MODULE 2

Strings

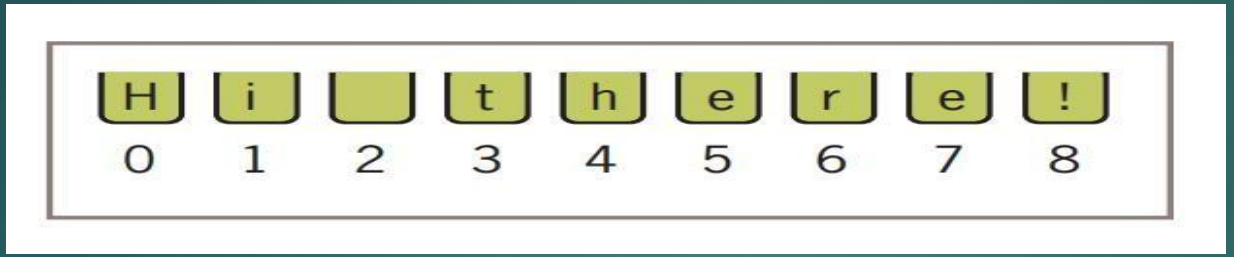
The Structure of Strings:

String is a data structure.
A string is a sequence of zero or more characters.
eg. "Hi there!"

A string's length is the number of characters it contains. Python's len function returns this value when it is passed a string.

```
>>> len("Hi there!")
9
>>> len("")
0
```

The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right.



The string is an immutable data structure. This means that its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.

The Subscript Operator:

a simple for loop can access any of the characters in a string, sometimes you just want to inspect one character at a given position without visiting them all. The subscript operator [] makes this possible.

<a string>[<an integer expression>]

The integer expression is also called an index.

```
>>> name = "Alan Turing"
>>> name[0]
'A'
>>> name[3]
'n'
>>> name[len(name)]
        Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        IndexError: string index out of range
>>> name[len(name) - 1]
'g'
>>> name[-l]
'g'
>>> name[-2]
'n'
```

The next code segment uses a count-controlled loop to display the characters and their positions:

```
>>> data = "apple"
>>> for index in range(len(data)):
 print(index, data[index])

1 a
2 p
3 p
4 l
5 e
```

Slicing for Substrings

You can use Python's subscript operator to obtain a substring through a process called slicing. To extract a substring, the programmer places a colon (:) in the subscript.

```
>>> name = "myfile.txt"    # The entire string
>>> name[0 : ]
'myfile.txt'
>>> name[0 : 1]             # The first character
'm'
>>> name[ 0 : 2]            # The first two characters
'my'
>>> name[ : len(name)]      # The entire string
'myfile.txt'
>>> name[-3 : ]             # The last three characters
'txt'
>>> name[ 2 : 6]            # Drill to extract 'file'
'file'
```

Testing for a Substring with the in Operator

Python's in operator : We can search for a substring or a character using this operator.

- ✓When used with strings, the left operand of in is a target substring, and the right operand is the string to be searched.
- ✓The operator in returns True if the target string is somewhere in the search string, or False otherwise.

eg.,

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
>>> for fileName in fileList:
        if ".txt" in fileName:
        print(fileName)
```

o/p - myfile.txt
yourfile.txt

Exercises

1.Assume that the variable data refers to the string "myprogram.exe". Write the values of the following expressions:

a. data[2]
b. data[-1]
c. len(data)
d. data[0:8]

2.Assume that the variable data refers to the string "myprogram.exe". Write the expressions that perform the following tasks:

a. Extract the substring "gram" from data.
b. Truncate the extension ".exe" from data.
c. Extract the character at the middle position from data.

3.Assume that the variable myString refers to a string. Write a code segment that uses a loop to print the characters of the string in reverse order.

4.Assume that the variable myString refers to a string, and the variable reversedString refers to an empty string. Write a loop that adds the characters from myString to reversedString in reverse order.

String Methods

- Python includes a set of string operations called methods.
- A method behaves like a function but has a slightly different syntax.
- Unlike a function, a method is always called with a given data value called an object, which is placed before the method name in the call.

  <an object>.<method name>(<argument-1>,..., <argument-n>)

Methods can also expect arguments and return values.

A method knows about the internal state of the object with which it is called.

String Method	What it Does
`s.center(width)`	Returns a copy of **s** centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring **sub** in **s**. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.endswith(sub)`	Returns **True** if **s** ends with **sub** or **False** otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in **s** where substring **sub** is found. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.isalpha()`	Returns **True** if **s** contains only letters or **False** otherwise.
`s.isdigit()`	Returns **True** if **s** contains only digits or **False** otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is **s**.

`s.lower()`	Returns a copy of **s** converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of **s** with all occurrences of substring **old** replaced by **new**. If the optional argument **count** is given, only the first **count** occurrences are replaced.
`s.split([sep])`	Returns a list of the words in **s**, using **sep** as the delimiter string. If **sep** is not specified, any whitespace string is a separator.
`s. startswith(sub)`	Returns **True** if **s** starts with **sub** or **False** otherwise.
`s.strip([aString])`	Returns a copy of **s** with leading and trailing whitespace (tabs, spaces, newlines) removed. If **aString** is given, remove characters in **aString** instead.
`s.upper()`	Returns a copy of **s** converted to uppercase.

some string methods in action:

```
>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e') 2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find("the")
3
```

```
>>> s.isalpha()
False

>>> 'abc'.isalpha()
True

>>> "326".isdigit()
True

>>> words = s.split()
>>> words
['Hi', 'there!']

>>> " ".join(words)
'Hithere!'
```

```
>>> " ". join(words)
'Hi there!'

>>> s.lower()
'hi there!'

>>> s.upper()
'HI THERE!'

>>> s.replace('i', 'o')
'Ho there!'
```

- Extracting a filename's extension using split() method
- >>> "myfile.txt". split('.')
  ['myfile', 'txt']
- >>> "myfile.py". split('.')
  ['myfile', 'py']
- >>> "myfile.html". split('.')
  ['myfile', 'html']

To write a general expression for obtaining any filename's extension, as follows:

```
filename.split('.')[-1]
```

```
>>> filename="myfile.txt"
>>> filename.split('.')[-1]
'txt'
```

- Exercises

1. Assume that the variable data refers to the string "Python rules!". Use a string method from perform the following tasks:

    a. Obtain a list of the words in the string.

    b. Convert the string to uppercase.

    c. Locate the position of the string "rules".

    d. Replace the exclamation point with a question mark.

- 2. Using the value of data from Exercise 1, write the values of the following expressions:

    a. data.endswith('i')

    b. " totally ". join(data.split())

Number System

The value of each digit in a number can be determined using –

- ✓ The digit
- ✓ The position of the digit in the number
- ✓ The base of the number system (where the base is defined as the total number of digits available in the number system)

1) decimal number system - base 10 number system
   0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits
2) binary number system - base 2 number system
   binary 0 and 1
3) octal number system - base 8 number system
   0, 1, 2, 3, 4, 5, 6, and 7
4) hexadecimal number system - base 16 number system
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

To identify the system being used, you attach the base as a subscript to the number.

415 in binary notation 110011111_2
415 in octal notation 637_8
415 in decimal notation 415_{10}
415 in hexadecimal notation $19F_{16}$

✓The digits used in each system are counted from 0 to n – 1, where n is the system's base.
✓Thus, the digits 8 and 9 do not appear in the octal system.

✓To represent digits with values larger than 9_{10}, systems such as base 16 use letters. Thus, A_{16} represents the quantity 10_{10}, whereas 10_{16} represents the quantity 16_{10}

The Positional System for Representing Numbers

positional notation—that is, the value of each digit in a number is determined by the digit's position in the number. In other words, each digit has a positional value.

How to find positional value of a digit in a number?

The positional value of a digit is determined by raising the base of the system to the power specified by the position . $(base^{position})$
For an n-digit number, the positions are numbered from n – 1 down to 0, starting with the leftmost digit and moving to the right.

eg., the positional values of the three-digit number 415_{10} are:
$100\ (10)^2$, $10\ (10)^1$, and $1\ (10)^0$, moving from left to right in the number.

To determine the quantity represented by a number in any system from base 2 through base 10, you multiply each digit (as a decimal number) by its positional value and add the results.

The following example shows how this is done for a three-digit number in base 10:

$$415_{10} =$$

$$4 * 10^2 + 1 * 10^1 + 5 * 10^0 =$$

$$4 * 100 + 1 * 10 + 5 * 1 =$$

$$400 + 10 + 5 = 415$$

Positional values	100	10	1
Positions	2	1	0

Converting Binary to Decimal

conversion process : Multiply the value of each bit (0 or 1) by its positional value and add the results.

$$1100111_2 =$$

$$1*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 =$$

$$1*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 1*1 =$$

$$64 + 32 + 4 + 2 + 1 = 103$$

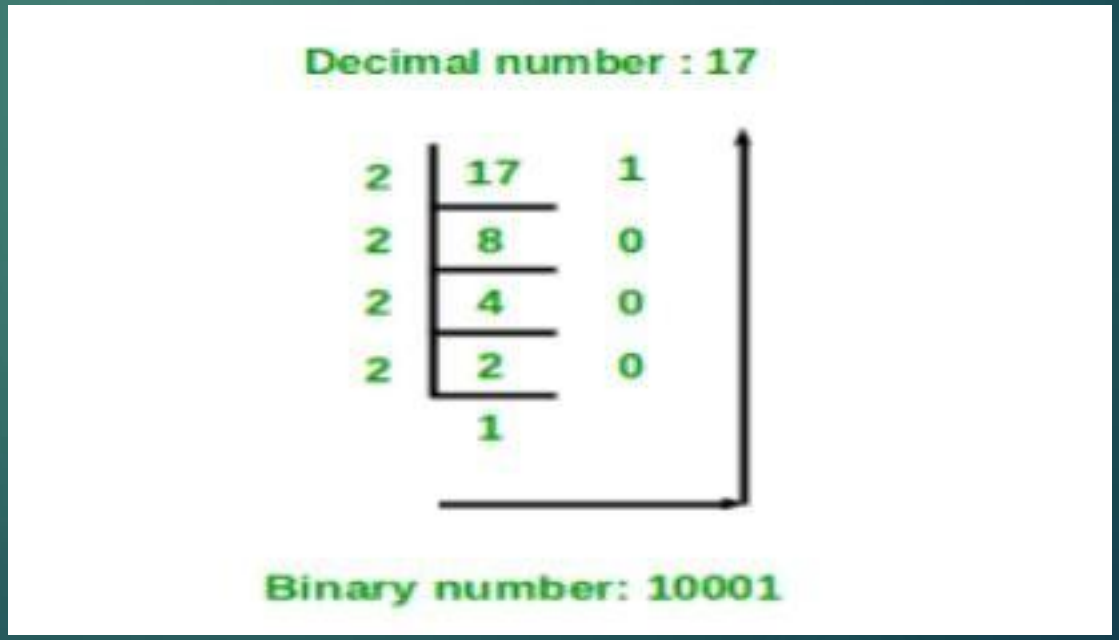
Converts a string of bits to a decimal integer.

```
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

```
Enter a string of bits: 1111
The integer value is 15
```

Converting Decimal to Binary

1) This algorithm repeatedly divides the decimal number by 2.
2) After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits.
3) The quotient becomes the next dividend in the process.
4) The string of bits is initially empty, and the process continues while the decimal number is greater than 0.



```
decimal = int(input("Enter a decimal number: "))
if decimal == 0:
    print(0)
else:
    bitString = ""
while decimal > 0:
    remainder = decimal % 2
    decimal = decimal // 2
    bitString = str(remainder) + bitString

print("The binary representation is", bitString)
```

```
Enter a decimal number: 156
The binary representation is 00111001
```

Conversion Shortcuts

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

- ✓Note the rows that contain exact powers of 2 (2, 4, and 8 in decimal).
- ✓Each of the corresponding binary numbers in that row contains a 1 followed by a number of zeroes that equal the exponent used to compute that power of 2.

- ✓Thus, a quick way to compute the decimal value of the number 10000_2 is 2^4 or 16_{10}.

- ✓ The rows whose binary numbers contain all 1s correspond to decimal numbers that are one less than the next exact power of 2.

- ✓For example, the number 111_2 equals 2^3-1 , or 7_{10}.
  Thus, a quick way to compute the decimal value of the number 11111_2 is 2^5 -1 or 31_{10}.

Octal Number System

- It requires only 3 bits to represent value of any digit.
- Octal numbers are indicated by the addition of either an 0o prefix or an 8 subscript.
- Position of every digit has a weight which is a power of 8.
- Numeric value of an octal number is determined by multiplying each digit of the number by its positional value and then adding the products.
- The main advantage of using Octal numbers is that it uses less digits than decimal and Hexadecimal number system. So, it has fewer computations and less computational errors.

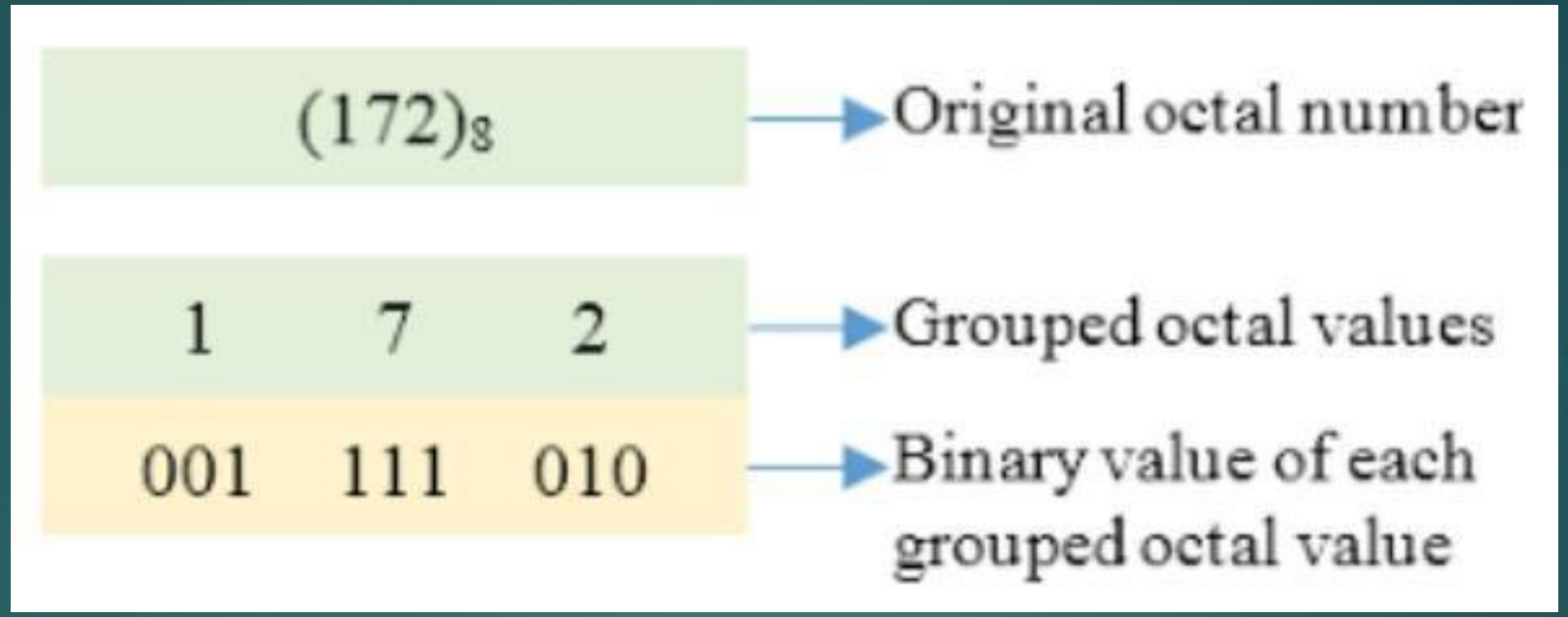
Conversions

Octal Digit Value	Binary Equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Binary to Octal Conversion

Binary Digit Value	001101010111001111
Group the bits into three´s starting from the right hand side	001 101 010 111 001 111
Octal Number form	15271_7

Octal to Binary Conversion



Octal to Decimal Conversion

Octal Digit Value	2322_8
In polynomial form	$=(2\times8^3)+(3\times8^2)+(2\times8^1)+(2\times8^0)$
Add the results	$=(1024)+(192)+(16)+(2)$
Decimal number form equals: 1234_{10}	

Decimal to Octal Conversion

Binary to Octal Conversion

```
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
 decimal = decimal + int(digit) * 2 ** exponent
 exponent = exponent - 1

i=1
octal=0
while decimal != 0:
    octal += int(decimal % 8)*i
    decimal /= 8
    i *= 10
print("octal number is: ",octal)
```

Octal to Binary Conversion

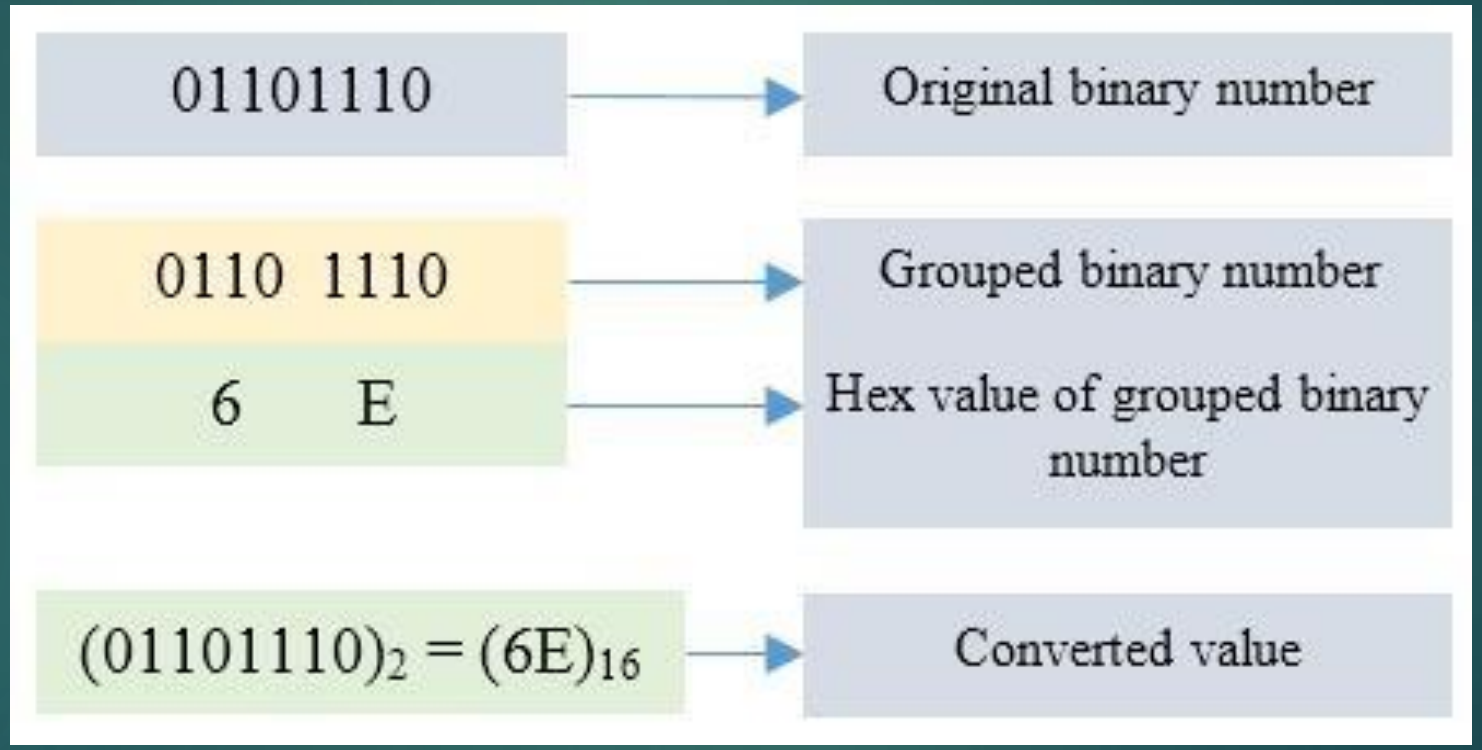
```
oc = int(input("Enter the octal number: "))
dec = 0
i = 0
while oc != 0:
    dec = dec + (oc % 10) * pow(8,i)
    oc = oc // 10
    i = i+1

bi = ""
while dec != 0:
    rem = dec % 2
    dec = dec // 2
    bi = str(rem) + bi
print("binary number is:", bi)
```

Hexadecimal Number System

- It requires only 4 bits to represent value of any digit.
- Hexadecimal numbers are indicated by the addition of either an 0x prefix or an 16 as subscript.
- Position of every digit has a weight which is a power of 16.
- Numeric value of a hexadecimal number is determined by multiplying each digit of the number by its positional value and then adding the products.
- So, it is also a positional (or weighted) number system.

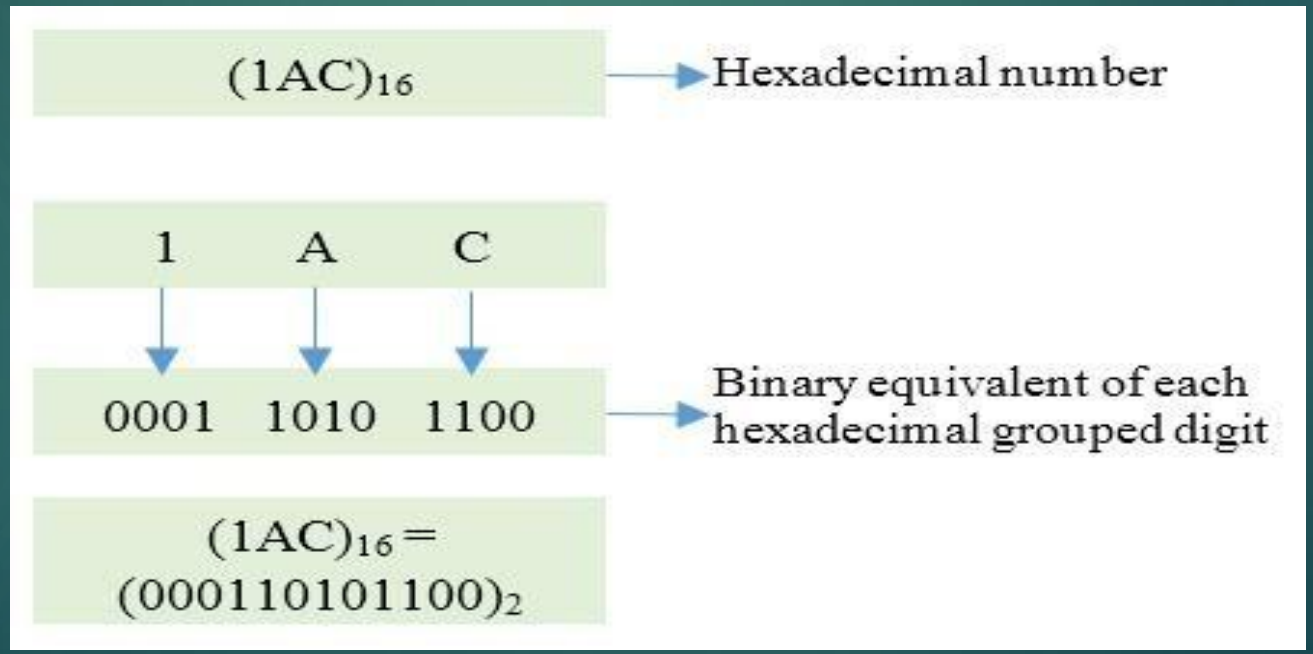
Binary to Hexadecimal Conversion

Factor the bits into groups of four and look up the corresponding hex digits.



Hexadecimal to Binary Conversion

- Each digit in the hexadecimal number is equivalent to four digits in the binary number.
- Thus, to convert from hexadecimal to binary, you replace each hexadecimal digit with the corresponding 4-bit binary number.



List Literals and Basic Operators

- literal string values are written as sequences of characters enclosed in quote marks.
- In Python, a list literal is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets ([and]).

Eg.,

```
[1951, 1969, 1984]                    # A list of integers
["apples", "oranges", "cherries"]     # A list of strings
[]                                    # An empty list
```

You can also use other lists as elements in a list, thereby creating a list of lists.

[[5, 9], [541, 78]]

Construction of two lists and their assignment to variables:

```
>>> first = [1, 2, 3, 4]
>>> second = list(range(1, 5))
>>> first
[1, 2, 3, 4]
>>> second
[1, 2, 3, 4]
```

The list function can build a list from any iterable sequence of elements, such as a string:

```
>>> third = list("Hi there!")
>>> third
['H', 'i', ' ', 't', 'h', 'e', 'r', 'e', '!']
```

The function len and the subscript operator [] work just as they do for strings:

```
>>> len(first)
4
>>> first[0]
1
>>> first[2:4]
[3, 4]
```

Concatenation (+) and equality (==) also work as expected for lists:

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
```

- To print the contents of a list without the brackets and commas, you can use a for loop, as follows:

```
>>> for number in [1, 2, 3, 4]:
print(number, end = " ")
1 2 3 4
```

- Finally, you can use the in operator to detect the presence or absence of a given element:

```
>>> 3 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
```

Operator or Function	What It Does
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L1 + L2`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range **0** through **`upper - 1`**.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns **`True`** if all the results are true, or **`False`** otherwise.
`for <variable> in L: <statement>`	Iterates through the list, binding the variable to each element.
`<any value> in L`	Returns **`True`** if the value is in the list or **`False`** otherwise.

Replacing an Element in a List

There is one huge difference between String and List.

ie., a string is immutable, its structure and contents cannot be changed. But a list is changeable—that is, it is mutable.

At any point in a list's lifetime, elements can be inserted, removed, or replaced. The list itself maintains its identity but its internal state—its length and its contents—can change.

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
```

- How to replace each number in a list with its square:

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> for index in range(len(numbers)):
 numbers[index] = numbers[index] ** 2
>>> numbers
[4, 9, 16, 25]
```

- This session uses the string method split to extract a list of the words in a sentence. These words are then converted to uppercase letters within the list:

```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> for index in range(len(words)):
        words[index] = words[index].upper()
>>> words
 ['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
```

List Methods for Inserting and Removing Elements

List Method	What It Does
`L.append(element)`	Adds **element** to the end of **L**.
`L.extend(aList)`	Adds the elements of **aList** to the end of **L**.
`L.insert(index, element)`	Inserts **element** at **index** if **index** is less than the length of **L**. Otherwise, inserts **element** at the end of **L**.
`L.pop()`	Removes and returns the element at the end of **L**.
`L.pop(index)`	Removes and returns the element at **index**.

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
```

```
>>> example = [1, 2]
>>> example
[1, 2]

>>> example.append(3)
>>> example
[1, 2, 3]

>>> example.extend([11, 12, 13])
>>> example
[1, 2, 3, 11, 12, 13]

>>> example + [14, 15]
[1, 2, 3, 11, 12, 13, 14, 15]

>>> example
[1, 2, 3, 11, 12, 13]
```

Searching a List

- After elements have been added to a list, a program can search for a given element.
- The in operator determines an element's presence or absence, but programmers often are more interested in the position of an element if it is found.
- Instead of find, you must use the method index to locate an element's position in a list. It is unfortunate that index raises an exception when the target element is not found.
- To guard against this unpleasant consequence, you must first use the in operator to test for presence and then the index method if this test returns True.

```
aList = [34, 45, 67]
target = 45
if target in aList:
print(aList.index(target))
else:
 print(-1)
```

Sorting a List

- you can arrange some elements in numeric or alphabetical order.
- A list of numbers in ascending order and a list of names in alphabetical order are sorted lists.
- When the elements can be related by comparing them for less than and greater than as well as equality, they can be sorted.
- The list method sort mutates a list by arranging its elements in ascending order.

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

Mutator Methods and the Value None

Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called mutators. Examples are the list methods insert, append, extend, pop, and sort.

Python nevertheless automatically returns the special value None even when a method does not explicitly return a value.

Eg.,

>>> aList = aList.sort()

Unfortunately, after the list object is sorted, this assignment has the result of setting the variable aList to the value None.

>>> print(aList)

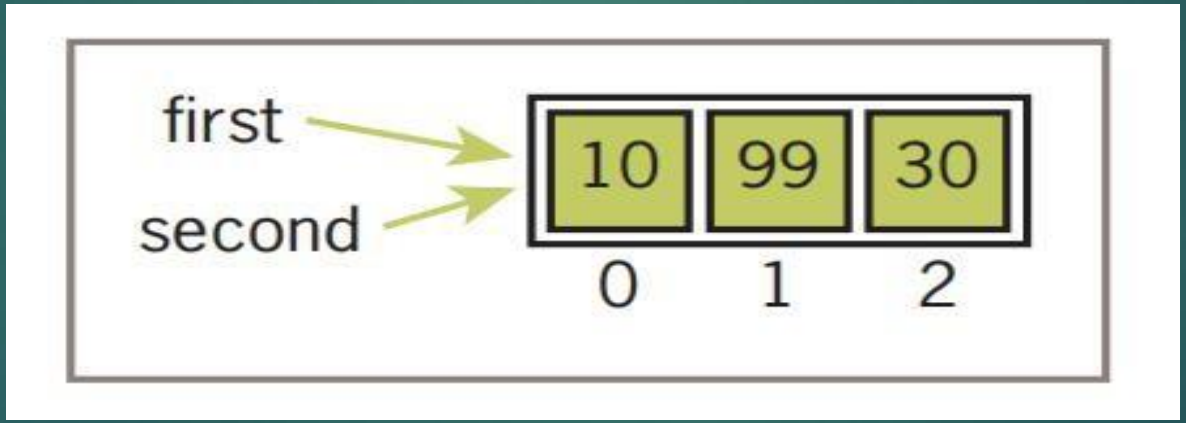
None

Aliasing and Side Effects

The mutable property of lists leads to some interesting phenomena, as shown in the following example:

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
 [10, 99, 30]
>>> second
 [10, 99, 30]
```

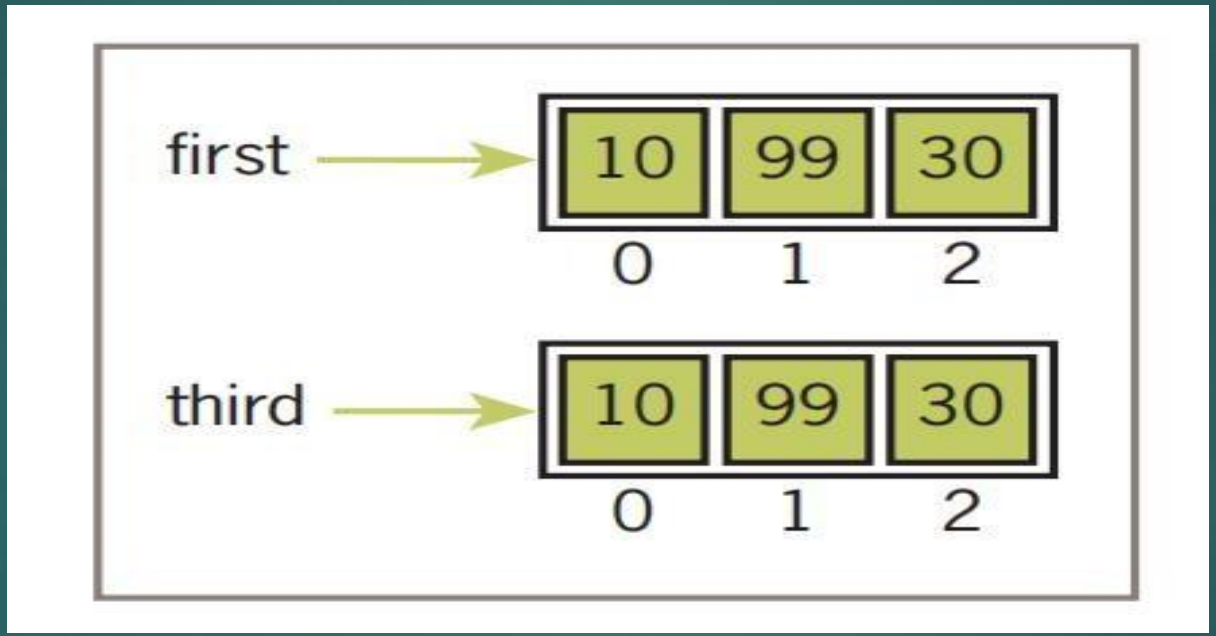
In this example, a single list object is created and modified using the subscript operator. When the second element of the list named first is replaced, the second element of the list named second is replaced also. This type of change is what is known as a side effect. This happens because after the assignment second = first, the variables first and second refer to the exact same list object. They are aliases for the same object. This phenomenon is known as aliasing.



To prevent aliasing, you can create a new object and copy the contents of the original to it

```
>>> third = []
>>> for element in first:
 third.append(element)
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```

The variables first and third refer to two different list objects, although their contents are initially the same, The important point is that they are not aliases, so you don't have to be concerned about side effects.



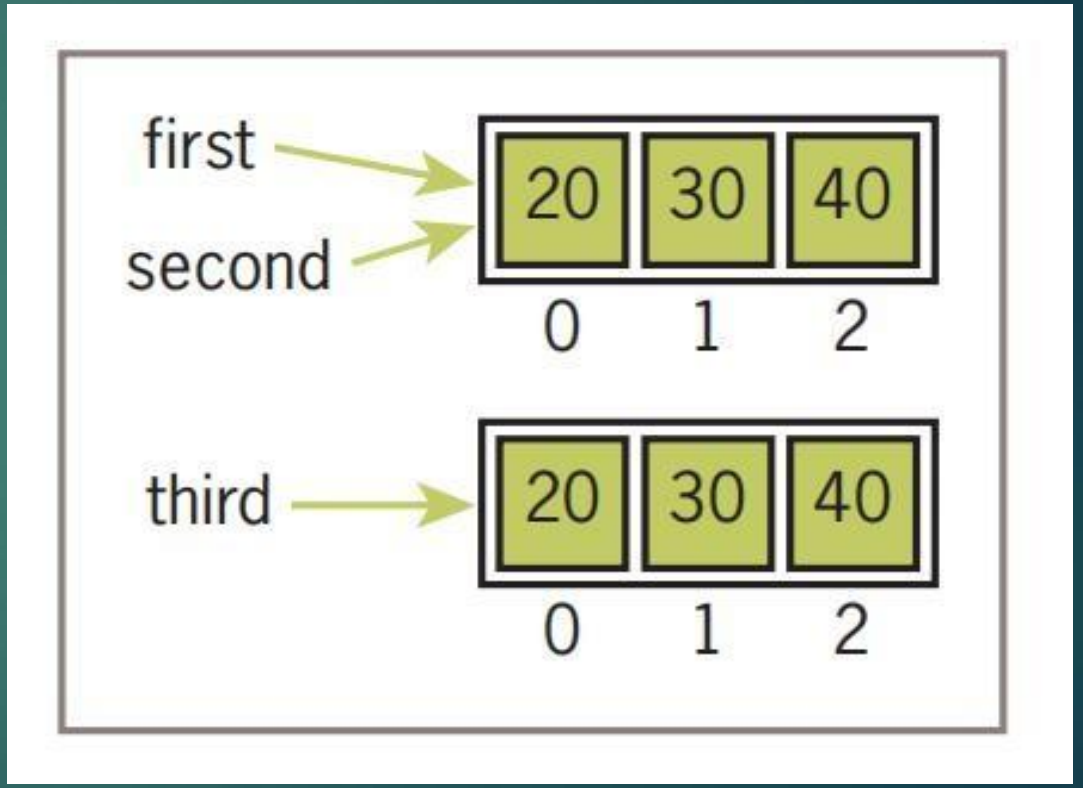
Equality: Object Identity and Structural Equivalence

- If you might want to determine whether one variable is an alias for another. The == operator returns True if the variables are aliases for the same object.
- Unfortunately, == also returns True if the contents of two different objects are the same.
- The first relation is called object identity, whereas the second relation is called structural equivalence.
- The == operator has no way of distinguishing between these two types of relations.

The solution to this problem :

Python's is operator can be used to test for object identity. It returns True if the two operands refer to the exact same object, and it returns False if the operands refer to distinct objects (even if they are structurally equivalent).

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first) # Or first[:]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



Using a List to Find the Median of a Set of Numbers

- If the number of values in a list is odd, the median of the list is the value at the midpoint when the set of numbers is sorted; otherwise, the median is the average of the two values surrounding the midpoint.

- Thus, the median of the list [1, 3, 3, 5, 7] is 3, and the median of the list [1, 2, 4, 4] is also 3.

```
fileName = input("Enter the filename: ") f = open(fileName, 'r')
numbers = []
for line in f:
 words = line.split()
 for word in words:
 numbers.append(float(word))
numbers.sort()
midpoint = len(numbers) // 2
print("The median is", end = " ")
if len(numbers) % 2 == 1:
print(numbers[midpoint])
else:
 print((numbers[midpoint] + numbers[midpoint - 1]) / 2)
```

This script inputs a set of numbers from a text file and print their median

List Comprehension

List comprehension is an elegant way to define and create lists based on existing lists.

List Comprehension vs For Loop in Python

Suppose, we want to separate the letters of the word “person” and add the letters as items of a list. The first thing that comes in mind would be using for loop.

```
>>> letters=[]
>>> for letter in "person":
        letters.append(letter)
>>> print(letters)
['p', 'e', 'r', 's', 'o', 'n']
```

- Iterating through a string Using List Comprehension

```
letters = [ letter for letter in 'person' ]
print( letters)
```

```
['p', 'e', 'r', 's', 'o', 'n']
```

- *Syntax for List Comprehension*

```
[expression for item in list]
```

Some Examples for List Comprehension:

```
>>> #compute the square of numbers upto 10
>>> sq = [ i*i for i in range(11)]
>>> print(sq)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> #create a vowels list from a string
>>> v = [ i for i in "everyone" if i in "aeiou"]
>>> print(v)
['e', 'e', 'o', 'e']
```

Coding Exercises:

1. Write a program to print the transpose of a matrix.

2. Write a program to add two matrices.