

# Course 2: Python Data Structures

The course introduces Python's core data structures—lists, dictionaries, and tuples—and shows how to use them for more advanced data analysis beyond basic procedural programming.

## Chapter 6: Strings

Covering Strings and moving into data structures.

### ➤ Part I

Strings – Part 1

PYTHON FOR EVERYBODY M

## Looking Inside Strings



- We can get at any single character in a string using an index specified in **square brackets**
- The index value must be an integer and starts at zero
- The index value can be an expression that is computed

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruit = 'banana'
>>> letter = fruit[1]
>>> print(letter)
a
>>> x = 3
>>> w = fruit[x - 1]
>>> print(w)
n
```

- You'll get a **an index error** if you attempt to index beyond the end of a string.

```
>>> print(fruit[6])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

- String have length: The built in **len()** function gives us the length.  

```
>>> print(len(fruit))
6
```

### Looping through Strings

Using a **while** statement and an **iteration variable**, and the **len** function, we can construct a loop to look at each of the letters in a string individually

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(index, letter)
    index = index + 1
```

- A definite loop using a **for** statement is much more elegant
- The **iteration variable** is completely taken care of by the **for** loop

```
fruit = 'banana'
for letter in fruit:
    print(letter)
```

### Slicing Strings

```
>>> s = 'Monty Python'
```

Monty Python  
0 1 2 3 4 5 6 7 8 9 10 11

```
>>> print(s[ :2])
Mo
```

```
>>> print(s[0:4])
```

Mont

We can also look at any continuous section of a string using a **colon operator**

```
>>> print(s[8: ])
```

```
>>> print(s[6:7])
```

P

The second number is one beyond the end of the slice — “**up to but not including**”.

```
thon
```

```
>>> print(s[6:20])
```

Python

If the second number is beyond the end of the string, it stops at the end

```
>>> print(s[:])
Monty Python
```

# If we leave off the first number or the last of the slice, its assumed to be the **beginning or end** of the string respectively.

### ➤ Part II: Manipulating Strings

#### String Concatenation

When the **+** operator is applied to strings, it means **“concatenation”**

```
>>> a = 'Hello'
>>> b = a + 'There'
>>> print(b)
HelloThere
>>> c = a + ' ' + 'There'
>>> print(c)
Hello There
>>>
```

#### Using **in** as a logical operator

- The **in** keyword can also be used to check to see if one string is “in” another string
- The **in** expression is a logical expression that returns **True** or **False** and can be used in an **if** statement

```
>>> fruit = 'banana'
>>> 'n' in fruit
True
>>> 'm' in fruit
False
>>> 'nan' in fruit
True
>>> if 'a' in fruit :
...     print('Found it!')
...
Found it!
>>>
```

## String Comparisons

```
if word == 'banana':
    print('All right, bananas.')

if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.)
```

## Common String Manipulation Methods

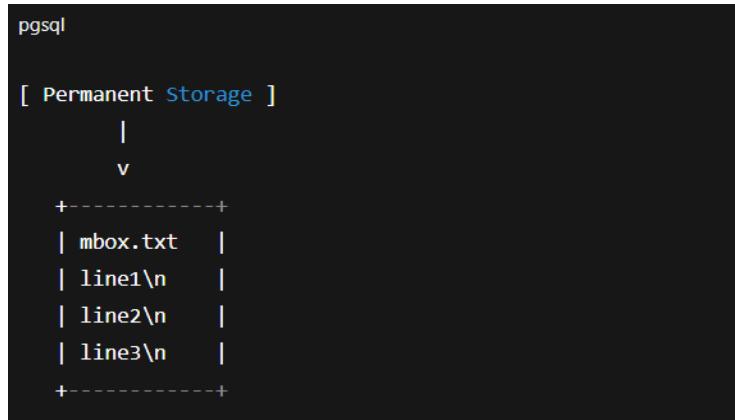
Code Example	Output
"Hello".lower()	hello
"hello".upper()	HELLO
"hello world".capitalize()	Hello world
"hello world".title()	Hello World
" hello ".strip()	hello
"Hello".startswith("He")	True
"Hello".endswith("lo")	True
"Hello".isalpha()	True
"Hello123".isalnum()	True
"12345".isnumeric()	True

Code Example	Output
"one,three".split(",")	['one', 'three']
"-".join(["a", "b", "c"])	a-b-c
"hello".find("e")	1
"hello".index("e")	1
"hallo world".count("o")	2
`"one,three".replace(",","")`	)`
len("hello")	5
"hello".startswith("h")	True
"hello world".split()	['hello','world']
"python".center(10, "*")	python

## Chapter 7: Files

Files store information permanently. Flat text files contain lines of text separated by newline characters.

### Diagram



### File Structure Internally

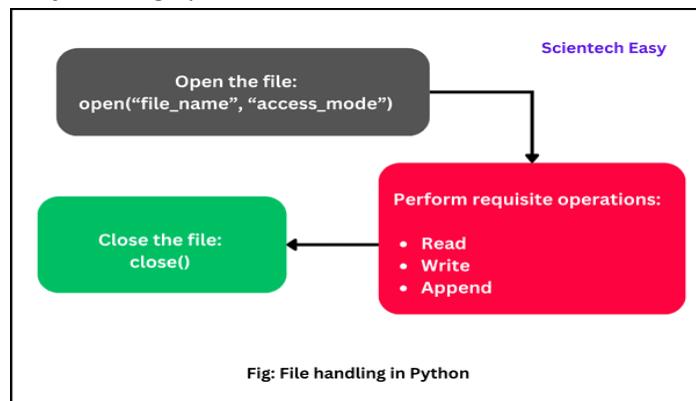
Files are actually one long sequence of characters. Newlines are what break them into lines.

```
A long stream of characters:  
A B C D \n E F G H \n I J K ...  
  
Line breaks created by \n
```

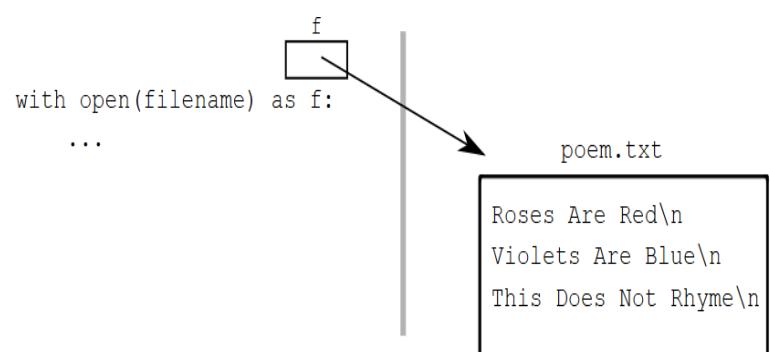
Newlines mark the end of lines in a file. written as \n

## Opening Files

### Way 1: Using open



### Way 2: Using with open



It's important to close a file after it has been used to free up system resources. Python automatically closes a file when used within a with statement. Otherwise, you should manually close it using `close()`.

## Reading Files

Once a file is opened, you can read its content by using functions below.

```
with open("example.txt", "r") as file:  
    content = file.read()  
    print(content) # Prints the entire file content
```

- **read()**: Reads the entire file.
- **readline()**: Reads one line at a time.
- **readlines()**: Reads all lines and returns them as a list.

## Writing to a File

Writing to a file in Python can be done using the write() method.

Make sure to open the file in write mode ("w") or append mode ("a").

```
with open("example.txt", "w") as file:  
    file.write("This is a new line of text.")
```

This example overwrites the file content with new text. If you want to add to the existing content without overwriting, use the append mode ("a").

- **write()**: It directly writes the string passed as an argument into the file.
- **readlines()**: Its used to write multiple lines into a file, with a list of string elements passed as an argument.

## Appending to a File

You can add content to an existing file using the append mode.

```
with open("example.txt", "a") as file:  
    file.write("\nThis is an appended line.")
```

## File Handling Modes

Here's a breakdown of the most commonly used file modes:

- "r": Read mode (Default).
- "w": Write mode (creates or overwrites).
- "a": Append mode (adds to the existing content).
- "b": Binary mode for non-text files (e.g., images, audio).
- "r+": Read and write mode.

## Handling Exceptions in File Handling

It's good practice to handle exceptions (like file **not found** errors) when working with files.

```
try:  
    with open("non_existing_file.txt", "r") as file:  
        content = file.read()  
except FileNotFoundError:  
    print("File not found!")
```

## Iterating Lines with rstrip()

```
filename = 'your_file.txt'  
  
with open(filename, 'r') as file:  
    for line in file:  
        # Remove trailing characters (by default, all whitespace, including '\n').  
        processed_line = line.rstrip()  
  
        print(processed_line)
```

## Chapter 8: Lists

As problems get more complex, we need variables that can hold more than one value. Until now we've used simple variables that store a single number or string. With lists, we can store many values in one variable and access them by index. These multi-value variables are called "collections" or "data structures."

### Part I

#### 1. What a list is?

A *list* is a single variable that holds multiple values.

You create one with **square brackets**:

```
friends = ["Joseph", "Glenn", "Sally"]  
carryon = ["socks", "shirt", "perfume"]
```

Lists can hold **any mix of types**:

```
mixed = ["red", 24, 98.6]  
nested = [1, [2, 3], 7] # lists can contain lists
```

#### 2. Lists & Definite Loops – Iterating through a list

```
friends = ['Joseph', 'Glenn', 'Sally']  
for friend in friends :  
    print('Happy New Year:', friend)  
print('Done!')
```

Happy New Year: Joseph  
Happy New Year: Glenn  
Happy New Year: Sally  
Done!



## Looking Inside Lists

Just like strings, we can get at any single element in a list using an index specified in **square brackets**

Joseph	Glenn	Sally
0	1	2

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]  
>>> print(friends[1])  
Glenn  
>>>
```

#### 3. Mutability (you *can* change lists)

- Strings are "**immutable**" - we cannot change the contents of a string - we must make a **new string** to make any change
- Lists are "**mutable**" - we can **change** an element of a list using the **index operator**

```
>>> fruit = 'Banana'  
>>> fruit[0] = 'b'  
Traceback  
TypeError: 'str' object does not support item assignment  
>>> x = fruit.lower()  
>>> print(x)  
banana  
>>> lotto = [2, 14, 26, 41, 63]  
>>> print(lotto)  
[2, 14, 26, 41, 63]  
>>> lotto[2] = 28  
>>> print(lotto)  
[2, 14, 28, 41, 63]
```

#### 4. Length

`len()` works on lists just like on strings:

```
len([1,2,3,4]) #4
```

#### 5. `range()` and counted loops

`range(n)` gives you numbers from 0 to n-1, useful for indexed loops:

```
for i in range(len(friends)):  
    print(i, friends[i])
```

Equivalent to the simpler loop—just gives you the index.

## Part II: Manipulating Lists

Operation	Example	Output / Notes
Concatenate lists	a=[1,2]; b=[3,4]; a+b	[1, 2, 3, 4]
( Works the <b>same way as strings</b> )	nums=[9,41,12,3,74,15]; nums[1:3]	[41, 12]
	nums[:4]	[9, 41, 12, 3]
	nums[3:]	[3, 74, 15]
Membership "in"	9 in nums	True
Membership "not in"	20 not in nums	True
Indexing	friends=["Joseph","Glenn","Sally"]; friends[1]	"Glenn"
Replace element	lotto=[2,14,26]; lotto[2]=28	[2, 14, 28]
Append	stuff=[]; stuff.append("book")	["book"]
Insert	stuff.insert(1,"pen")	["book", "pen"]
Remove	nums=[3,9,3]; nums.remove(3)	[9, 3]
Pop last	nums.pop()	Returns last element; list shrinks
Pop at index	nums.pop(1)	Returns element at index 1
Extend	a=[1,2]; a.extend([3,4])	[1,2,3,4]
Sort	friends.sort()	List becomes alphabetized
Reverse	nums.reverse()	Reverses list in place
len()	len([1,2,3])	3
max()	max([3,9,5])	9
min()	min([3,9,5])	3
sum()	sum([3,9,5])	17
Average	sum(nums)/len(nums)	Numeric average
range()	list(range(4))	[0,1,2,3]
For-each loop	for x in friends: print(x)	Prints each element
Indexed loop	for i in range(len(friends)): print(i,friends[i])	Prints index + element
Build list (append)	vals=[]; vals.append(3); vals.append(9)	[3, 9]
Empty list	list() or []	[]
Average (manual)	total+=x; count+=1	Low memory usage
Average (list-based)	nums.append(x); sum(nums)/len(nums)	Easier but uses more memory

**Check Python List Docs:** See the official Python documentation for full list methods, behaviors, and examples.

## Part III: Lists and Strings

### Best Friends: Strings and Lists

```
>>> abc = 'With three words'
>>> stuff = abc.split()
>>> print(stuff)
['With', 'three', 'words']
>>> print(len(stuff))
3
>>> print(stuff[0])
With
Three
Words
>>>
```

**Split** breaks a string into parts and produces a list of strings. We think of these as words. We can **access** a particular word or **loop** through all the words.

**split()**, treats multiple **spaces** as a single space — kind of **smart**.

```
>>> line = 'A lot           of spaces'
>>> etc = line.split()
>>> print(etc)
['A', 'lot', 'of', 'spaces']
>>>
>>> line = 'first;second;third'
>>> thing = line.split()
>>> print(thing)
['first;second;third']
>>> print(len(thing))
1
>>> thing = line.split(';')
>>> print(thing)
['first', 'second', 'third']
>>> print(len(thing))
3
>>>
```

- When you do not specify a **delimiter**, multiple spaces are treated like one delimiter
- You can specify what **delimiter** character to use in the **splitting**.

### join(), List to String

#### CODE

```
animals = ['Cats', 'Dogs', 'Birds']
string1 = ''.join(animals)
print(string1)
```

#### OUTPUT

Cats Dogs Birds

### split(), Real world Example

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print(words[2])
```

Sat  
Fri  
Fri  
Fri  
...

```
>>> line = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> words = line.split()
>>> print(words)
['From', 'stephen.marquard@uct.ac.za', 'Sat', 'Jan', '5', '09:14:16', '2008']
```

## Chapter 9: Dictionaries

The Python dictionary is one of its most powerful data structures. Instead of representing values in a linear list, dictionaries store data as key / value pairs. Using key / value pairs gives us a simple in-memory "database" in a single Python variable.

### Comparing Lists and Dictionaries

Dictionaries are like lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd)
{'age': 21, 'course': 182}
>>> ddd['age'] = 23
>>> print(ddd)
{'age': 23, 'course': 182}
```

		List
Key	Value	
[0]	21	lst
[1]	183	
		Dictionary
Key	Value	
['course']	182	ddd
['age']	21	
['age']	182	

### Counting with Dictionaries

```
>>> ccc = dict()
>>> print(ccc['csev'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'csev'
>>> 'csev' in ccc
False
```

You can use the `in` operator to see if any key is in the dictionary. "A traceback is just a form of communication b/w you and Python"

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    if name not in counts:
        counts[name] = 1
    else:
        counts[name] = counts[name] + 1
print(counts)
```

→ [csev: 2, cwen: 2, zqian: 1]

csev	111
zqian	1111
cwen	1111
manquaid	111

Now, this notion of this if then else. If it's in, do one thing. If it's not, do another thing. It's so common. Thankfully there's a method `get`.

### The `get` Method for Dictionaries

The pattern of checking to see if a key is already in a dictionary and assuming a default value if the key is not there is so common that there is a method called `get()` that does this for us

```
if name in counts:
    x = counts[name]
else:
    x = 0
```

Default value if key does not exist (and no traceback).

```
(csev: 2, cwen: 2, zqian: 1)
```

It takes a key and a default value (two pars). `x` becomes either 0 or the key's current value, and **no traceback** occurs.

### Definite loops and Dictionaries

Even though dictionaries are not stored in order, we can write a `for` loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

```
>>> counts = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for key in counts:
...     print(key, counts[key])
...
chuck 1
fred 42
jan 100
```

```
jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
for aaa,bbb in jjj.items():
    print(aaa, bbb)
```

aaa	bbb
[chuck]	1
[fred]	42
[jan]	100

### Retrieving lists of Keys and Values

You can get a list of keys, values, or items (both) from a dictionary

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(list(jjj))
['chuck', 'fred', 'jan']
>>> print(list(jjj.keys()))
['chuck', 'fred', 'jan']
>>> print(list(jjj.values()))
[1, 42, 100]
>>> print(list(jjj.items()))
[('chuck', 1), ('fred', 42), ('jan', 100)]
```

### Bonus: Two Iteration Variables!

- We loop through the key-value pairs in a dictionary using \*two\* iteration variables
- Each iteration, the first variable is the key and the second variable is the corresponding value for the key

```
chuck 1
fred 42
jan 100
```

### Summary

- What is a collection?
- Lists versus Dictionaries.
- Dictionary constants.
- Using the `get()` method.
- Writing dictionary loops.

```
#Making a dictionary
dict = {'color': 'green', 'points': 5}
```

```
#Getting the value associated with a key
print(dict['color'])
print(dict['points'])
```

```
#Adding a key-value pair
dict['x'] = 25
dict['speed'] = 1.5
```

```
#Modifying values in a dictionary
dict['color'] = 'yellow'
dict['points'] = 10
```

```
#Deleting a key-value pair
del dict['points']
```

```
#Looping through all key-value pairs
for k, v in dict.items():
    print(k + ": " + v)
```

```
#Looping through all the keys
for key in dict.keys():
    print(key)
```

```
#Looping through all the keys in order
for key in sorted(dict.keys()):
    print(key + ": " + value)
```

```
#Finding a dictionary's length
num_responses = len(dict)
```

```
#Storing dictionaries in a list
list = []
list.append(dict)
```

```
#Storing lists in a dictionary
dict = {'color': ['green', 'red'], 'points': [5, 6]}
```

## PYTHON DICTIONARIES CHEATSHEET

## Chapter 10: Tuples

Tuples are like lists except we're going to use parentheses.

Tuples are another kind of sequence that functions much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')      >>> for iter in y:
>>> print(x[2])                         ...     print(iter)
Joseph                                         ...
>>> y = ( 1, 9, 2 )                      1
>>> print(y)                           9
(1, 9, 2)                                     2
>>> print(max(y))                     >>>
9
```

## Things not to do With Tuples

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```

### Tuples are more efficient

Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in the terms of memory use and performance than lists. In our program, when we're making "temporary vars" we prefer tuples over lists.

We can also put a tuple on the left-hand side of an assignment statement

We can even omit the parentheses

**Tuples & Assignment**

```
>>> (x, y) = (4, 'fred')
        /   \
        (x, 'y')  
>>> print(y)           if you do
        fred
        >>> (a, b) = (99, 98) (a, b) = 9
        >>> print(a)          Python ⊥ ≠ happy
        99
```

### Tuples are comparable

The comparison operators work with tuples & other sequences. If the first item is equal, Python goes to the next element & so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ('Jones', 'Sally') < ('Jones', 'Sam')
True
>>> ('Jones', 'Sally') > ('Adams', 'Sam')
True
```

### Using sorted()

We can do this even more directly using the built-in function `sorted` that takes a sequence as a parameter and returns a sorted sequence

```
>>> d = {'b':1, 'c':22, 'a':10}
>>> t = sorted(d.items())
>>> t
[(('a', 10), ('b', 1), ('c', 22))]
>>> for k, v in sorted(d.items()):
...     print(k, v)
...
a 10
b 1
c 22
```

IN ORDER

## Sort by Values Instead of Key

- If we could construct a list of tuples of the form (value, key) we could sort by value
- We do this with a for loop that creates a list of tuples

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> tmp = list()
>>> for k, v in c.items() :
...     tmp.append((v, k))
...
>>> print(tmp)
[(10, 'a'), (1, 'b'), (22, 'c')]
>>> tmp = sorted(tmp, reverse=True)
>>> print(tmp)
[(22, 'c'), (10, 'a'), (1, 'b')]
```

## Summary

- Tuple syntax
- Immutability
- Comparability
- Sorting
- Tuples in assignment statements
- Sorting dictionaries by either key or value