer: ")
try:
   n = int(NumberString)
   print(n)
except:
   print("This was not an integer")
```

```
a = int(input("Enter number for variable a: "))
b = int(input("Enter number for variable b: "))
try:
   c = ((a + b) / (a - b))
except ZeroDivisionError:
   print("a/b result in 0")
else:
   print(c)
```

```
a = int(input("Enter a number: "))
try:
    k = 5 // a  # raises divide by zero exception.
    print(k)
# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")
finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

File processing

Text files only allow us to write strings in a serial or sequential manner. We can append strings to the end of the file.

When we want to store records in a file, we create a binary file. We can store records serially or sequentially. We can also store records using direct access to a **random file.**

Sequential file processing

If we have an array of records, we may want to store the records on disk before the program quits, so that we don't lose the data. We can open a binary file and write one record after another to the file. We can then read the records back into the array when the program is run again.

```
import pickle # this library is required to create binary files
Car = [CarRecord() for i in range(100)]
CarFile = open('CarFile.DAT', 'wb') # open file for binary write
for i in range(100):   # loop for each array element
   pickle.dump(Car[i], CarFile) # write a whole record to the binary
file
CarFile.close()   # close file
```

```
import pickle
CarFile = open('CarFile.DAT', 'rb')  # open file for binary read
Car = []  # start with empty list
moreLines = True
while moreLines:  # check for end of file
   try:
      Car.append(pickle.load(CarFile))  # append record from file to
end of list
   except EOFError:
      print("File ended")
      moreLines = False
CarFile.close()
```

Random-access file processing

Instead of storing records in an array, we may want to store each record in a binary file as the record is created. We can then update the record in situ (read it, change it and save it back in the same place). Note that this only works for fixed-length records. We can use a hashing function to calculate an address from the record key and store the record at the calculated address in the file. Just as with a hash table, collisions may occur and records need to be stored in the next free record space.

```
import pickle  # this library is required to create binary files
ThisCar = CarRecord()
CarFile = open('CarFile.DAT','rb+')  # open file for binary read and
write
Address = hash(ThisCar.VehicleID)
CarFile.seek(Address)
pickle.dump(ThisCar, CarFile)  # write a whole record to the binary
file
CarFile.close()  # close file
```

```
# to find a record from a given VehicleID:
CarFile = open('CarFile.DAT','rb')  # open file for binary read
vID = input("Enter vehicle id to search in the file")
Address = hash(vID)
CarFile.seek(Address)
ThisCar = pickle.load(CarFile)  # load record from file
CarFile.close()
```