Introduction to Programming

Lab Worksheet

Week 8

Prior to attempting this lab tutorial ensure you have read the related lecture notes and/or viewed the lecture videos on MyBeckett. Once you have completed this lab you can attempt the associated exercises.

You can download this file in Word format if you want to make notes in it.

You can complete this work using the Python interpreter in interactive mode. This could be inside an IDE, or just a command prompt.

Topics covered:

- Basics of I/O
- String Formatting
- File Handling
- Reading and Writing Files
- Random File Access

For more information about the module delivery, assessment and feedback please refer to the module within the MyBeckett portal.

Basics of I/O

All programs take *input*, do some *processing*, then generate *output*. When running a command line program output can be produced using the `print()` function and the user can be prompted to enter values using the `input()` function, as we have seen multiple times. Many desktop based applications also take input from Graphical User Interface (GUI) components, such as text entry fields, menu options, or buttons etc. Input and output however is not limited to interaction with users, it can also occur with other sources such as files and network systems.

When generating output, either directly to the console or to the contents of a file, it is often necessary to be able to *format* the output. This may be done so the output looks presentable, or because the system that reads the output (which could be another program) requires a very specific layout to be followed. Typical formatting may involve placing text into columns, displaying numbers with a specific number of decimal places, etc.

The Python language provides several mechanisms for improving the “formatting” or “layout” of generated textual output. Hence, there is more than one way to perform formatting, and the techniques have evolved slightly over the lifetime of the language.

Formatting with ‘f-strings’

The most recent approach to formatting is called *Formatted String Literals*, or f-strings for short. This has only been present within the Python language since version 3.6. An f-string allows formatting information to be added into a regular string; it is integrated directly into the language syntax.

An f-string is identified by prefixing a string literal with a `f` or `F` character, then embedding expressions into the string between braces `{ }`. Any Python expression can appear between the braces, and these often refer to variables. A simple example looks like this:

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

Notice the `f` that appears directly before the string. When this executes, the braces and expressions will be replaced by their result. So if the value of the variable `name` was “`Donald`”, and the value of `addr` was “`1600 Pennsylvania Ave.`”, then the above example would display the following:

```
Your name is Donald, and your address is 1600 Pennsylvania Ave.
```

Since any expression can appear between the braces, it is possible to include more complex calculations:

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

In this example, the expression to calculate the area is evaluated prior to being displayed. So if `r` was set to `50` then the following would be displayed -

```
A circle with radius 50 has an area of 7853.981633974483
```

TASK: Write some code that uses an f-string to calculate then display a message stating “The area of a rectangle with a width of 104.32 and a height of 15.654 is ….”. Showing the correct answer at the end of the message.

width = 104.32

height = 15.654

area = width * height

message = f"The area of a rectangle with a width of {width} and a height of {height} is {area}."

print(message)

Format Specifiers

The use of f-strings shown above may be convenient, but it does not really add much in terms of actual *formatting*. The real power of using f-strings comes from the use of **Format Specifiers**. A format specifier is a small string that is included after expressions within the braces; they must be prefixed by a colon ‘**:**’ character. The information provided within the format specifier allows control over the format or layout of the printed value.

For example, the following uses a format specifier to ensure the result is output to two decimal places. In this case the format specifier is given as `:.2f`.

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

When executed, with `r` set to `50`, the following output would be shown. Compare this with the earlier version that did not include the format specifier.

```
A circle with radius 50 has an area of 7853.98
```

Although format specifiers are difficult to understand at first, due to their somewhat cryptic syntax, they are very powerful. Also there are typical scenarios that often re-occur, such as the one above which is used to specify the *precision* of an output number.

TASK: Rewrite your earlier code that displayed the area of a rectangle, but include a format specifier that limits the displayed result to three decimal places.

width = 104.32

height = 15.654

area = width * height

message = f"The area of a rectangle with a width of {width} and a height of {height} is {area:.3f}."

print(message)

Another common usage scenario of a format specifier is to control the column *width* and *alignment*. A minimum width can be specified simply by providing a single numeric value after the '`:`' character, as in:.

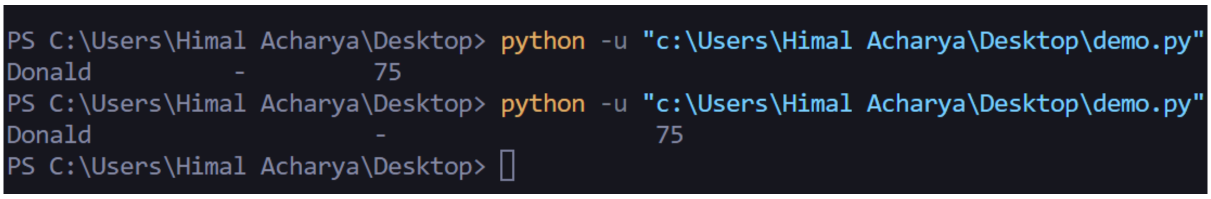
```
print(f"{name:15} - {age:10}")
```

If the `name` was "`Donald`", and `age` was `75`, then the above example would display the following. Notice the spacing that has been introduced due to the requested minimum column width.

```
Donald          -         75
```

When a column width is specified, the *alignment* of the values becomes important. In the above example the *string* `name` is aligned to the left of the column, but the *number* `age` is aligned to the right of the column. This is the default behaviour if no alignment information is provided within the format specifier - text to the left, numbers to the right.

TASK: Try setting the `name` and `age` variables to different values and executing the above `print()` statement multiple times. Notice the alignment and column width enforced due to the print specifier.



```
PS C:\Users\Himal Acharya\Desktop> python -u "c:\Users\Himal Acharya\Desktop\demo.py"
Donald          -         75
PS C:\Users\Himal Acharya\Desktop> python -u "c:\Users\Himal Acharya\Desktop\demo.py"
Donald                    -                   75
PS C:\Users\Himal Acharya\Desktop> 
```

The default alignment can be overridden by providing explicit alignment information as part of the format specifier. This is done by prefixing the column width value with one of the following characters:

<	Aligns the value left within the available space
>	Aligns the value right within the available space
^	Aligns the value to the centre within the available space
=	Adds padding after the sign (numerical values only).

For example, to right-align the name on the previous example, we would use the following variation:

```
print(f"{name:>15} - {age:10}")
```

Or, to centre both the name and age we could use:

```
print(f"{name:^15} - {age:^10}")
```

This would result in the following output:

```
    Donald      -     75
```

When padding is applied, the default *fill character* is a space. This can be changed however by prefixing the alignment character with an alternative value:

```
print(f"{name:@^15} - {age:#^10}")
```

This would result in the following output:

```
@@@@Donald@@@@@ - ####75####
```

TASK: Write a `print()` statement that displays the `name` and `age` values, with a column width of `20` for each, both right aligned, and with the `age` being shown to two decimal places. The fill character should be a dollar symbol `$`.

```
name = "Donald"
age = 75

print(f"{name:$>20} - {age:.2f}")
```

We have covered some of the common cases of format specifiers, however there is more to learn. Some annotated examples are shown below.

For more information see https://docs.python.org/3/library/string.html#formatspec

```
# Specifies a 0 'fill' character and displays as hexadecimal
val = 255
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
val = 23
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
item, cost = "bread", 12.36453
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
score = 125
print(f"high score is {score:-^20}")
high score is --------125---------
```

Formatting with str.format()

As already mentioned, there are multiple mechanisms that support *formatting* within Python. Since 'f-strings' was introduced fairly recently, the previously used methods are still commonly applied by programmers and are found in older code.

A very popular approach to formatting involves the use of the `str.format()` method. This approach works by first defining a string that contains *replacement fields*, which are identified within the string using braces `{ }`. Arguments then passed to the `format()` method replace each of these fields. It looks like this:

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

Unlike the f-string version, notice how the `format()` method is explicitly called with the values to be displayed.

Rather than just rely on the position of each *replacement field*, it is possible to provide a positional value. This means the replacement values do not necessarily need to appear in the same order when the method is called.

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

In this example, both *replacement fields* contain a number (which is a zero based index) identifying the parameter to be used for replacement. Notice how the `addr` is passed first when the `format()` method is called, but it will actually be printed last.

As an alternative to a positional value, a keyword may also be provided for each replacement field. Another example uses this:

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

In this case, the first replacement field can be identified by the keyword `id`, which is then provided as a keyword argument to the `format()` method.

TASK: Write some code that uses the `str.format()` method to display the area of a circle with a radius specified by the variable `r`. Use a *keyword replacement* field called `area` to identify the calculated area and refer to this when passing the value to the `str.format()` method. The output should look like the following, in the case where `r` is set to `52`.

```
A circle with radius 52 has an area of 8494.8665353068

r = 52
area = 3.14159 * r**2

output = "A circle with radius {} has an area of {:.13f}".format(r, area)
print(output)
```

Like f-strings the real power of using `str.format()` comes from the use of *format specifiers*. Luckily these can be used within *replacement fields* in exactly the same way as with f-strings; they appear after a colon '`:`' within the braces.

```
print("The cost of '{:^11}' was £{:8.2f}".format(item,cost))
```

This example aligns the first value '`item`' to the centre of a column that is 15 characters wide. Then displays the second value '`cost`' in a column at least 8 characters wide using two decimal places.

The similarities between f-strings and `str.format()` make it easy to use either approach. In most cases the f-strings version is slightly more elegant since it does not require an explicit call to the `format()` method. The real key to using either approach is to understand *format specifiers*.

TASK: Convert the f-string based statement below into an equivalent that uses the `str.format()` method to generate the same output.

```
print(f"{name:@^15} - {age:#^10}")

name = "Donald"
age = 75

output = "{name:@^15} - {age:#^10}".format(name=name, age=age)
print(output)
```

Alternative Formatting Approaches

The Python language originally used a formatting method loosely based on the techniques used in the 'C' programming language. This is sometimes referred to as `%`-formatting.

Although superseded by f-strings and `str.format()` it is still advisable to be aware of this approach, even though it probably should not be used in new code. It uses a '`%`' operator, between a *string* and associated values:

```
print("Name is %s, and age is %.2f" % (name, age))
```

The '`%`' values within the string itself represent the replacement fields and format specifier information, but the format of these is different from what we have seen available within f-strings and the `str.format()` method.

It also is possible to perform a certain amount of formatting without using any of the mechanisms described. A more manual approach can be taken simply by making calls to some of the string *methods* that are available.

Specifically *methods* such as `rjust()`, `ljust()` and `center()` can be used to align values within a specified column width. The following two statements result in the exactly the same output.

```
print(name.rjust(15), " - ", str(age).center(10))

print(f"{name:>15} - {age:^10}")
```

Using a manual approach is occasionally useful, but under most circumstances it is easier to use either the f-strings or the `str.format()` method to perform formatting. The resulting code is easier to understand, and probably easier to create!

File Handling

The ability to read and write to files is a common requirement for many programs, and it is not uncommon for nearly all I/O to be file based. All computer files are basically just a list of numbers, more specifically bytes. The actual *format* of the file determines what these bytes represent, but that knowledge is built into the programs that process the files rather than the file-system of the operating system.

At a very basic level files can be considered to contain either *text* or *binary* information. This distinction only needs to be made so that additional processing can be done when working with text files, such as identifying special characters that represent the end of a line.

The Python language provides built-in support for manipulating both *text* and *binary* based files. There are also modules available that can deal with well known *file-formats*, such as JSON and CSV files. Modules can also be found that will manipulate proprietary formats such as Excel, image file formats, movies, and more.

Opening, Reading and Closing

Before the contents of a file can be accessed it must be *opened*. This can be done with a single call to the `open()` function. This function can take various parameters, but must be always provided with the name of the file to be opened. This typically matches the name of the file as specified within the underlying operating system.

The `open()` function returns a *file object*, which we can then use to manipulate the file contents. So to open a text file for reading we could use the following call.

```
f = open("info.txt")
```

Following this call, the *file object* stored in the variable `f` allows us to refer to that specific file from within the program. For example, we can use this variable to read the entire contents of the file as follows:

```
file_contents = f.read()
```

Once we are done processing a file it must be closed, by calling the `close()` method.

```
f.close()
```

Files will most likely be closed when any program with them open terminates, possibly after a delay while the operating system notices. Nevertheless, it is always good practice to close any files before a program terminates.

TASK: Use a text editor to create a file called `info.txt` and enter the text shown below. Once the file has been created and saved, write a small program that:

1. Opens the file,
2. reads and prints the contents,
3. closes the file.

```
This is a text file
It contains multiple lines of text
This is the final line within the file

# Open the file in read mode
with open("info.txt", "r") as file:
    # Read and print the contents
    contents = file.read()
    print(contents)

# File is automatically closed when exiting the 'with' block
```

Rather than read the entire contents of a text file, it is more common to read it one line at a time. This is because the program usually needs to process or understand the contents of the file in some way, and it is often easier or more memory efficient to load each line individually.

An individual line can be read from an open file using the `readline()` method, or all the lines of a file can be read into a List using the `readlines()` method. The decision of which to use depends on how the program is intending to process the content, and the likely size of the file.

TASK: write a small program that opens the `info.txt` file, then reads and displays each of the three lines of text using calls to the `readline()` method. Remember to close the file once the content has been read.

# Open the file in read mode

with open("info.txt", "r") as file:

    # Read and display each line

    line1 = file.readline()

    print(line1.strip()) # Strip to remove the newline character

    line2 = file.readline()

    print(line2.strip())

    line3 = file.readline()

    print(line3.strip())

# File is automatically closed when exiting the 'with' block

When processing individual lines, it is very common to see lines being read within some kind of loop. The `for...in` statement supports direct *iteration* of a *file object*, doing the reading of each line automatically during each iteration. This can be quite neat:

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

TASK: write a small program that opens the `info.txt` file, then reads and displays each line of text using a `for...in` loop. Remember to close the file once the content has been read. # Open the file in read mode

with open("info.txt", "r") as f:

    # Read and display each line using a for...in loop

    for line in f:

        print(line.strip()) # Strip to remove the newline character

# File is automatically closed when exiting the 'with' block

Modes of Operation

If the `open()` function is called with just a filename, then this opens an existing *text* file for *reading*. However, we have the ability to pass additional parameters that allows the file to be opened in a different *mode* other than for reading.

The mode of operation is specified using a second string type parameter. This can contain a single character that specifies whether the file is to be opened for reading (`r`), writing (`w`), appending (`a`) or reading and writing (`r+`). An additional letter can also be appended that, if present, indicates the file should be processed as *binary* (`b`) rather than *text*.

For example:

```
# open new file for writing
f1 = open("nextfile.txt", "w")

# open existing file for appending
f2 = open("file123.txt", "a")

# open file for for reading and writing
f3 = open("fileABC.txt", "r+")

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

If a file that does not exist is opened for writing (`w`) or appending (`a`) then the file is automatically created. If it is opened for reading (`r`) or reading and writing (`r+`) then the file must exist otherwise an error will be reported.

The following example opens a *binary* file called '`image.png`' then reads and displays its contents. Working with binary files is slightly different to *text* files since the `read()` method returns a *bytes object* rather than a string. This example reads 8 bytes at a time from the file, and prints them in *hexadecimal*.

```
f = open("image.png", "rb")

bytes = f.read(8)

while bytes:
      print(bytes.hex())
      bytes = f.read(8)

f.close()
```

Writing to a file

For a file to be written to, it has to have been opened for writing (`w`), appending (`a`) or reading and writing (`r+`). Once opened then content can be written by calling the `write()` method:

```
f.write("this is written to the file")
```

TASK: Write a small program that opens the "`info.txt`" file in append (`a`) mode. Use the `write()` method to add an extra line of text saying "`this is an extra line`". Remember to close the file once the content has been read. Open the file with a text editor and examine the contents.

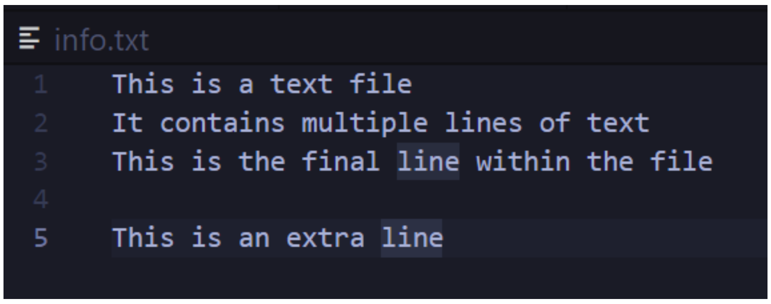
# Open the file in append mode

```
with open("info.txt", "a") as f:

    # Write an extra line of text

    f.write("\nThis is an extra line")

# File is automatically closed when exiting the 'with' block
```



```
info.txt
This is a text file
It contains multiple lines of text
This is the final line within the file

This is an extra line
```

The `write()` method takes a string type parameter, hence when writing non-string type values, these must be first converted. e.g.

```
age = 50
f.write(str(age))
```

Random Access

When a file is being processed the *file object* maintains a value that refers to the current position within the file. We can read the position using the `tell()` method, and change it using the `seek()` method.

Using seek() allows a program to perform random access, i.e. have the ability to change read and write locations on an open file. For example, the following call would move the position back to the beginning of the file, allowing content to be re-read.

```
f.seek(0)
```

The `seek()` method takes a second parameter (known as 'whence') that can be used to make the provided offset relative to something other than the start of the file. This takes one of the following values - start (0), current (1), or end (2) position of the file. For example, the following call would move the current position to the end of the file.

```
f.seek(0,2)
```

When a file is opened in text mode, only those values returned by `tell()` should be used for seeking purposes, any other offset value produces undefined behaviour. Also only a 'whence' value of `0` is allowed (the exception being when seeking to the very end of the file, as in the above example). Therefore random access tends to be of more use when working within *binary* rather than *text* files. Also note that files opened in append (`a`) mode always write to the end of the file, hence `seek()` is of limited use in this case.

Handling Exceptions

When working with files, dealing with run-time errors is a common requirement. This is because things can easily go wrong. The most obvious example would be that an attempt may be made to open a file that does not exist.

The Python language has a generic mechanism for dealing with such errors, called *exception handling*. This mechanism is used throughout the language and the supporting libraries.

As already mentioned, it is important that a file is always closed after use. This is true even if an error does occur. If a file is not closed properly, then new content may not be fully written or other programs within the same system may not be able to access the file. To help ensure that a file is always closed, whether an error (exception) occurs or not, there is a special construct provided within Python.

The `with` statement can be used to wrap file access within a code block. The advantage of this is that the file is *automatically closed* once the code block finishes. The `with` statement includes the opening of the file, then the code within the associated block accesses that file, for example:

```
with open("info.txt") as f:
      lines = f.readlines()
      print(lines)

# file 'f' automatically closed
```

Although this approach is not enforced, it is often the best method to use when working with files. It is certainly more Pythonic.

TASK: write a small program that opens the `info.txt` file, then reads and displays each line of text using a `for...in` loop. Rather than explicitly call the `close()` method, use the '`with`' statement to wrap the file handling code.

```
with open("info.txt", "r") as file:

    # Iterate through each line in the file

    for line in file:

        # Display each line of text

        print(line.strip())
```

____

Key Terminology

Understanding key terminology is important not only when programming but for understanding computer science in general. Certain terms, technologies and phrases will often re-occur when reading or learning about many computer science topics. Having an awareness of these is important, as it makes learning about the various subjects and communication with others in the field easier.

TASK: Look at each of the phrases below and ensure you understand what each of these means. For any that you do not understand, do a little research to find a definition of each term. This research may involve looking back over these notes, or the associated lecture notes. It may also involve searching for these terms on the Internet.

- ‘f-string’
- Format Specifier
- File modes
- Binary Files
- Random Access
- Exceptions

'f-string': A way to format strings in Python using the f-prefix, allowing expressions to be embedded.

Format Specifier: Special syntax for controlling the presentation of values in string formatting.

File modes: Represent access modes when working with files (e.g., 'r' for reading, 'w' for writing).

Binary Files: Files containing data in a format other than plain text (e.g., images, audio).

Random Access: Ability to directly access any data in a file, not just sequentially.

Exceptions: Events disrupting the normal flow of program instructions, handled using Python's exception mechanism.

Practical Exercises

You have now completed this tutorial. Now attempt to complete the associated exercises.