Fundamentals of Computer Programming

Lecture slides - I/O and File Handling

Intro

- This lesson covers
  - Overview of I/O
  - Approaches to formatting output
    - F-strings
    - str.format()
    - %-style formatting
  - File Handling
    - File types
    - Reading and Writing to Files
    - Controlling access position

Intro

- All computer programs consist of: **input -> processing -> output**
- Up until this point we have limited the input and output (I/O) to using the `input()` and `print()` functions Fine for simple programs
- Although this is suitable for development of basic programs it is not suitable for larger more professional type systems
- On many Graphical User Interface (GUI) based systems input and output is handled by what was traditionally called a Windows Icons Menus and Pointers (WIMP) graphical environment

typical name

- I/O however is not just limited to interaction with humans (via keyboards and displays), I/O is often includes other sources such as files, networks etc.

Non human things

Formatting output

Various methods in Python to improve the formatting when generating an output.

- Prior to looking at other I/O it is worth revisiting basic output, and specifically look at improving the format
- Python provides several mechanisms for improving the “layout” or “formatting” of generated textual output
- These mechanisms include -
  - Formatted String Literals
  - The `str.format()` method
  - Manual Formatting
  - Old style `printf()` `%`-style formatting

Formatted String Literals

- Formatted String Literals (sometimes call *f-strings*) are built directly into the Python language
- These provide the *modern* way of formatting, and only appeared since version 3.6 of Python

Designed to make formatting of strings easy

- An *f-string* allows information to be added to a regular string so that when displayed formatting is applied
- f-strings are identified simply by prefixing a string literal with a '`f`' or '`F`' character, then embedding *expressions* into the string between braces `{ }`

if a string has those it has a solution to f-strings

- An optional *format-specifier* can be included, which dictates the actual formatting

f-string Examples

- Any Python expression can appear between the braces, and these often refer to variables, e.g.

```
print(f"Your name is {name}, and your address is {addr}")
```

referring to a variable in this example

refers to variables in this example

stops it being immediately printed as it is. The f sign indicates special information that will be replaced in the string before it is displayed

- More complex expressions can be included, such as arithmetic expressions and calls to functions, e.g.

```
print(f"A circle with radius {r} has an area of {math.pi * r * r}")
```

this is just a variable again

This is an expression

- Actual formatting is applied via the use of *format-specifiers*, which are added using a ‘:’, e.g. print to two decimal places only

```
print(f"A circle with radius {r} has an area of {math.pi * r * r:.2f}")
```

The colon symbols formatting as well as the replacement value

Format-Specifiers

- Format-specifiers are powerful but rather cryptic to use, the official specification defines these as follows -

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

- Although this may look daunting, there are typical scenarios that re-occur when using these to specify a desired output format
- We have already seen a very common use, which is to determine the number of digits that should be displayed following a decimal point, specified using `.precision`, e.g.

```
print(f"The value of π (pi) to 10 decimal places is {math.pi :.10f}")
print(f"The average cost was £{total/items :.2f}")
```

Format-Specifiers: Common Usage

The formatting is useful for both file output and print outputs.

- Another very common use is to specify the column *width* and *alignment*
- A *minimum* width is specified by providing a number following the ‘ : ’, e.g.

```
for name, grade in student_marks:
    print(f"{name:8} ---> {grade:4d} ")
```

↳ indicates the column width.

↳ indicates decimal number

```
John     --->    7
Eric     --->   98
Terry    ---> 100
```

Ensures things are aligned

- By default text is aligned to the left of the available column width, and numbers are aligned to the right
- *Alignment* can be changed by prefixing with < (align left), > (align right), ^ (align centre), or = (padding after the sign, for numerical values only)

↳ Typically maths symbols but not when used in an f-string

Format-Specifiers: Examples

```
# Specifies a 0 'fill' character and displays as hexadecimal
print(f"The decimal value {val:6d} is {val:0>8X} in hex ")
The decimal value    255 is 000000FF in hex

# Specifies a 0 'fill' character and displays as binary
print(f"The decimal value {val:3d} is {val:0>8b} in binary")
The decimal value  23 is 00010111 in binary

# Shows use of centre alignment, and specifies 2 digits after decimal
point
print(f"The cost of '{item:^11}' was £{cost:8.2f} ")
The cost of '   bread   ' was £   12.36

# Specifies a - 'fill' character, centre aligned with 20 min column width
print(f"high score is {score:-^20}")
high score is --------125---------
```

Using str.format()

Works similarly to ant String but using methods

- This approach appeared in Python 2.6, and is based on calling a *method* to handle formatting
- A string is provided that contains *replacement fields*, identified using `{}`
- These fields are then replaced by objects passed into the `format()` method call, e.g.

```
print("Your name is {}, and your address is {}".format(name, addr))
```

- A number within the brackets can be used to refer to the position of the passed object, e.g.

```
print("Your name is {1}, and your address is {0}".format(addr, name))
```

It is important to remember it is NOT built into the language - it is a function being called.

str.format() examples

- A *keyword* can also be specified within the brackets to refer to the passed object, e.g.

```
print("Your name is {id}, and address is {0}".format(addr,  id = name))
```

- *Format-specifiers* can also be included within the *replacement fields*, using the same format as f-strings, e.g.

```
print("The value of π (pi) to 3 decimal places is
{:.3f}".format(math.pi))
print("The cost of '{:^11}' was £{:8.2f}".format(item,cost))
```

- Hence, it is possible to use `str.format()` as an alternative to *f-strings*, but in most cases *f-strings* are the more elegant solution

Manual formatting

- As well as `format()` other string *methods* exist that allow a certain amount of manual formatting to take place
- Alignment can be achieved using `rjust()`, `ljust()` and `center()` to right-justify, left-justify or centre output, e.g.

```
print(name.ljust(8),"--->",str(grade).rjust(4))
John     --->    7
```

- A method called `zfill()` also allows padding of numeric values, e.g.

```
print("3.2".zfill(6))
0003.2
```

- In most cases however it is easier to use *f-strings* or `format()`

Allows pretty much the same functions as others but with little more work

Older style %-formatting

The very original method.

- Prior to *f-strings* and the `format()` method, Python used a style loosely based on the approach taken by the C programming language
- Although out of date some programs still use this style, although it is no longer advised and *f-strings* or `format()` are a much better
- Hence you should at least be able to recognise this, even if you don't use it yourself
- This approach uses a `%` operator, applied to a string and *values*, e.g.

```
print("The value of π (pi) to 2 decimal places is %.2f"  % math.pi)
```

- This approach does work and is flexible, but can be very difficult to read when many values or complex formatting is involved

File Handling

- A very common requirement is the ability to read and write information to and from files, this is known as **file handling**
- All computer files are stored as a list of numbers (bytes), and it is the *file format* that determines what the list of bytes actually represents

Files are a list of numbers. File format is what those numbers mean

- At a very basic level files can be thought of as containing *text* or *binary* information (in reality all files are just binary)
- When a file is treat as *text*, additional processing is performed to take account of this, e.g. recognition of special end-of-line codes
- Python provides functions to allow creation, reading, writing and deletion both text and binary files

Opening a file

- In order to access the contents of a file it first must be opened, in Python this is achieved using the `open()` function
- When calling open() we pass a *filename* parameter, on success a **file object** is returned, e.g.

```
# open an existing file called 'info.txt' for reading
f = open("info.txt")
```

- The returned object allows us to perform further operations on the file, such as reading and writing
- Once we are done processing a file it must be closed, by calling the `close()` method, e.g.

```
f.close()    # close the file 'f'
```

File opening modes

- By default the `open()` function opens the file for reading, and assumes it is a text file
- A second parameter can be passed that indicates the *mode* of operation, such as reading (`r`), writing (`w`), appending (`a`) or reading and writing (`r+`)

```
f1 = open("nextfile.txt", "w")   # open new file for writing
f2 = open("file123.txt", "a")    # open existing file for appending
f3 = open("fileABC.txt", "r+")   # open file for for reading and
writing
```

Anything written will go to the end of the file

- This mode parameter can have a '`b`' appended to indicate the file should be processed as a binary file rather than a text file, e.g.

```
f4 = open("image.png", "rb")     # open file for reading in binary mode
```

Not done with text files
Done if a file is not text.

Reading file contents

- Once a file has been opened the contents can be read, e.g.

```
file_contents = f.read()  # read entire contents of 'f' contents
```

This command writes the contents of the text file to a variable

- When working with text files it is common to read a line at a time, e.g.

```
line1 = f.readline()      # read first line (as a string)
line2 = f.readline()      # read second line (as a string)

line_list = f.readlines() # read all lines (as a list)
```

Reads each line down as an element in a list.

- Rather than read each line in-turn, this is normally done using a loop (which supports direct *iteration* of a file object, e.g.

```
for line in f:
    print(line)
```

Writing file contents

- If a file has been opened in *write*, *append* or *read/write* mode, then we can write to the file, e.g.

```
f1.write("this text will be written to the file")
```

- When writing non-string values, we need to convert them first, e.g.

```
age = 50                          # 'age' is an integer type
f1.write(str(age))
```

- We must do the same thing with more complex collection types, such as *lists*, *tuples*, *sets* etc.

```
details =   ("mark", "cae118") # 'details' is a tuple type
f1.write(str(details))
```

File position

- The file object maintains an integer value that refers to the current *position* within the file, this is used by methods such as `read()` and `write()`, to determine from where to next access data
- We can find the current position using the `tell()` method, e.g.

```
cur_pos = f.tell()    # find out current file position
```

- We can also change the current position using the `seek()` method, e.g.

```
f.seek(0)             # move position back to the start of the file
```

- The `fseek()` method takes another parameter, allowing movement to be relative to the start (`0`), current (`1`), or end (`2`) position of the file, e.g.

```
f.seek(0,2)           # move position to the very end of the file
```

Handling Exceptions

- File handling often involves the need to handle **exceptions**, which is a mechanism for dealing with *run-time* errors
- This can be done using something called a `try...except...finally` construct
- If this approach is taken, a `finally` block should be used to close the file, since file closing should always be done
- An alternative construct exists however which effectively does the file closing for us, whether an *exception* occurs or not, e.g.

```
with open("some_file") as f:
     lines = f.readlines()
     do_something(lines)

# file is always closed by this point, whether an exception occurred or not
```

Summary

- All programs deal with input and output, called **I/O**
- Output of text often requires some sort of formatting to be performed
- Python supports '**f-strings**', the **str.format()** method, and '**% formatting**'
- **File handling** allows both reading and writing of file content
- A file must be **opened** before it can be accessed
- Files can be treat as containing either **text** or **binary**
- The '`with`' statement is often used to encompass *file handling* code

Useful Resources

- A guide to *f-strings*, with a comparison to the older techniques - https://realpython.com/python-f-strings/
- The Format Specification Mini-Language - https://docs.python.org/3/library/string.html#formatspec
- File Object Methods https://www.tutorialspoint.com/python/file_methods.htm