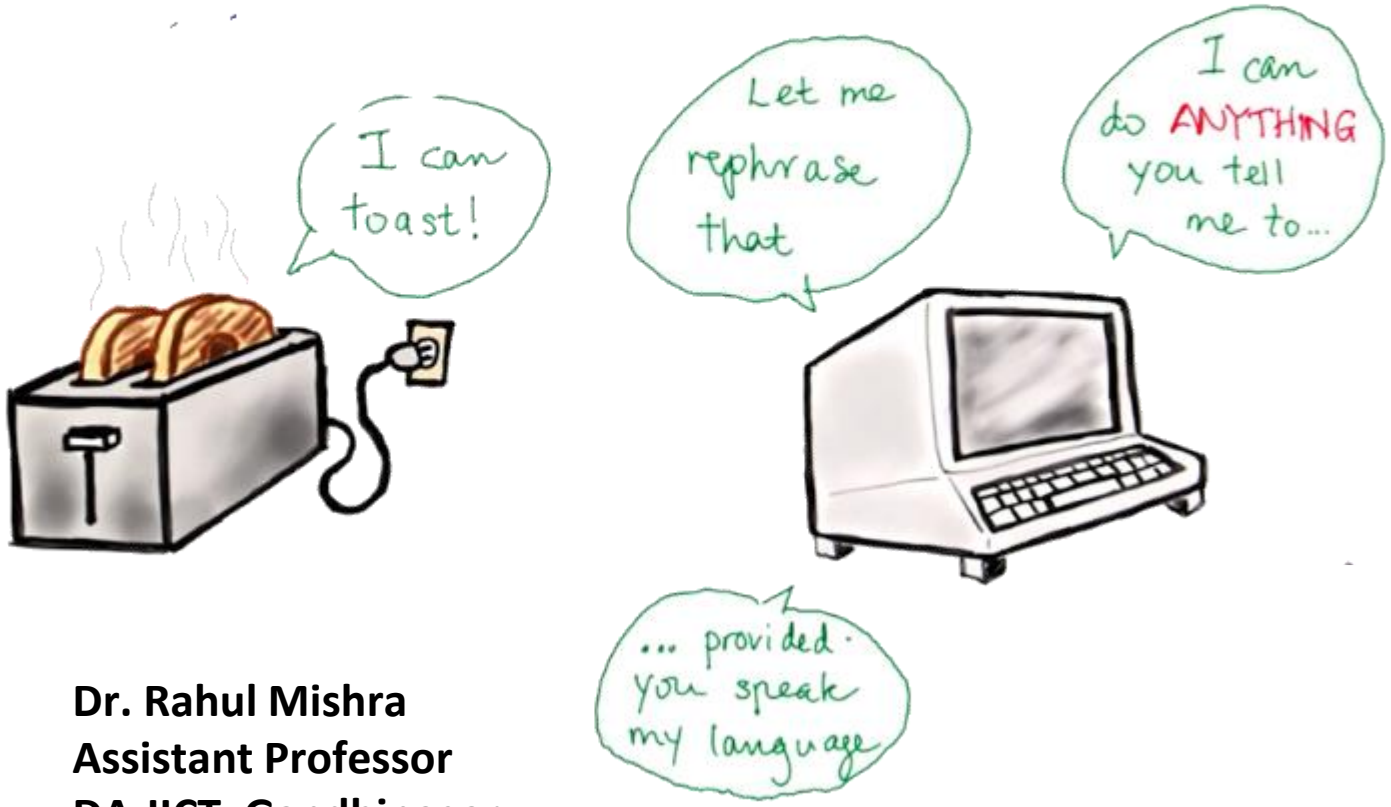
Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 14

Input and Output

Input and Output

- There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use.

- So far we've encountered two ways of writing values: ***expression statements and the print() function*****.**

- A third way is using the **write()** method of file objects; the standard output file can be referenced as **sys.stdout.**

- To use formatted string literals, begin a string with **f or F** before the opening quotation mark or triple quotation mark.

- Inside this string, you can write a ***Python expression between { and } characters that can refer to variables or literal values.***

Input and Output

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
>>>
```

- The **str.format()** method of strings requires more manual effort. You'll still use { and } to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
>>>
```

Input and Output

- We can perform the string handling using string slicing and concatenation operations to create any layout you can imagine.

- The string type has some methods that perform useful operations for padding strings to a given column width.

- The **str() function** is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is no equivalent syntax).

- For objects which don't have a particular representation for human consumption, **str() will return the same value as repr().**

- Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. **Strings, in particular, have two distinct representations.**

Input and Output

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
>>>
```

Input and Output

Formatted String Literals

- Formatted string literals *(also called f-strings for short)* let you include the value of Python expressions inside a string by prefixing the string with **f or F and writing expressions as {expression}.**

- An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds **pi to three places after the decimal**:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Input and Output

Formatted String Literals

- Passing an integer after the ':' will cause that field to be a minimum number of characters wide.

- This is useful for making columns line up.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
>>>
```

Input and Output

- Other modifiers can be used to convert the value before it is formatted. **'!a' applies ascii(), '!s' applies str(), and '!r' applies repr():**

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
>>>
```

- The = specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
```

The String format() Method

- Basic usage of the str.format() method looks like this:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
>>>
```

- The brackets and characters within them (called format fields) are replaced with the objects passed into the str.format() method.
- A number in the brackets can be used to refer to the position of the object passed into the str.format() method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
>>>
```

The String format() Method

- If keyword arguments are used in the str.format() method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}.'.format(
...       food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
>>>
```

- Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                                   other='Georg'))
The story of Bill, Manfred, and Georg.
>>>
```

The String format() Method

- If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position.

- This can be done by simply passing the dict and using square brackets '[]' to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
>>>
```

- This could also be done by passing the table dictionary as keyword arguments with the ** notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
>>>
```

The String format() Method

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>>
```

Manual String Formatting

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...

 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Manual String Formatting

- The **str.rjust()** method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left.
- There are similar methods **str.ljust() and str.center()**.
- These methods do not write anything, they just return a new string.
- *If the input string is too long, they don't truncate it, but return it unchanged*; this will mess up your column layout but that's usually better than the alternative, which would be lying about a value.
- There is another method, *str.zfill(), which pads a numeric string on the left with zeros.* It understands about plus and minus signs:

Manual String Formatting

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Old string formatting

The ***% operator (modulo)*** can also be used for string formatting.

Given *'string' % values, instances of % in string are replaced with zero or more elements of values.*

This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

Reading and Writing Files

- **open()** returns a file object, and is most commonly used with two positional arguments and one keyword argument: **open(filename, mode, encoding=None)**

```
f = open('workfile', 'w', encoding="utf-8")
```

- The first argument is a string containing the filename.
- The second argument is another string containing a few characters describing the way in which the file will be used.
- mode can be ***'r' when the file will only be read***, ***'w' for only writing (an existing file with the same name will be erased)***, and ***'a' opens the file for appending***; any data written to the file is automatically added to the end.
- ***'r+' opens the file for both reading and writing***. The mode argument is optional; 'r' will be assumed if it's omitted.

Reading and Writing Files

- Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding.

- If encoding is not specified, the default is platform dependent (see open()). Because UTF-8 is the modern de-facto standard, encoding="utf-8" is recommended unless you know that you need to use a different encoding.

- Appending a *'b' to the mode opens the file in binary mode.* Binary mode data is read and written as bytes objects. You can not specify encoding when opening file in binary mode.

- In text mode, the default when reading is to convert platform-specific line endings (**\n on Unix, \r\n on Windows) to just \n.**

- When writing in text mode, the default is to convert occurrences of \n back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files.

Reading and Writing Files

- It is good practice to use *the with keyword when dealing with file objects*.
- The advantage is that the file is properly closed after its suite finishes, *even if an exception is raised at some point.*
- Using *with is also much shorter than writing equivalent try-finally blocks:*

```
>>> with open('workfile', encoding="utf-8") as f:
...     read_data = f.read()
...
>>> # We can check that the file has been automatically closed.
... f.closed
True
>>> read_data
''
>>>
```

Reading and Writing Files

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` **might** result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```