Lecture2

September 6, 2024

1 Lecture 2: Strings & Files

1.0.1 Objective:

This lecture aims to present `strings`, reviewing some of the most commonly encountered operations, particularly in text processing tasks. We will also examine the string class in greater detail, including how to use the `open()` function to open, read, and write to `files`.

1.1 0. Introduction

Strings in Python are not merely standalone objects; rather, they are sequences of characters that represent text data. Similar to numerical data types, which consist of sequences of digits, strings consist of sequences of characters. These sequences are fundamental in Python programming, as they are often used to construct and manipulate more complex data structures like lists, tuples, and dictionaries.

Python treats strings as immutable sequences, meaning that once a string is created, its contents cannot be changed. However, various operations can be performed on strings to generate new strings or extract specific information. These operations include concatenation, slicing, formatting, and various methods for searching and transforming the text. Understanding how to effectively manipulate strings is crucial, as strings are one of the most commonly used data types in Python, especially in tasks involving data processing, file handling, and user input.

Moreover, strings serve as the foundation for more advanced objects and data structures. For instance, strings can be used to represent elements within lists or tuples, and they play a critical role in tasks ranging from basic text processing to complex data analysis. As we delve deeper into Python, we will explore the extensive functionality provided by string operations and how they integrate with other Python data types.

In addition to handling strings, Python provides robust support for working with files, which is essential for tasks like reading input data, storing output, or simply managing data over the course of a program's execution. Python uses the `open()` function to handle files, allowing you to open, read, write, and close them efficiently.

The `open()` function is versatile, enabling you to open a file in various modes, such as reading ('r'), writing ('w'), appending ('a'), or even reading and writing in binary mode ('rb', 'wb'). Once a file is opened, you can perform operations like reading the entire file content into a string, reading it line by line, or writing data back into the file. After completing these operations, it is important to close the file using the `close()` method to free up system resources and ensure data integrity.

Understanding how to work with files in Python is critical, especially when dealing with real-world data that often needs to be read from or written to external files. This capability, combined with string manipulation techniques, forms the backbone of many Python programs, enabling efficient data processing, storage, and retrieval.

1.2 1. Strings

1.2.1 1.0. Definition of a string variable

A string variable is defined in the traditional manner by direct assignment using the equality symbol (as discussed in the section on variable creation). The only distinction between string variables and numerical variables is that the values of string variables must be enclosed in quotation marks during assignment. The three examples below illustrate this.

[30]:
```python
ch1 = 'Will you be at the meeting tonight?'
ch2 = '"Yes," he replies.'
ch3 = "Alright, I would appreciate it."
ch4 = """This sentence is a long string containing all types of quotes: " ' « » ␣
↪as well as many special characters."""

print(ch1)
print("==================================================")
print('\t')

print(ch2)
print("==================================================")
print('\t')

print(ch3)
print("==================================================")
print('\t')

print(ch4)
```

```
Will you be at the meeting tonight?
==================================================

"Yes," he replies.
==================================================

Alright, I would appreciate it.
==================================================

This sentence is a long string containing all types of quotes: " ' « » as well
as many special characters.
```

These four examples demonstrate different uses of quotation marks when defining a string variable.

- In the first example (`ch1`), single quotes are used because there is no issue with the string specified. Double quotes could also be used.

- In the second example (`ch2`), single quotes are used because the specified string already contains double quotes as values.
- In the third example (`ch3`), double quotes are used because the text contains apostrophes, which are actually single quotes.
- In the fourth example (`ch4`), triple quotes are used because the text contains not only single quotes, apostrophes, and double quotes but also spans multiple lines. In this case, triple quotes are necessary.

1.2.2 1.1. Indexing strings and slicing

A string is a sequence of ordered and indexed values. This means you can access each element in the sequence by specifying its index, similar to lists. Portions of a string can also be extracted using slices, with the general notation `[start=0]:[stop=len][:step=1]`. Indexing in Python starts at 0: the 1st element of a list is at index 0, the 2nd is at index 1, and so on. Thus, the n elements of a list are indexed from *0* to *n-1*. The examples below illustrate this.

Indexing:

To access a specific character in a string, you use the index inside square brackets. For example: - `ch[0]` returns the first character of the string `ch`. - `ch[2]` returns the third character. - `ch[-1]` returns the last character of the string.

Slicing:

Slicing allows you to extract a portion of the string. The syntax for slicing is `[start:stop:step]`, where: - `start` is the index of the first character you want to include in the slice. - `stop` is the index of the character just after the last character you want to include. - `step` specifies the interval between characters in the slice (optional; default is 1).

Examples of slicing: - `ch[:6]` returns a substring from the beginning of `ch` up to, but not including, index 6. - `ch[6:]` returns a substring from index 6 to the end of the string. - `ch[0:10:2]` returns every second character from index 0 to 10.

Examples:

```
ch = "Christelle"
print(ch[0])     # Returns 'C'
print(ch[2])     # Returns 'r'
print(ch[-1])    # Returns 'e', the last character
print(ch[:6])    # Returns 'Christ'
print(ch[6:])    # Returns 'elle'
print(ch[0:10:2]) # Returns 'Crsel'
```

These operations make it easy to manipulate and extract data from strings, which is crucial for text processing and data manipulation in Python.

---

[31]:
```
ch="Christelle"
print(ch[0]) # returns 'C'
print(ch[2]) # returns 'r'
print(ch[-1]) # returns 'e', the last element of ch
print(ch[:6]) # returns 'Christ'
print(ch[6 :]) # returns 'elle'
print(ch[0:10:2]) # returns 'Crsel'
```

```
C
r
e
Christ
elle
Crsel
```

The elements of a string are not defined by words separated by spaces but by the characters that make up the string. In fact, even if the string consists of a sentence with multiple words, indexing is done solely based on the individual characters that form the entire string.

You can manipulate a string (and generally any sequence) using functions (procedural concept) or methods (object-oriented concept). We will discuss this further below.

1.2.3 1.2. Length of a string**

To determine the length of a string, the `len()` function is used.

[32]:
```
ch = "Ceci est une chaîne de plusieurs mots"
print(len(ch))  # returns 37; The string 'ch' consists of 37 characters,␣
 ↪including spaces.
```

```
37
```

1.2.4 1.3. Addition of strings (Concatenation)

String addition refers to the process of placing strings together, one after the other, to form a single string. This operation is known as **concatenation**. To perform concatenation, the `+` symbol is used between the string variable names. The resulting string is then assigned to a new variable. Examples:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2  # Concatenates 'Hello' and 'World' with a space in between
print(result)  # Outputs: Hello World
```

[33]:
```
x = "Un petit pas pour l'homme,"
y = "un grand pas pour l'humanité"
z = x + " " + y  # Concatenates the two strings with a space in between
print(z)  # Outputs: 'Un petit pas pour l'homme, un grand pas pour l'humanité'
```

```
Un petit pas pour l'homme, un grand pas pour l'humanité
```

Be cautious when using the concatenation operator to combine a numeric value with a string to form a single string. For example:

```
x = 'Le prix du stylo est'
y = 5
```

```
z = 'euros'
```

If you want to concatenate x, y, and z to obtain the sentence 'Le prix du stylo est 5 euros', you can't directly do `x + y + z` because y is a numeric type. You need to first convert y to a string using the `str()` function. Here's how it should be done:

```
result = x + " " + str(y) + " " + z
print(result)  # Outputs: 'Le prix du stylo est 5 euros'
```

[34]:
```
x = 'Le prix du stylo est'
y = 5
z = 'euros'
ch = x + " " + " " + str(y) + " " + z
print(ch) # returns 'Le prix du stylo est5euros'
```

```
Le prix du stylo est  5 euros
```

1.2.5 1.4. Methods

Strings come with numerous features—referred to as **methods** in Object-Oriented Programming (OOP)—that facilitate their manipulation. The `dir()` function lists all available methods for strings. Among these, the most commonly used are `upper()`, `lower()`, `capitalize()`, `find()`, `replace()`, `count()`, `startswith()`, `endswith()`, and the `in` keyword.

These methods allow for operations such as converting text to different cases, searching for and replacing substrings, and checking string content for specific patterns. They are crucial for text processing and data manipulation tasks, enabling efficient handling and transformation of string data. Understanding and utilizing these methods effectively can greatly enhance the capability to work with and analyze text-based information in Python.

[35]:
```
title = "Python course"
help(title.replace)
```

```
Help on built-in function replace:

replace(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    If the optional argument count is given, only the first count occurrences
are
    replaced.
```

[36]:
```
enfant, peluche = "Calvin", 'Hobbes'  # Multiple assignment

titre = enfant + ' et ' + peluche  # +: String concatenation
```

```python
print(titre)
print("==================================================")
print('\t')

print(titre.replace('et', '&'))  # Substring replacement: 'Calvin & Hobbes'
print("==================================================")
print('\t')

print(' & '.join(titre.split(' et ')))  # Splitting and joining
print("==================================================")
print('\t')

print('Hobbes' in titre)  # in: Inclusion test
print("==================================================")
print('\t')

print(titre.find("Hobbes"))  # str.find: Substring search

print(titre.center(30, '-'))  # '-------Calvin et Hobbes-------'

dir(str)  # Lists all string methods
```

```
Calvin et Hobbes
==================================================

Calvin & Hobbes
==================================================

Calvin & Hobbes
==================================================

True
==================================================

10
-------Calvin et Hobbes-------
```

```
[36]: ['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'removeprefix',
 'removesuffix',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

1.2.6 1.5. Formatting

It is often necessary to retrieve the value of a variable and incorporate it into a string to form a new, usable string for other purposes. For instance, consider a variable `prix` defined as follows:

```
prix = 2
```

We want to display a message like `"The price of the pen is 2 euros"`.

The goal of formatting is to embed the value of a variable within a string. There are three primary methods to achieve this:

- Using the concatenation operator `+`
- Using the formatting operator `%`
- Using the `str.format()` method

We will illustrate just one example using the `str.format()` method. The formatting system allows for precise control over the conversion of variables into strings. It primarily relies on the `str.format()` method, which enables detailed specification of how variables should be inserted and formatted within a string. For more information, you can refer to the formatting documentation.

[37]:
```
print("{nom} a {age} ans".format(nom='Calvin', age=6)) # 'Calvin a 6 ans'
print("==================================================")
print('\t')
print("{} a {} ans".format('Calvin', 6)) # Shortcut 'Calvin a 6 ans'
print("==================================================")
print('\t')

pi = 3.1415926535897931
print("{x:f} {x:.2f} {y:f} {y:g}".format(x=pi, y=pi*1e9)) # '3.141593 3.14␣
↪3141592653.589793 3.14159e+09'
```

```
Calvin a 6 ans
==================================================

Calvin a 6 ans
==================================================

3.141593 3.14 3141592653.589793 3.14159e+09
```

Explanation:

- **Named Fields**: In the first example, `"{nom} a {age} ans".format(nom='Calvin', age=6)` uses named fields in the format string. The placeholders `{nom}` and `{age}` are replaced by the values provided in the `format` method.
- **Positional Fields**: The second example, `"{} a {} ans".format('Calvin', 6)`, uses positional placeholders `{}`. The values are inserted in the order they appear in the `format` method.
- **Formatting Numbers**: The third example demonstrates formatting numerical values. `"{x:f} {x:.2f} {y:f} {y:g}".format(x=pi, y=pi*1e9)` formats the floating-point number `pi` in various ways:
  - `{x:f}` displays `pi` with six decimal places.
  - `{x:.2f}` formats `pi` to two decimal places.
  - `{y:f}` shows `pi*1e9` with six decimal places.
  - `{y:g}` formats `pi*1e9` in a more compact form using scientific notation.

[38]:
```
print("Calvin and Hobbes\nScientific progress goes 'boink'") # Calvin and␣
↪Hobbes Scientific progress goes 'boink'
```

```
Calvin and Hobbes
Scientific progress goes 'boink'
```

[39]:
```
print("{0} fois {1} font {2}".format(3, 4, 3*4)) # Formatting and display
# Output: 3 fois 4 font 12
# A small exercise in this section is to display a multiplication table.
```

```
3 fois 4 font 12
```

Explanation:

- **Indexed Fields**: In the example `"{0} fois {1} font {2}".format(3, 4, 3*4)`, the `{0}`, `{1}`, and `{2}` are positional placeholders. They are replaced by the values `3`, `4`, and `3*4` (which evaluates to `12`) respectively. This results in the output: `3 fois 4 font 12`.

- **Multiplication Table Exercise**: The comment suggests a small exercise where one can display a multiplication table, which would involve iterating through numbers and formatting them into a table structure using similar formatting techniques.

In concluding this section, we mention the study of regular expressions (regex or `re`), which will not be discussed in this course. As a guide, we can say that regular expressions are extensions of the `str()` function that allow for more complex pattern-matching operations than what the `str()` functions alone can achieve.

Practical

Explore the functions:

1. `upper()`, `lower()`, and `capitalize()`
2. `find()`
3. `replace()`
4. `count()`
5. `split()`
6. `startswith()` and `endswith()`
7. `islower()` and `isupper()`
8. `istitle()`
9. `isalpha()` and `isalnum()` 10.`isdigit()`

Note: It is important to distinguish between a character value and an alphabetical value. An alphabetical value is always a string, whereas a string is not necessarily an alphabetical value. In Python, a character value is always enclosed in quotes (single, double, or triple), while a numeric value is expressed without quotes. There are two main types of data: numeric data and character data. It is worth noting that a numeric value enclosed in quotes is automatically treated as a character value, even if it is not alphabetical. Keeping these details in mind is crucial when handling sequences of values in Python (see the example below):

```
x = 12 # x is a numeric variable
y = "12" # y is a string formed by digits
z = "mon texte" # z is a string formed by alphabetical values
k = "Ce stylo coûte 5 euros" # k is a string formed by alphanumeric values
```

For more details on string processing, refer to this page.

1.3 2. File I/O

File I/O (Input/Output) in Python involves reading from and writing to files. Python provides several built-in functions and methods to handle files. The primary function for file manipulation is `open()`, which allows you to open a file and returns a file object. This file object provides methods for reading and writing data. Here's a detailed overview with various examples:

1.3.1 Basic File Operations

1. **Opening and Closing a File**
    - To open a file, use the `open()` function. The basic syntax is `open(filename, mode)`, where `filename` is the name of the file and `mode` is the mode of operation (`'r'` for read, `'w'` for write, `'a'` for append, etc.).
    - To close a file, use the `close()` method of the file object.

```python
# Opening a file in read mode
file = open('example.txt', 'r')
# Perform operations on the file
file.close()
```

2. **Reading from a File**
   - **Read the entire file**: `read()` method reads the entire content of the file.
   - **Read line by line**: `readline()` method reads a single line from the file, and `readlines()` method reads all lines into a list.

```python
# Read entire file
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()

# Read line by line
file = open('example.txt', 'r')
line = file.readline()
while line:
    print(line, end='')
    line = file.readline()
file.close()

# Read all lines into a list
file = open('example.txt', 'r')
lines = file.readlines()
print(lines)
file.close()
```

3. **Writing to a File**
   - **Write a string**: `write()` method writes a string to the file.
   - **Write multiple lines**: `writelines()` method writes a list of strings to the file.

```python
# Write a string to a file
file = open('example.txt', 'w')
file.write('Hello, World!')
file.close()

# Write multiple lines to a file
lines = ['First line\n', 'Second line\n', 'Third line\n']
file = open('example.txt', 'w')
file.writelines(lines)
file.close()
```

4. **Appending to a File**
   - To append data to an existing file, open the file in append mode (`'a'`).

```python
file = open('example.txt', 'a')
file.write('This is an appended line.\n')
file.close()
```

5. **Using `with` Statement**
   - The `with` statement is used for resource management. It ensures that the file is properly closed after its suite finishes, even if an exception is raised.

```python
# Reading a file using with statement
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

# Writing to a file using with statement
with open('example.txt', 'w') as file:
    file.write('This is a new content.')
```

6. **File Modes**
   - `'r'`: Read (default mode, opens the file for reading)
   - `'w'`: Write (creates a new file or truncates an existing file)
   - `'a'`: Append (opens the file for appending)
   - `'b'`: Binary (reads or writes in binary mode, e.g., `'rb'` or `'wb'`)
   - `'t'`: Text (default mode, opens the file in text mode, e.g., `'rt'` or `'wt'`)

```python
# Binary mode example
with open('example.jpg', 'rb') as file:
    content = file.read()
    # Process binary data
```

7. **File Positioning**
   - `seek(offset, whence)`: Moves the file pointer to a specific position.
   - `tell()`: Returns the current file pointer position.

```python
with open('example.txt', 'r') as file:
    file.seek(10)  # Move to the 10th byte
    print(file.tell())  # Print the current file pointer position
    content = file.read(20)  # Read 20 bytes from the current position
    print(content)
```

8. **Working with File Paths**
   - Use the `os` and `pathlib` modules to handle file paths.

```python
import os
from pathlib import Path

# Get current working directory
print(os.getcwd())

# Join paths
file_path = os.path.join('folder', 'example.txt')

# Using pathlib
path = Path('folder') / 'example.txt'
```

9. **Handling File Exceptions**
   - Use `try` and `except` blocks to handle file-related exceptions.

```python
try:
    with open('example.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print('File not found.')
except IOError:
    print('An IOError occurred.')
```

These examples cover a wide range of file operations in Python. Whether you are reading from, writing to, or managing files, Python's file I/O capabilities provide a flexible and powerful way to handle file data.

The help information for open is below:

[40]:
```
help(open)
```

```
Help on built-in function open in module io:

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
    Open file and return a stream.  Raise OSError upon failure.

    file is either a text or byte string giving the name (and the path
    if the file isn't in the current working directory) of the file to
    be opened or an integer file descriptor of the file to be
    wrapped. (If a file descriptor is given, it is closed when the
    returned I/O object is closed, unless closefd is set to False.)

    mode is an optional string that specifies the mode in which the file
    is opened. It defaults to 'r' which means open for reading in text
    mode.  Other common values are 'w' for writing (truncating the file if
    it already exists), 'x' for creating and writing to a new file, and
    'a' for appending (which on some Unix systems, means that all writes
    append to the end of the file regardless of the current seek position).
    In text mode, if encoding is not specified the encoding used is platform
    dependent: locale.getencoding() is called to get the current locale
encoding.
    (For reading and writing raw bytes use binary mode and leave encoding
    unspecified.) The available modes are:

    ========= ===============================================================
    Character Meaning
    --------- ---------------------------------------------------------------
    'r'       open for reading (default)
    'w'       open for writing, truncating the file first
    'x'       create a new file and open it for writing
    'a'       open for writing, appending to the end of the file if it exists
    'b'       binary mode
    't'       text mode (default)
    '+'       open a disk file for updating (reading and writing)
    ========= ===============================================================

    The default mode is 'rt' (open for reading text). For binary random
    access, the mode 'w+b' opens and truncates the file to 0 bytes, while
    'r+b' opens the file without truncation. The 'x' mode implies 'w' and
    raises an `FileExistsError` if the file already exists.
```

```
Python distinguishes between files opened in binary and text modes,
even when the underlying operating system doesn't. Files opened in
binary mode (appending 'b' to the mode argument) return contents as
bytes objects without any decoding. In text mode (the default, or when
't' is appended to the mode argument), the contents of the file are
returned as strings, the bytes having been first decoded using a
platform-dependent encoding or using the specified encoding if given.

buffering is an optional integer used to set the buffering policy.
Pass 0 to switch buffering off (only allowed in binary mode), 1 to select
line buffering (only usable in text mode), and an integer > 1 to indicate
the size of a fixed-size chunk buffer.  When no buffering argument is
given, the default buffering policy works as follows:

* Binary files are buffered in fixed-size chunks; the size of the buffer
  is chosen using a heuristic trying to determine the underlying device's
  "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`.
  On many systems, the buffer will typically be 4096 or 8192 bytes long.

* "Interactive" text files (files for which isatty() returns True)
  use line buffering.  Other text files use the policy described above
  for binary files.

encoding is the name of the encoding used to decode or encode the
file. This should only be used in text mode. The default encoding is
platform dependent, but any encoding supported by Python can be
passed.  See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to
be handled---this argument should not be used in binary mode. Pass
'strict' to raise a ValueError exception if there is an encoding error
(the default of None has the same effect), or pass 'ignore' to ignore
errors. (Note that ignoring encoding errors can lead to data loss.)
See the documentation for codecs.register or run 'help(codecs.Codec)'
for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text
mode). It can be None, '', '\n', '\r', and '\r\n'.  It works as
follows:

* On input, if newline is None, universal newlines mode is
  enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
  these are translated into '\n' before being returned to the
  caller. If it is '', universal newline mode is enabled, but line
  endings are returned to the caller untranslated. If it has any of
  the other legal values, input lines are only terminated by the given
  string, and the line ending is returned to the caller untranslated.
```

```
    * On output, if newline is None, any '\n' characters written are
      translated to the system default line separator, os.linesep. If
      newline is '' or '\n', no translation takes place. If newline is any
      of the other legal values, any '\n' characters written are translated
      to the given string.

    If closefd is False, the underlying file descriptor will be kept open
    when the file is closed. This does not work when a file name is given
    and must be True in that case.

    A custom opener can be used by passing a callable as *opener*. The
    underlying file descriptor for the file object is then obtained by
    calling *opener* with (*file*, *flags*). *opener* must return an open
    file descriptor (passing os.open as *opener* results in functionality
    similar to passing None).

    open() returns a file object whose type depends on the mode, and
    through which the standard file operations such as reading and writing
    are performed. When open() is used to open a file in a text mode ('w',
    'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open
    a file in a binary mode, the returned class varies: in read binary
    mode, it returns a BufferedReader; in write binary and append binary
    modes, it returns a BufferedWriter, and in read/write mode, it returns
    a BufferedRandom.

    It is also possible to use a string or bytearray as a file for both
    reading and writing. For strings StringIO can be used like a file
    opened in a text mode, and for bytes a BytesIO can be used like a file
    opened in a binary mode.
```

In the code below, I've opened a file that contains one line:

```
$ cat testfile.txt
abcde
fghij
aims class
senagal 2024
this is really cool
```

Now let's open this file in Python:

[41]:
```
f = open('testfile.txt','r')
```

[42]:
```
s = f.read(3)
print(s)
```

```
abc
```

We read the first three characters, where each character is a byte long. We can see that the file handle points to the 4th byte (index number 3) in the file:

```
[43]: f.tell()
```

```
[43]: 3
```

```
[44]: f.read(1)
```

```
[44]: 'd'
```

```
[45]: f.close() # close the old handle
```

```
[46]: f.read()  # can't read anymore because the file is closed.
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [46], line 1
----> 1 f.read()  # can't read anymore because the file is closed.

ValueError: I/O operation on closed file.
```

The file we are using is a long series of characters, but two of the characters are new line characters. If we looked at the file in sequence, it would look like "abcdenfghijn". Separating a file into lines is popular enough that there are two ways to read whole lines in a file. The first is to use the `readlines()` method:

```
[47]: f = open('testfile.txt','r')
      lines = f.readlines()
      print(lines)
      f.close()  # Always close the file when you are done with it
```

```
['abcde\n', 'fghij\n', 'aims class\n', 'senagal 2024\n', 'this is really
cool\n']
```

A very important point about the readline method is that it *keeps* the newline character at the end of each line. You can use the strip() method to get rid of the string.

File handles are also iterable, which means we can use them in for loops or list extensions:

```
[48]: f = open('testfile.txt','r')
      lines = [line.strip() for line in f]
      f.close()
      print(lines)
```

```
['abcde', 'fghij', 'aims class', 'senagal 2024', 'this is really cool']
```

```
[49]: lines = []
      f = open('testfile.txt','r')
      for line in f:
```

```
    lines.append(line.strip())
f.close()
print(lines)
```

```
['abcde', 'fghij', 'aims class', 'senagal 2024', 'this is really cool']
```

These are equivalent operations. It's often best to handle a file one line at a time, particularly when the file is so large it might not fit in memory.

The other half of the story is writing output to files. We'll talk about two techniques: writing to the shell and writing to files directly.

If your program only creates one stream of output, it's often a good idea to write to the shell using the print function. There are several advantages to this strategy, including the fact that it allows the user to select where they want to store the output without worrying about any command line flags. You can use `\>` to direct the output of your program to a file or use `|` to pipe it to another program.

Sometimes, you need to direct your output directly to a file handle. For instance, if your program produces two output streams, you may want to assign two open file handles. Opening a file for reading simply requires changing the second option from `r` to `w` or `a`.

Caution! Opening a file with the 'w' option means start writing *at the beginning*, which may overwrite old material. If you want to append to the file without losing what is already there, open it with 'a'.

Writing to a file uses the `write()` command, which accepts a string.

[50]:
```
outfile = open('outfile.txt','w')
outfile.write('This is the first line!')
outfile.close()
```

Another way to write to a file is to use `writelines()`, which accepts a list of strings and writes them in order. *Caution!* `writelines` does not append newlines. If you really want to write a newline at the end of each string in the list, add it yourself.

1.4 Further reading

0. Programming with Python; Section 6
1. Programming with Python; Section 7
2. Programming with Python; Section 8
3. Programming with Python; Section 26