ILP 2024 : Computing

Week 4
23 May 2024

Objectives

- String
- Tuple
- Sets
- Dictionary
- Algorithm Complexity
- Algorithm Optimization

Strings in Python

- Python strings are sequences of characters, represented using single quotes ('), double quotes ("") or triple quotes (''' or """).
  - Example:

```
str1 = 'Hello, world!'
str2 = "Python Programming"
str3 = '''This is a multi-line
          string'''
```

String Operations

- Characters in a string can be accessed using indexing.
- Indexing starts from 0 for the first character.
- Negative indexing can be used to access characters from the end of the string.
- Example:

```
#String in Pyhton

str = "Python"
print(str[0])    # Output: P
print(str[-1])   # Output: n
```

```
P
n
```

Python provides several built-in methods for string manipulation:

`upper()`: Convert a string to uppercase.

`lower()`: Convert a string to lowercase.

`strip()`: Remove leading and trailing whitespace.

`split()`: Split a string into a list of substrings.

`join()`: Concatenate strings from an iterable using a delimiter.

`find()`: Find the index of a substring within a string.

`replace()`: Replace occurrences of a substring with another substring.

Examples

```
str = "   Python Programming   "
print(str.strip())
print(str.lower())
print(str.split())
print(str.replace("Python", "Java"))
```

```
Python Programming
   python programming
['Python', 'Programming']
   Java Programming
```


Use the `format()` method to insert numbers into strings:

```
age = 23
txt = "My name is John, and I am {}"
print(txt.format(age))
```

```
My name is John, and I am 23
```

Contd…

Because **strings** are roughly the same things as **lists**, they can be traversed using a `for` loop,

- Index-wise,
- Index- and element-wise with enumerate.
- (Zip, etc. also works!)


```
str1 = "Python"
str2 = "Programming"

print(str1[:5])

print(str1 + str2)

print(str1 + " " + str2)

for index, character in enumerate(str1):
    print(index, character)
```

```
Pytho
PythonProgramming
Python Programming
0 P
1 y
2 t
3 h
4 o
5 n
```

Contd…

You can check for characters types in a string with 3 methods.

`isalpha()` returns True is characters are letters only; and False otherwise.

`isdigit()` returns True if the characters are digits only; and False otherwise.

`isalnum()` returns True if the characters are letters and digits only; and False otherwise.


```python
def check_character_types(string):
    # Check if all characters are letters
    if string.isalpha():
        print("All characters in the string are letters.")
    else:
        print("The string contains non-letter characters.")

    # Check if all characters are digits
    if string.isdigit():
        print("All characters in the string are digits.")
    else:
        print("The string contains non-digit characters.")

    # Check if all characters are alphanumeric (letters or digits)
    if string.isalnum():
        print("All characters in the string are alphanumeric.")
    else:
        print("The string contains non-alphanumeric characters.")

# Test the function with a sample string
sample_string = "Hello123"
check_character_types(sample_string)

print("\n")
sample_string = "AnotherTestCase"
check_character_types(sample_string)
```

```
The string contains non-letter characters.
The string contains non-digit characters.
All characters in the string are alphanumeric.


All characters in the string are letters.
The string contains non-digit characters.
All characters in the string are alphanumeric.
```

Tuple

- Tuple is built-in data type in Python used to store collections of data
- A tuple is a collection which is **ordered** and **unchangeable** (Immutable)
- Elements cannot be changed, added, or removed after they are created.
  - Example: Storing coordinates (latitude, longitude) for geographic locations. Once defined, these coordinates remain fixed, ensuring data consistency and integrity throughout the program execution.

```
my_tuple = (element1, element2, …)
```



```
# Tuple in Python

# Creating a tuple
my_tuple = (1, 2, 3, 4, 5)

print(type(my_tuple),"\n")

# Accessing tuple elements
print("First element:", my_tuple[0])
print("Second element:", my_tuple[1])
print("Last element:", my_tuple[-1])

print("Slicing example:", my_tuple[2:5])

my_tuple.append(6)
```

```
<class 'tuple'>

First element: 1
Second element: 2
Last element: 5
Slicing example: (3, 4, 5)
```

```
---------------------------------------------------------------------
AttributeError                            Traceback (most recen
Cell In[5], line 15
     11 print("Last element:", my_tuple[-1])
     13 print("Slicing example:", my_tuple[2:5])
---> 15 my_tuple.append(6)

AttributeError: 'tuple' object has no attribute 'append'
```

Contd …

- In some cases, we may need to update the tuple value.
- Even though tuple is unchangeable, we can apply trick to update it

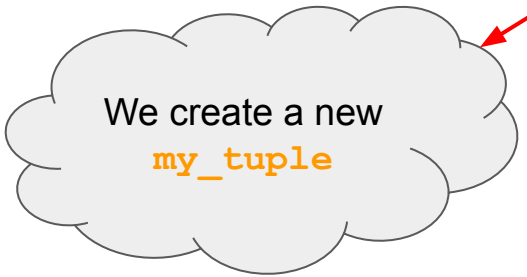
```
#Updating Tuple

# Original tuple
my_tuple = (1, 2, 3)


# Desired update: Change the second element to 4
my_tuple = my_tuple[:2] + (4,) + my_tuple[2:]

print(" Tuple:", my_tuple)
```

```
 Tuple: (1, 2, 4, 3)
```

Unpacking variables

Unpacking: In Python, we can extract the values from a tuple into variables.

```
#Unpacking a tuple:

fruits = ("apple", "banana", "cherry")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
apple
banana
cherry
```

Note: size of the tuple and number of variables should be same during unpacking

```
#Unpacking a tuple:

fruits = ("apple", "banana", "cherry", "mango")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most
Cell In[19], line 5
      1 #Unpacking a tuple:
      3 fruits = ("apple", "banana", "cherry", "mango")
----> 5 (green, yellow, red) = fruits
      7 print(green)
      8 print(yellow)

ValueError: too many values to unpack (expected 3)
```

```
#Unpacking a tuple:

fruits = ("apple", "banana", "cherry")

(green, yellow, red, black) = fruits

print(green)
print(yellow)
print(red)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most rec
Cell In[20], line 5
      1 #Unpacking a tuple:
      3 fruits = ("apple", "banana", "cherry")
----> 5 (green, yellow, red, black) = fruits
      7 print(green)
      8 print(yellow)

ValueError: not enough values to unpack (expected 4, got 3)
```

Contd…

If the number of variables is less than the number of values, you can add an **(*)** to the variable name and the values will be assigned to the variable as a list:

```
#Unpacking a tuple:

fruits = ("apple", "banana", "cherry", "mango")

(green, yellow, red) = fruits

print(green)
print(yellow)
print(red)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most
Cell In[19], line 5
      1 #Unpacking a tuple:
      3 fruits = ("apple", "banana", "cherry", "mango")
----> 5 (green, yellow, red) = fruits
      7 print(green)
      8 print(yellow)

ValueError: too many values to unpack (expected 3)
```



```
#Solution for unpacking problem

fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

(green, yellow, *red) = fruits

print(green)
print(yellow)
print(red)

print(type(red))
```

```
apple
banana
['cherry', 'strawberry', 'raspberry']
<class 'list'>
```

Tuple iteration

You can loop through the tuple items by using a `for` loop.

```
# Tuple iteration

fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")

for i in fruits:
    print(i)

print("\n-------Another method--------\n")
for i in range(len(fruits)):
    print(fruits[i])

#Can you try using 'while' loop? Do it now :)
```

```
apple
banana
cherry
strawberry
raspberry

-------Another method--------

apple
banana
cherry
strawberry
raspberry
```

Concatenation

You can merge/join/concat two tuples using the **“+”** operator

```
#Join/Concat/Merging

tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print("New Tuple \n")
print(tuple3)

print("\n------Another Operation ---------\n")

tuple3 = tuple3*2
print(tuple3)
```



```
New Tuple

('a', 'b', 'c', 1, 2, 3)

------Another Operation ---------

('a', 'b', 'c', 1, 2, 3, 'a', 'b', 'c', 1, 2, 3)
```

```python
# Concatenation
concatenated_tuple = (1, 2, 3) + (4, 5, 6)
print("Concatenated Tuple:", concatenated_tuple)

# Repetition
repeated_tuple = (1, 2, 3) * 3
print("Repeated Tuple:", repeated_tuple)

# Membership Testing
membership_result = 2 in (1, 2, 3)
print("Membership Testing Result:", membership_result)

# Indexing
indexing_result = (1, 2, 3)[0]
print("Indexing Result:", indexing_result)

# Slicing
sliced_result = (1, 2, 3)[1:3]
print("Sliced Result:", sliced_result)

# Length
length_result = len((1, 2, 3))
print("Length Result:", length_result)

# Iteration
print("Iterating Over Tuple:")
my_tuple = (1, 2, 3)
for item in my_tuple:
    print(item)

# Tuple Methods
count_result = (1, 2, 2, 3).count(2)
print("Count Result:", count_result)

# Packing and Unpacking
packed_tuple = 1, 2, 3
a, b, c = packed_tuple
print("Unpacked Values:", a, b, c)
```



```
Concatenated Tuple: (1, 2, 3, 4, 5, 6)
Repeated Tuple: (1, 2, 3, 1, 2, 3, 1, 2, 3)
Membership Testing Result: True
Indexing Result: 1
Sliced Result: (2, 3)
Length Result: 3
Iterating Over Tuple:
1
2
3
Count Result: 2
Unpacked Values: 1 2 3
```

`zip()` function

- In Python, the `zip()` function is used to combine multiple iterables (such as lists, tuples, or strings) into a single iterator of **tuples**.
- Each tuple contains elements from the corresponding positions of the input iterables.

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c', 'd']

zipped = zip(list1, list2)

for item in zipped:
    print(item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)

for num, lett in zipped:
    print(num)
    print(lett)
    print("------")
```

```
1
a
------
2
b
------
3
c
------
```

Contd…

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)

for item in zipped:
    print(item)

print("Printing Again")

for item in zipped:
    print(item)

print("Done")
```

```
(1, 'a')
(2, 'b')
(3, 'c')
Printing Again
Done
```

- In Python, **iterators** such as those returned by zip() are **consumed** as they are iterated over.
  - When you iterate over zipped with `for` item in zipped, it consumes the iterator and exhausts it.
- Once an iterator is exhausted, it **cannot be iterated over again**, and **subsequent attempts** to iterate over it will **yield no results**.

Set

- A set is a collection which is **unordered**, **unchangeable***, and **unindexed**.
    - Set items are unchangeable, but you **can remove** items and **add** new items
- Sets cannot have two items with the **same value**.

```
my_set = {1, 2, 3}
```

`add()`, `remove()`, `discard()`, `pop()`, `clear()`, `union()`, `intersection()`, `difference()`, `symmetric_difference()` `update()`, `intersection_update()`, `difference_update()`, `symmetric_difference_update()` `copy()`, `isdisjoint()`, `issubset()`, `issuperset()`


```python
#Python Set
# Defining a Set
my_set = {1, 2, 3, 4, 5}

# Adding Values to the Set
my_set.add(6)
my_set.add(7)

# Deleting Values from the Set
my_set.remove(3)
my_set.discard(8)  # Safe deletion, won't raise error if element not present

popped_value = my_set.pop()  # Removes and returns an arbitrary element

# Iterating Over the Set
print("Elements in the Set:")
for item in my_set:
    print(item)

my_set.add(2)
```

```
Elements in the Set:
2
4
5
6
7
```

```python
#Set operations
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# Adding Elements
set1.add(6)
set1.add(7)

# Removing Elements
set1.remove(3)
set1.discard(8)
popped_value = set1.pop()  # Removes and returns an arbitrary element

# Set Operations
union_set = set1.union(set2)
intersection_set = set1.intersection(set2)
difference_set = set1.difference(set2)
symmetric_difference_set = set1.symmetric_difference(set2)

# Updating Sets
set1.update({8, 9, 10})
set2.intersection_update({6, 7, 8})

# Set Comprehension
set3 = {x for x in range(10) if x % 2 == 0}
# Frozen Sets
frozen_set = frozenset([1, 2, 3, 4, 5])

print("Set 1:", set1)
print("Set 2:", set2)
print("Union Set:", union_set)
print("Intersection Set:", intersection_set)
print("Difference Set:", difference_set)
print("Symmetric Difference Set:", symmetric_difference_set)
print("Updated Set 1:", set1)
print("Updated Set 2:", set2)
print("Set Comprehension Result:", set3)
print("Frozen Set:", frozen_set)
```

```
Set 1: {2, 4, 5, 6, 7, 8, 9, 10}
Set 2: {8, 6, 7}
Union Set: {2, 4, 5, 6, 7, 8}
Intersection Set: {4, 5, 6, 7}
Difference Set: {2}
Symmetric Difference Set: {2, 8}
Updated Set 1: {2, 4, 5, 6, 7, 8, 9, 10}
Updated Set 2: {8, 6, 7}
Set Comprehension Result: {0, 2, 4, 6, 8}
Frozen Set: frozenset({1, 2, 3, 4, 5})
```

Feature	List	Tuple	Set
Mutable	Yes	No	Yes
Ordering	Ordered	Ordered	Unordered
Syntax	Square brackets []	Parentheses ()	Curly braces {}
Duplicate Elements	Allowed	Allowed	Not allowed
Use Cases	Dynamic data, variable length	Immutable data, fixed length	Unique elements, no duplicates
Examples	`my_list = [1, 2, 3]`	`my_tuple = (1, 2, 3)`	`my_set = {1, 2, 3}`
Accessing Elements	Indexing and Slicing	Indexing and Slicing	Iteration or membership testing
Modification	Easily modified with methods	Immutable, cannot be modified	Adding/removing elements
Memory Efficiency	More memory consumption	Less memory consumption	Memory-efficient for unique elements
Performance	Slower for large datasets	Faster than lists	Fast membership testing

Dictionary

- Dictionaries are used to store data values in **key**:value pairs.
- Dictionary items are **ordered**, **changeable**, and do not allow **duplicates**.

```
my_dict = {'name': 'Alice',
'age': 30, 'city': 'New York'}
```

- Elements are **referred/accessed** to by using the **key** name.

```
print(my_dict["name"])
```

```
# Dictionary to store students' grades
student_grades = {
    'Alice': 85,
    'Bob': 70,
    'Charlie': 92,
    'David': 78,
    'Eve': 88
}

# Accessing a student's grade
print("Bob's Grade:", student_grades['Bob'])

# Adding a new student
student_grades['Frank'] = 90

# Updating a student's grade
student_grades['David'] = 80

# Removing a student
del student_grades['Charlie']

print(student_grades)
```

```
Bob's Grade: 70
{'Alice': 85, 'Bob': 70, 'David': 80, 'Eve': 88, 'Frank': 90}
```

Operations on Dictionary

```python
# Dictionary containing information about a person
person = {
    'name': 'Alice',
    'age': 30,
    'city': 'New York'
}

# Accessing elements
print("Name:", person['name'])
print("Age:", person['age'])
print("City:", person['city'])

# Changing an element
person['age'] = 31
print("\nAfter Changing Age:")
print("Age:", person['age'])

# Adding a new element
person['gender'] = 'Female'
print("\nAfter Adding Gender:")
print("Gender:", person['gender'])

# Deleting an element
del person['city']
print("\nAfter Deleting City:")
print("City:", person.get('city', 'City not found'))
```



```
Name: Alice
Age: 30
City: New York

After Changing Age:
Age: 31

After Adding Gender:
Gender: Female

After Deleting City:
City: City not found
```

Keys and Values: Dictionary

- `for` loop can be used for iteration
- The `keys()` method will return a list of all the keys in the dictionary.
- The `values()` method will return a list of all the values in the dictionary.

```
student_grades = {
    'Alice': 85,
    'Bob': 70,
}
print("Accessing Keys")
for my_key in student_grades:
    print(my_key)

keys = student_grades.keys()
print(keys)

print("\n Accessing Values")
for my_values in student_grades.values():
    print(my_values)

vals = student_grades.values()
print(vals)

print("\n Accessing Keys and Values together")
for keys, values in student_grades.items():
  print(keys, values)
```

```
Accessing Keys
Alice
Bob
dict_keys(['Alice', 'Bob'])


 Accessing Values
85
70
dict_values([85, 70])


 Accessing Keys and Values together
Alice 85
Bob 70
```

Nested dictionary

A dictionary can contain dictionaries, this is called nested dictionaries.

Suppose, we need student's name as the key, and we need to add information such as ***age*** and ***grade*** for each student.


```
student_records = {
    'Alice': {'age': 20, 'grade': 'A'},
    'Bob': {'age': 21, 'grade': 'B'},
    'Charlie': {'age': 19, 'grade': 'C'},
}
```

In this case,

Outer dictionary contains the name as key for each student

Inner dictionary contains age and grade as keys

Nested dictionary

```python
student_records = {
    'Alice': {'age': 20, 'grade': 'A'},
    'Bob': {'age': 21, 'grade': 'B'},
    'Charlie': {'age': 19, 'grade': 'C'},
}

print("Keys:")
for key in student_records:  # Iterating through keys
    print(key)

print("\nValues:")
for value in student_records.values(): # Iterating through values
    print(value)

# Iterating through key-value pairs
print("\nKey-Value Pairs:")
for key, value in student_records.items():
    print(f"Name: {key}, Details: {value}")

print("\n--------Important----------")
alice_age = student_records['Alice']['age']
print("Alice's Age:", alice_age)

if 'Alice' in student_records:
        print("Student alread exists")
```



```
Keys:
Alice
Bob
Charlie

Values:
{'age': 20, 'grade': 'A'}
{'age': 21, 'grade': 'B'}
{'age': 19, 'grade': 'C'}

Key-Value Pairs:
Name: Alice, Details: {'age': 20, 'grade': 'A'}
Name: Bob, Details: {'age': 21, 'grade': 'B'}
Name: Charlie, Details: {'age': 19, 'grade': 'C'}

--------Important----------
Alice's Age: 20
Student alread exists
```

Nested dictionary

```
student_records = {
    'Alice': {'age': 20, 'grade': 'A'},
    'Bob': {'age': 21, 'grade': 'B'},
    'Charlie': {'age': 19, 'grade': 'C'},
}
```

```
name = input("Enter the Student name")

if name in student_records:
        print("This student alread exists")
        print(student_records[name])
else:
    age = int(input("Enter the 'Age' of the student"))
    grade = input ("Enter the 'Grade' of the student")

    student_records[name] = {'age': age, 'grade': grade}

    for i in student_records:
        print(i)
```



```
Enter the Student name John
Enter the 'Age' of the student 23
Enter the 'Grade' of the student A
Alice
Bob
Charlie
John
```

Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Practise test

Write a Python program to ask a user to enter username and password. The password should be minimum 9 characters long and should contain at least one alphabet and one number. Use validate_password () to validate the password. If validation is successful, add the username and password into a dictionary along with a hint msg. If the user already exists, show the hint msg and do not update/add the user.

Practice Test

1. Develop a program that simulates a simple ATM using functions and dictionaries. Users should be able to check balance, deposit money, and withdraw money.
2. Create a dictionary-based program that tracks the scores of players in a game. Allow users to add new players, update their scores, and remove players.

How to evaluate an algorithm (Algorithm complexity)

- An Algorithm is evaluated by means of the **resources** required by the algorithm to solve a given computational problem.
- Common resources include
  - **time complexity**, which measures the amount of time an algorithm takes to run,
  - **space complexity**, which measures the amount of memory an algorithm requires.

- Time complexity is concerned about how **fast** or **slow** particular algorithm performs.
- It can be defined as a numerical function *T(n) - time versus the input size n*.
- Time taken by an algorithm without depending on the implementation details

Time complexity

Constant time O(1):

- An algorithm is said to run in constant time
- it requires the **same amount of time** regardless of the *input size*.
- A single instruction will run exactly once when executed.
- It represents a straightforward, linear flow of execution without any repetition.


```
...
x = 5
y = 10
z = x + y
print(z)
...
```

- Time taken by this program is ***constant;*** denoted by **O(1)** [read as **Big O of one**]

Contd…

Linear time O(n): :

- An algorithm is said to run in linear time if it's **time execution** is **directly proportional** to the **input size**, i.e. time grows linearly as input size increases.

```
…
n = 5
for i in range(n):
    print("Iteration", i)
…
```


- Time taken by this program depends on *n;* denoted by **O(n)** [read as **Big O of N**]

Contd…

Quadratic time $O(n^2)$: :

- An algorithm is said to run in linear time if it's **time execution** is **directly proportional** to the **square** of the **input size**, i.e. time grows quadratically as input size increases.

```
…
n = 5
for i in range(n):
    for j in range(n):
        print("Iteration", j)
…
```


- Time taken by this program depends on n^2; denoted by **$O(n^2)$** [read as **Big O of N^2**]

Contd…

Polynomial time $O(n^k)$ (k>2)

- An algorithm is said to run in linear time if it's **time execution** is **directly proportional** to the **polynomial times** of the **input size**, i.e. time grows polynomially as input size increases.

```
…
n = 5
for i in range(n):
    for j in range(n):
        for k in range(n):
            print("Iteration", k)
…
```


- Time taken by this program depends on n^3; denoted by **$O(n^3)$** [read as **Big O of N^3**]

Contd…

Logarithmic time O(log n): :

- An algorithm is said to run in linear time if it's **time execution** is **directly proportional** to the **logarithm of** of the **input size**.

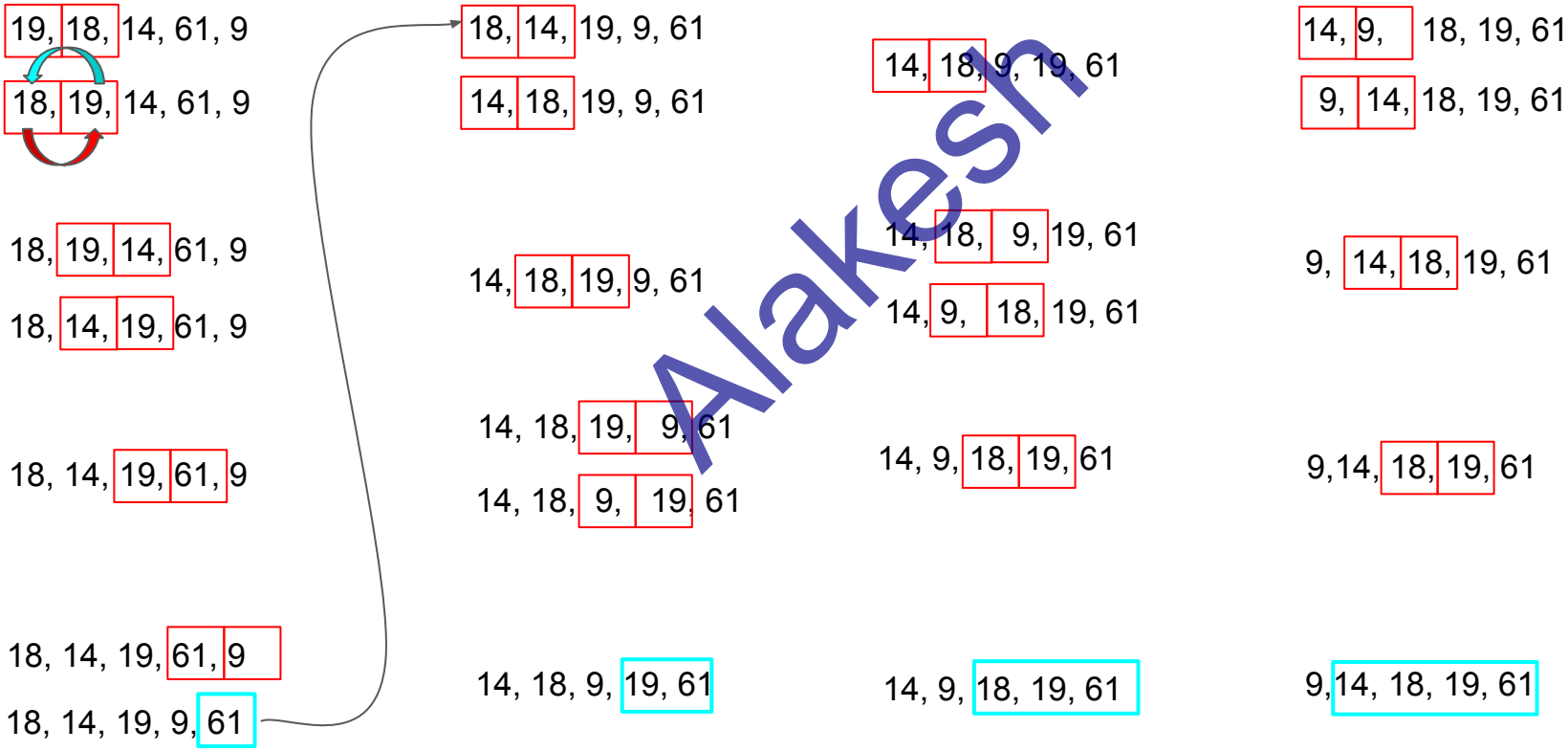
```
…
n = 100
current = 1
while current <= n:
    print(current)
    current *= 2
…
```


- Time taken by this program depends on *n;* denoted by **O(log n)** [read as **Big O of log n**]

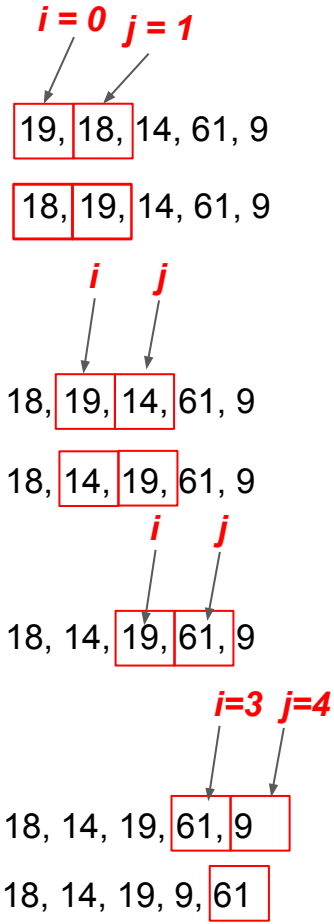
Improving the bubble sort

Bubble Sort: How it works

Input: 19, 88, 14, 61, 9



Bubble Sort: Implementation



- Start from the first and second elements
- If the first element is greater than the next one, swap them
- Move to the next pair of adjacent elements
- Do comparison and swap process
- Repeat until the end of array
- Repeat the above steps until the entire array is sorted.

Outer loop: k=1 to n-1 } *N-1 Iterations*

Inner loop: j=1 to n-1

i = 0, j = 1

if A[i]> A[j]:

```
temp =A[i]
A[i] =A[j]
A[j] =temp
```

} *N-1 Comparison*

```
i +=1
j +=1
```

4 x 4 = 16 comparisons !!!

Optimised Bubble Sort

19, 18, 14, 61, 9

18, 19, 14, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 9, 61

18, 19, 14, 61, 9

18, 14, 19, 61, 9

14, 18, 19, 9, 61

14, 18, 19, 9, 61

18, 14, 19, 61, 9

14, 18, 19, 9, 61

14, 18, 9, 19, 61

14, 9, 18, 19, 61

14, 18, 9, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

Outer loop: k=1 to n-1

Inner loop j=1 to n-1

i = j-1

if A[i]> A[j]:

temp =A[i]
A[i] =A[j]
A[j] =temp

n=n-1

Unordered

4 + 3 + 2 +1 =10 comparisons !!!

Optimised Bubble Sort

Input: 99, 1, 2, 3,4

99, 1, 2, 3,4

1, 99, 2, 3,4

1, 99, 2, 3,4

1, 2, 99, 3,4

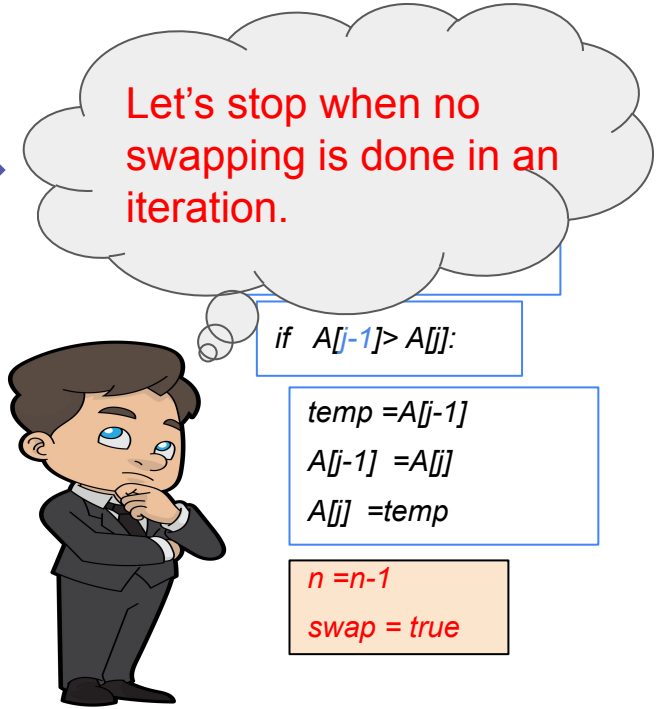
1, 2, 99, 3,4

1, 2, 3, 99,4

1, 2, 3, 4,99

```
Outer loop: k=1 to n-1
    Inner loop: j=1 to n-1
        if   A[j-1]> A[j]:
            temp =A[j-1]
            A[j-1]  =A[j]
            A[j]  =temp
        n =n-1
```


4 + 3 + 2 +1 =10 comparisons !!!



4 + 3 =7 comparisons !!!

Optimised Bubble Sort

Input: 3, 2, 1, 7, 8

Unordered

m = i

3, 2, 1, 7, 8

2, 3, 1, 7, 8

2, 1, 3, 7, 8

swapping

1, 2, 3, 7, 8

2, 3, 1, 7, 8

2, 1, 3, 7, 8

2, 1, 3, 7, 8

2, 1, 3, 7, 8

What if we remember the point where the last swapping is done.

Alakesh

4 + 1 =5 comparisons !!!

```
while (swap):
    swap = false
    Inner loop:  j=1 to m
        if   A[j-1]> A[j]:
            temp =A[j-1]
            A[j-1]  =A[j]
            A[j]  =temp
            m = j-1
            swap = true
```