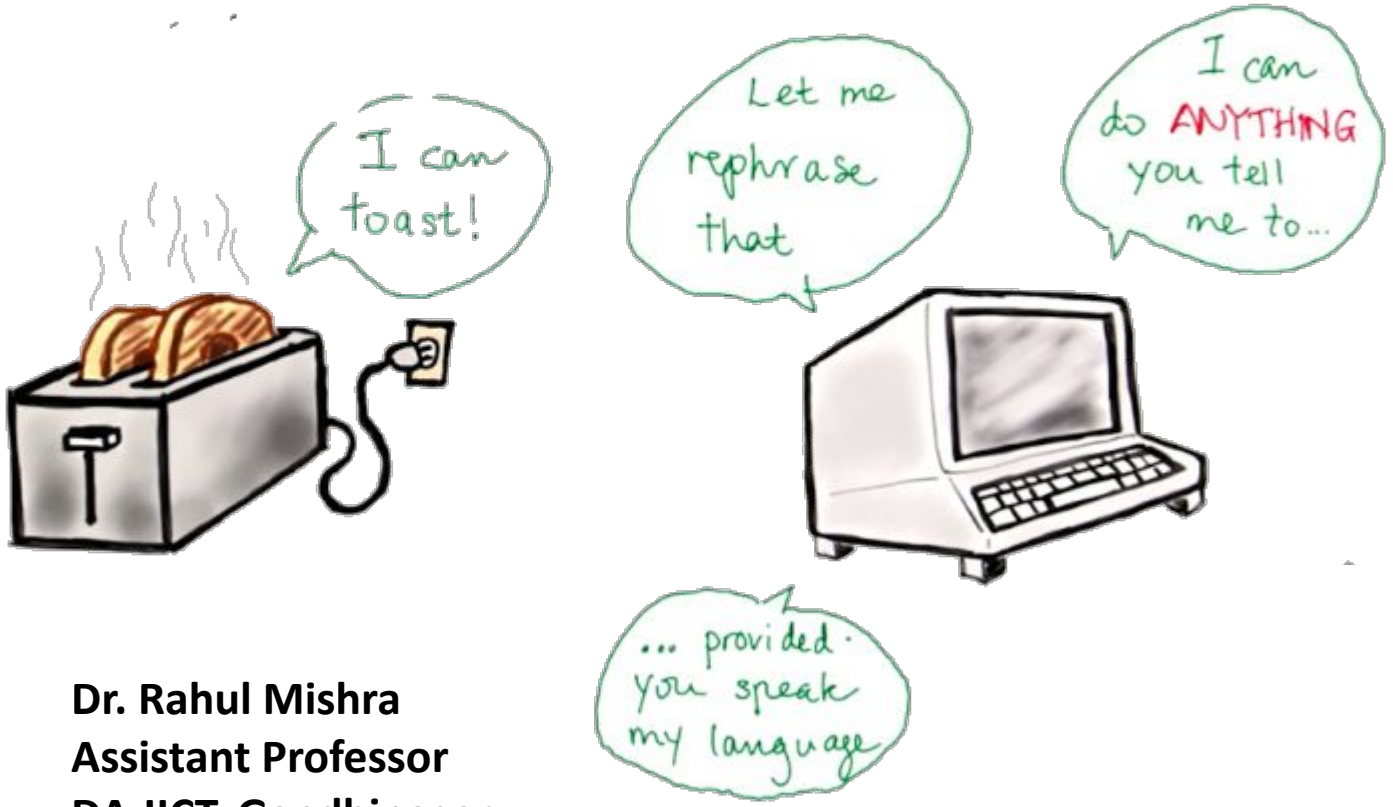
Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Lecture 15

Input and Output

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

- To read a file's contents, ***call f.read(size)***, which reads some quantity of data and returns it as a string *(in text mode) or bytes object (in binary mode)*.

- size is an optional numeric argument. When size is omitted or negative, the entire contents of the file will be read and returned; *it's your problem if the file is twice as large as your machine's memory.*

- Otherwise, at most size characters (in text mode) or size bytes (in binary mode) are read and returned.

- If the end of the file has been reached, *f.read() will return an empty string ('')*

```
>>>f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Reading and Writing Files

- ***f.readline()*** reads a single line from the file; a newline character **(\n)** is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline.

- This makes the return value unambiguous; if **f.readline()** returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

```
>>>f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
 ''
```

Reading and Writing Files

- For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

Reading and Writing Files

- *f.tell()* returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.
- To change the file object's position, use *f.seek(offset, whence)*.
- The position is computed from adding offset to a reference point; the reference point is selected by the whence argument.
- A whence value of 0 measures from the beginning of the file, *1 uses the current file position*, and 2 uses the end of the file as the reference point.
- whence can be omitted and defaults to 0, using the beginning of the file as the reference point.

Reading and Writing Files

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
>>>
```

Saving structured data with json

- Strings can easily be written to and read from a file.
- Numbers take a bit more effort, since the *read()* method only returns strings, which will have to be passed to a function like int(), which takes a string like '123' and returns its numeric value 123.
- When you want to save more complex data types *like nested lists and dictionaries*, parsing and serializing by hand becomes complicated.
- Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation).
- The standard module named json can take Python data hierarchies, and convert them to string representations; this process is called serializing.

Saving structured data with json

- Reconstructing the data from the string representation is called deserializing.

- Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

```
>>> import json
>>> x = [1, 'simple', 'list']
>>> json.dumps(x)
'[1, "simple", "list"]'
```

- Another variant of the dumps() function, called dump(), simply serializes the object to a text file. So if *f is a text file object opened for writing, we can do this:*

```
>>>json.dump(x, f)
```

Errors and Exceptions

1. *Syntax Errors*

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

- The parser repeats the offending line and displays a little *'arrow' pointing at the earliest point* in the line where the error was detected.
- The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function *print(), since a colon (':') is missing before it.*
- File name and *line number are printed so you know where to look in case the input came from a script.*

Errors and Exceptions

2. Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

- Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.

- Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str
(not "int") to str
>>>
```

Errors and Exceptions

2. Exceptions

- The last line of the error message indicates what happened.
- Exceptions come in different types, and the type is printed as part of the message: the types in the example are ***ZeroDivisionError, NameError and TypeError.***
- The string printed as the exception type is the name of the built-in exception that occurred.
- This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention).
- Standard exception names are built-in identifiers (not reserved keywords).

Errors and Exceptions

2. Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

Errors and Exceptions

3. Handling Exceptions

- It is possible to write programs that handle selected exceptions.
- Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program
- (using Control-C or whatever the operating system supports);
- note that a user-generated interrupt is signalled by raising the KeyboardInterrupt exception.

Errors and Exceptions

3. Handling Exceptions

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
Please enter a number: q
Oops!  That was no valid number.  Try again...
Please enter a number: 12
>>>
```

Errors and Exceptions

3. Handling Exceptions

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
...
Please enter a number: q
Oops!  That was no valid number.  Try again...
Please enter a number: 12
>>>
```

Errors and Exceptions

3. Handling Exceptions

The try statement works as follows.

- First, the *try clause* (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then, if its type matches the exception named after the except keyword, the *except clause* is executed, and then execution continues after the try/except block.
- If an exception occurs which does not match the exception named in the *except clause*, **it is passed on to outer try statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.**

Errors and Exceptions

3. Handling Exceptions

- A try statement may have more than one *except clause*, to specify handlers for different exceptions. At most one handler will be executed.

- Handlers only handle exceptions that occur in the corresponding *try clause*, not in other handlers of the same try statement.

- An *except clause* may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```