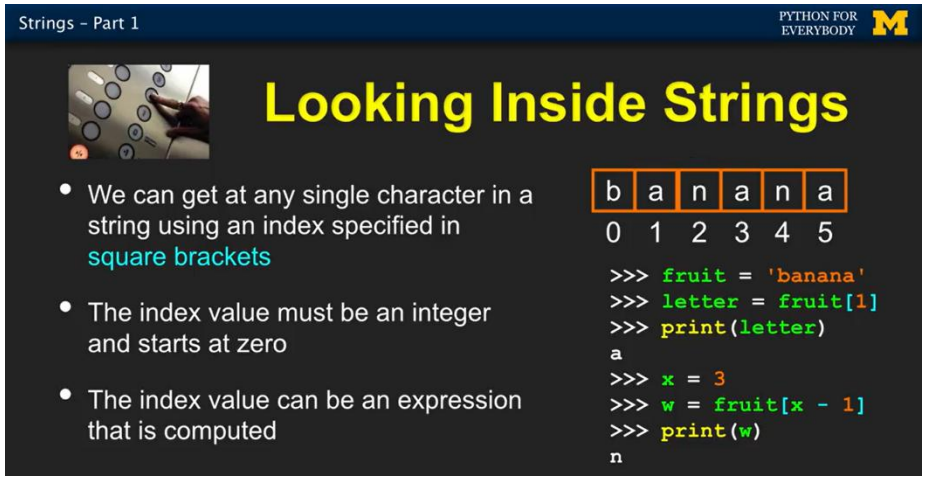
Course 2: Python Data Structures

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**



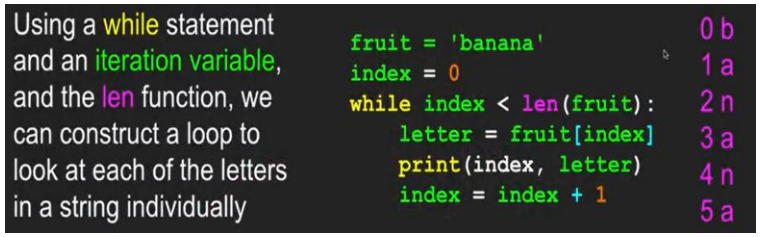
- You'll get a **an index error** if you attempt to index beyond the end of a string.

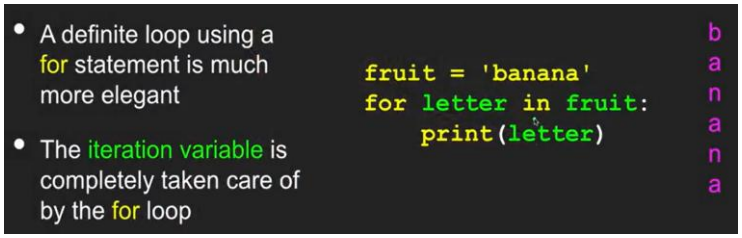
```
>>> print(fruit[6])
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

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

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

Looping through Strings





Slicing Strings

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

M o n t y P y t h o n
0 1 2 3 4 5 6 7 8 9 10 11

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

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

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

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

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

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

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

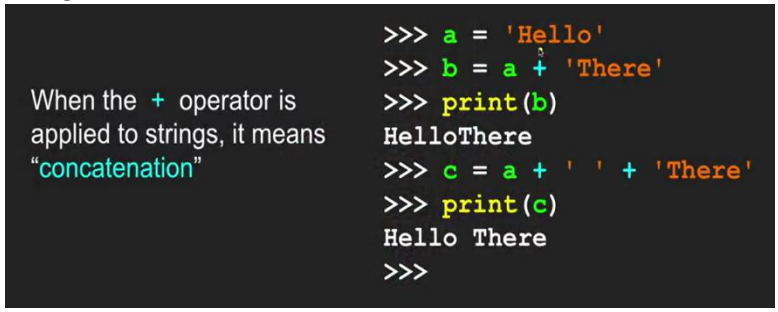
```
>>> print(s[8: ]
thon
```

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

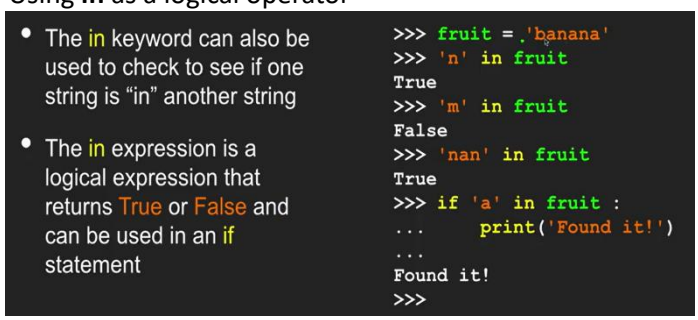
If we leave off the first number or the last of the slice, its assumed to be the **beginning or end** of the **string** respectively.

➢ **Part II**: **Manipulating Strings**

String Concatenation



Using **in** as a logical operator



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

```
pgsql

[ Permanent Storage ]
        |
        v
   +------------+
   | mbox.txt   |
   | line1\n    |
   | line2\n    |
   | line3\n    |
   +------------+
```

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

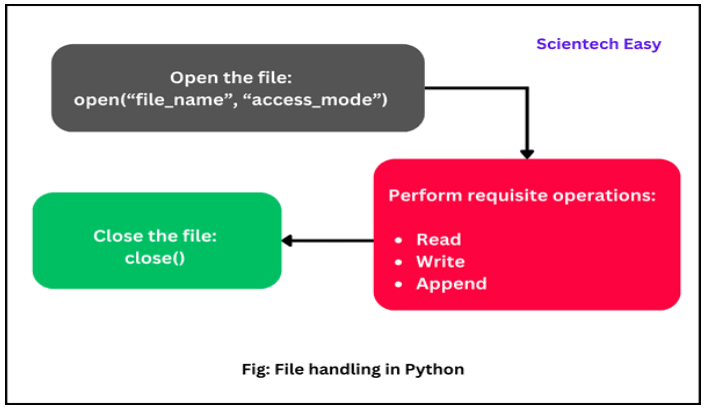
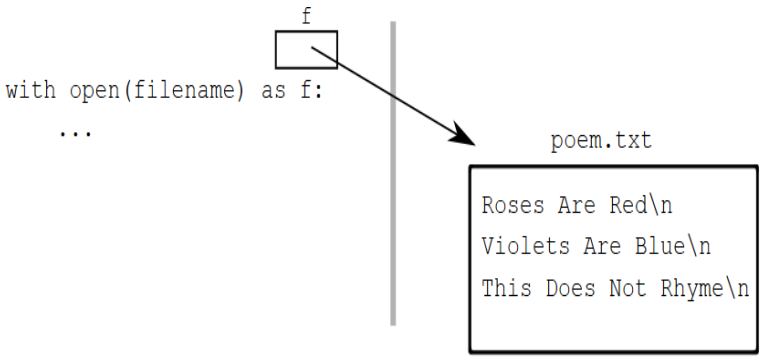


Fig: File handling in Python

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"]
```

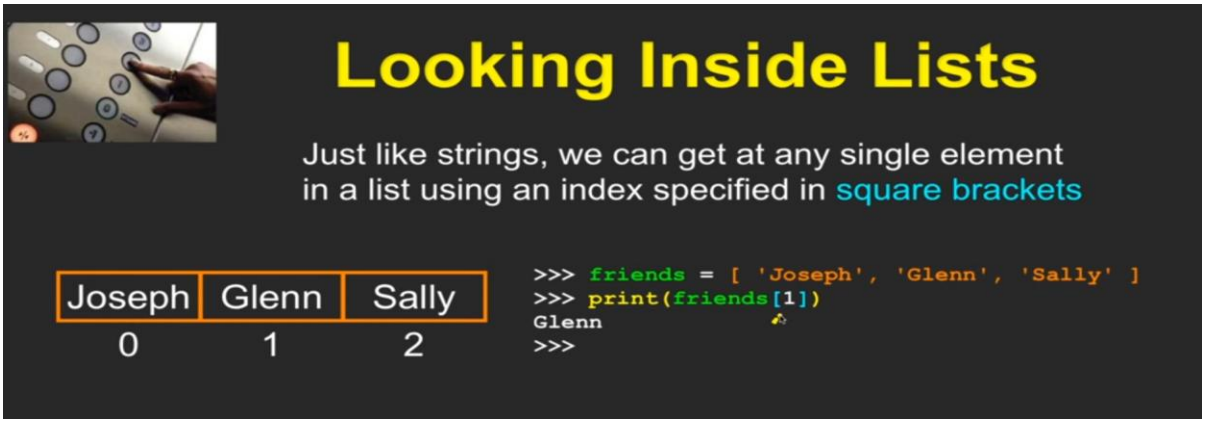
Lists can hold **any mix of types**:

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

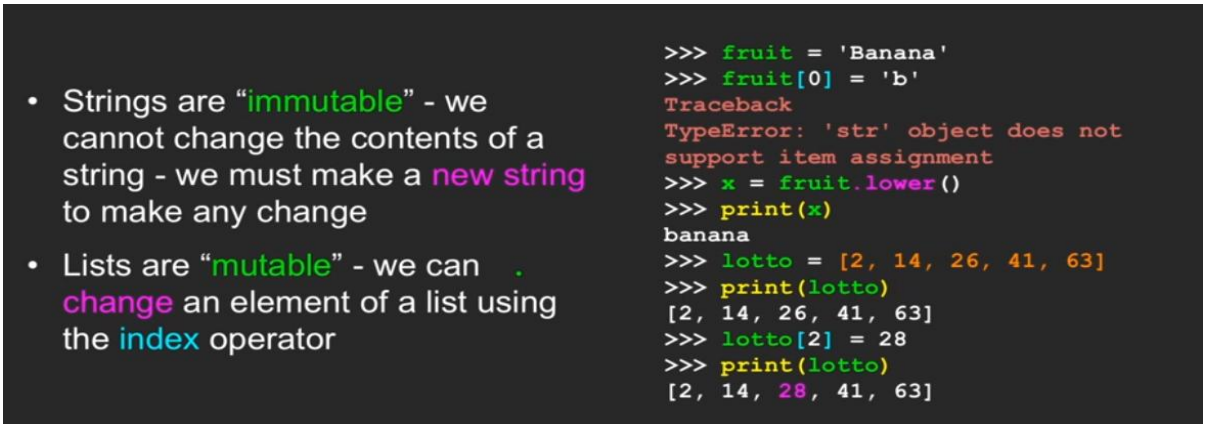
2. Lists & Definite Loops – Iterating through a list

```
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!
```



3. Mutability (you *can* change lists)



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])
```

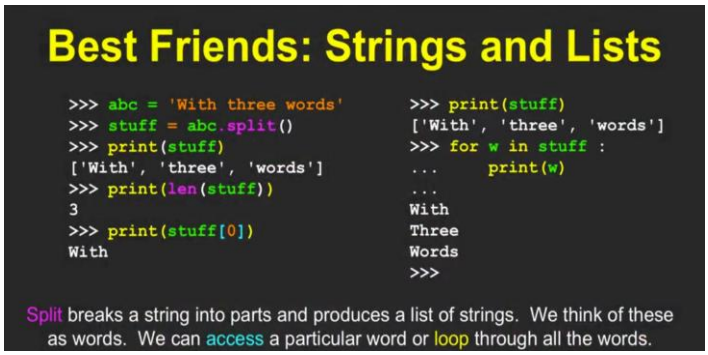
Equivalent to the simpler loop—just gives you the index.

Part II: Manipulating Lists

Operation	Example	Output / Notes
Concatenate lists	a=[1,2]; b=[3,4]; a+b	[1, 2, 3, 4]
Slicing (Works the **same way as strings**)	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

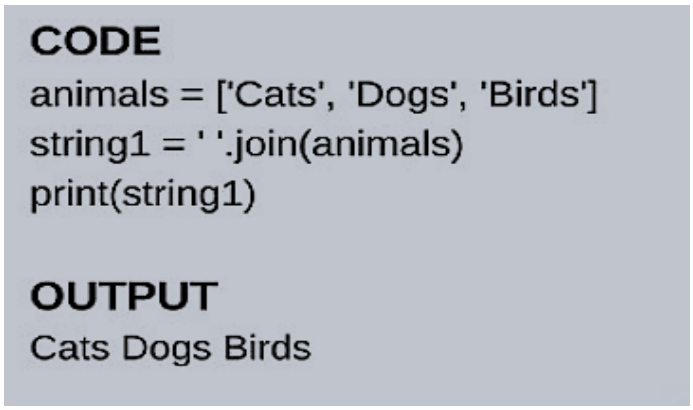
Check Python List Docs: See the official Python documentation for full list methods, behaviors, and examples.

Part III: Lists and Strings

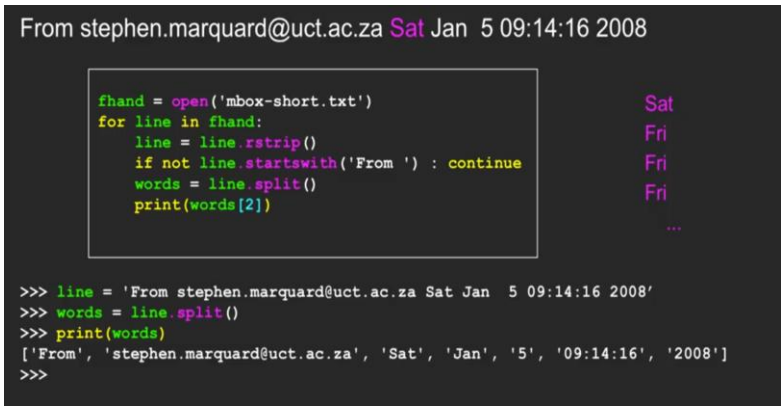


split(), treats multiple **spaces** as a single space — kind of **smart.**

join(), List to String

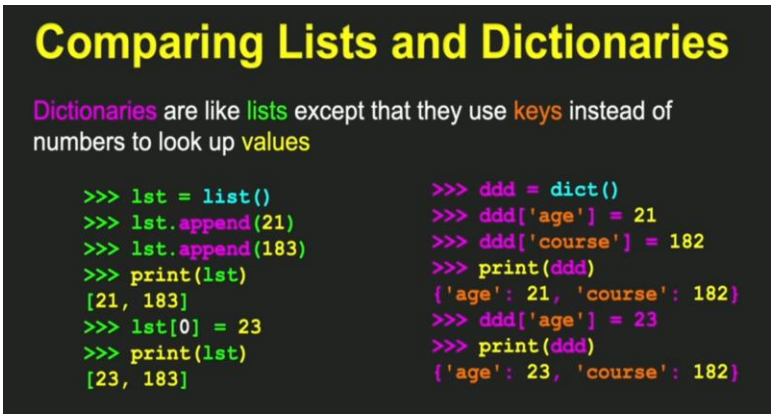


split(), Real world Example



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.

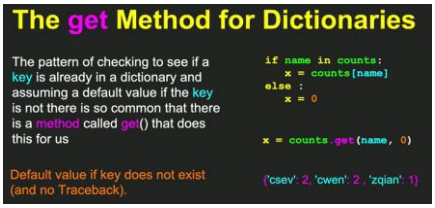


Counting with Dictionaries

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

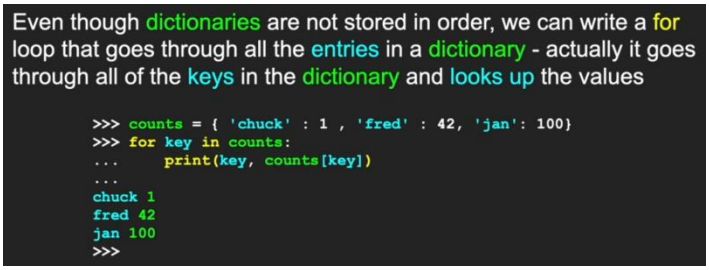
You can use the in operator to see if any key is in the dictionary. *"A traceback is just a form of communication b/w you and Python"*

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.

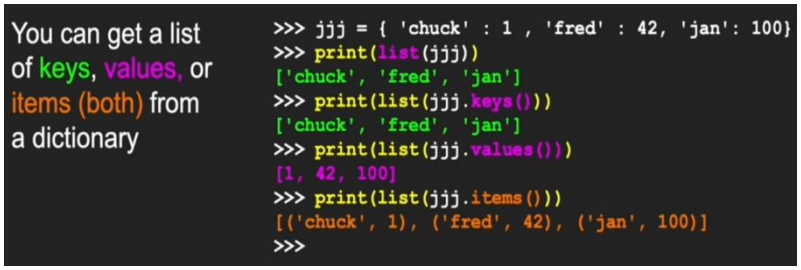


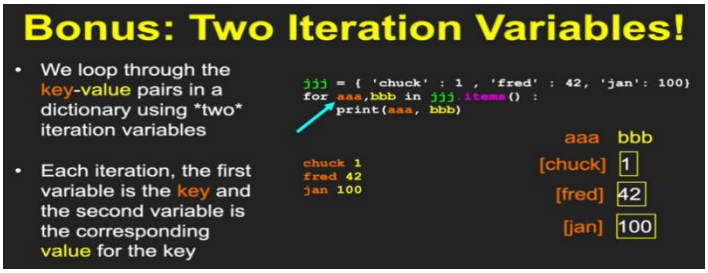
It takes a key and a default value (two pars). **x** becomes either 0 or the key's current value, and **no traceback** occurs.

Definite loops and Dictionaries



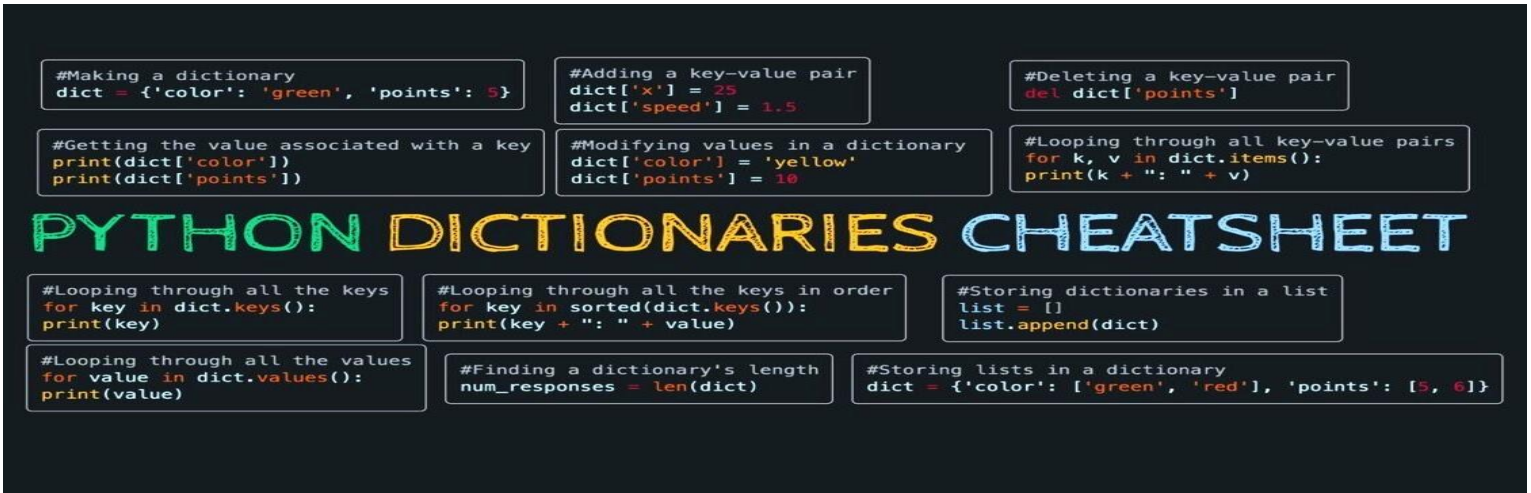
Retrieving lists of Keys an d Values





Summary

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



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')
>>> print(x[2])
Joseph
>>> y = ( 1, 9, 2 )
>>> print(y)
(1, 9, 2)
>>> print(max(y))
9
```

```
>>> for iter in y:
...     print(iter)
...
1
9
2
>>>
```

Tuples & Strings are IMMUTABLE

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
>>>[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] = 'D'
Traceback:'str'
object does
not support item
Assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
Assignment
>>>
```

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'
>>>
```

A Tale of Two Sequences

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

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')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```

if you do (a, b) = 9

Python 😐 ≠ happy

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
```

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