Chapter 2.

Python Programming

Together for Tomorrow!
Enabling People
Education for Future Generations

UNIT 4.

Python IV

Samsung Innovation Campus

UNIT 4.
Python IV

What this unit is about:

- This unit will be our introduction to the object oriented programming.
- You will learn how to handle exceptions (run-time errors).

Expected outcome:

- Familiarity with the concepts of classes and objects.
- Ability to define classes.
- Ability to create objects based on classes.
- Ability to utilize methods and variables of the objects.

How to check your progress:

- Coding Exercises.
- Quiz.

Chapter 2.

Python Programming

Together for Tomorrow!
Enabling People
Education for Future Generations

UNIT 3. Python III

3.1. Control Structures.

3.2. Python Functions.

3.3. Python Input and Output.

Unit 4. Python IV

4.1. Classes and Objects.

4.2. Exception Handling.

Unit 5. Python V

5.1. Algorithms.

5.2. Data Structures.

5.3. Working with Files.

5.4. Working with Excel, Word, PDF Documents.

Classes and Objects (1/11)

Object Oriented Programming:

- Procedural Programming and Object Oriented Programming are both programming paradigms.
- Procedural Programming relies on calls to the subroutines and functions.
- Object Oriented Programming offers several advantages over Procedural Programming: debugging,

maintenance, code reusability, etc.

- Python is an Object Oriented Programming language.
- In Object Oriented Programming, objects with properties and methods are created.

Classes and Objects (2/11)

Object Oriented Programming:

- A class acts like a blueprint for creating objects.
- A class can be likened to a “cookie cutter”.
- An object is an “instantiated” class.
- Think of objects as “cookies” cut out with the cookie cutter (class).

Classes and Objects (3/11)

Object Oriented Programming:

- More than one object can be created based on the same class.
- An instance refers to an object allocated in the memory.
- There are concepts of member variable, member function (or method), class variable, etc.
- A class can be inherited by another class.

Classes and Objects (4/11)

Constructor method:

- This method is called when an object is first created.

```
In[1]  : class Dog:                                         # Declare Dog class.
   ... :         def __init__(self, name, age):             # Constructor method.
   ... :             self.name = name                       # Define a member variable.
   ... :             self.age = age                         # Define a member variable.
   ... :             print('A Dog object is created!')
In[2]  : dog1 = Dog('Fido', 2)                              # Constructor method called.
A Dog object is created!
```

Classes and Objects (5/11)

Destructor method:

- This method is called just before the object is destroyed (deleted).

```
In[1]  : class Dog:                                   # Declare Dog class.
    ... :            def __init__(self, name, age):   # Constructor method.
    ... :                self.name = name             # Define a member variable.
    ... :                self.age = age               # Define a member variable.
    ... :                print('A Dog object is created!')
 ... :               def __del__(self):               # Destructor method.
    ... :                print('A Dog object is deleted!')
In[2]  : dog1 = Dog('Fido', 2)
A Dog object is created!
In[3]  : del dog1
A Dog object is deleted!
```

Classes and Objects (6/11)

Member variable:

- Belongs to each object.

```
In[1]  : class Dog:                                          # Declare Dog class.
   ... :          def __init__(self, name, age):             # Constructor method.
   ... :              self.name = name                       # Member variable.
   ... :              self.age = age                         # Member variable.
In[2]  : dog1 = Dog('Fido', 2)                               # Object dog1 of class Dog.
In[3]  : dog2 = Dog('Dido', 3)                               # Object dog2 of class Dog.
In[4]  : dog1.name
Out[4]: Fido
In[5]  : dog2.age
Out[5]: 3
```

Classes and Objects (7/11)

Class variable:

- Belongs to a class and not to a particular object.

```
In[1]  : class Dog:                                  # Declare Dog class.
   ... :     counter = 0                              # Define a class variable.
   ... :     def __init__(self, name):
   ... :         self.name = name                    # Member variable.
   ... :         Dog.counter += 1                    # Increase the class variable.
   ... :     def __del__(self):
   ... :         Dog.counter -= 1                    # Decrease the class variable.
In[2]  : dog1 = Dog('Fido')
In[3]  : dog2 = Dog('Dido')
In[4]  : Dog.counter                                 # Class variable.
Out[4]: 2
```

Classes and Objects (8/11)

Class variable:

- Belongs to a class and not to a particular object.

```
In[5]  : del dog2
In[6]  : Dog.counter                                  # Class variable.
Out[6]: 1
In[7]  : del dog1
In[8]  : Dog.counter                                  # Class variable.
Out[8]: 0
```

Classes and Objects (9/11)

Member method:

- Belongs to each object and operates on the object’s member variables.

```
In[1]  : class Dog:
   ... :           def __init__(self, name, age):
   ... :               self.name = name
   ... :               self.age = age
   ... :           def bark(self):                                                # Member method.
   ... :               print(self.name + ' is barking.... woof... woof...')
In[2]  : dog1 = Dog('Fido', 2)
In[3]  : dog1.bark()
Fido is barking... woof... woof...
```

Classes and Objects (10/11)

Inheritance:

- A class (child) can be based on the methods and variables of another class (parent).

```
In[1]  : class Pet:                                          # Declare Pet class.
   ... :          def __init__(self, name):
   ... :           self.name = name
In[2]  : class Cat(Pet):                                     # Cat class inherited from Pet class.
   ... :          def purr(self):                           # A member method specific to the Cat class.
   ... :          print(self.name + ' is purring...')
In[3]  : class Dog(Pet):                                     # Dog class inherited from Pet class.
   ... :          def bark(self):                           # A member method specific to the Dog class.
   ... :          print(self.name + ' is barking...')
```

Classes and Objects (11/11)

Inheritance:

- A class (child) can be based on the methods and variables of another class (parent).

```
In[4]  : cat1 = Cat('Kitty')
In[5]  : dog1 = Dog('Fido')
In[6]  : cat1.purr()
Kitty is purring...
In[7]  : dog1.bark()
Fido is barking...
```

Coding Exercise #0107

Follow practice steps on 'ex_0107.ipynb'

Coding Exercise #0108

Follow practice steps on 'ex_0108.ipynb'

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 2.

Python Programming

UNIT 3. Python III

3.1. Control Structures.

3.2. Python Functions.

3.3. Python Input and Output.

Unit 4. Python IV

4.1. Classes and Objects.

4.2. Exception Handling.

Unit 5. Python V

5.1. Algorithms.

5.2. Data Structures.

5.3. Working with Files.

5.4. Working with Excel, Word, PDF Documents.

Exception Handling (1/6)

What exception handling is:

- Programs with no syntax errors could crash during the run-time.
- Run-time errors (exceptions) need to be handled properly to avoid crash.
- Exception handling controls the program flow when a run-time error is raised.

Some of the most common run-time errors are listed as below:

Error Name	Explanation
ZeroDivisionError	Raised when division by zero happens.
IndexError	Raised when index of a tuple, list, etc. is out of bounds.
ValueError	Raised when search fails on a tuple, list, etc.
KeyError	Raised when non-existing key is used to access a dictionary.

Exception Handling (2/6)

- Handle all exceptions:

```
try:
    <Code block where an exception may happen>
except:
    <Code block to run when an exception happens>
```

Exception Handling (3/6)

Handle a specific exception:

```
try:
    <Code block where an exception may happen>
except <Error name>:
    <Code block to run when an exception happens>
```

```
try:
    <Code block where an exception may happen>
except <Error name> as <Error variable>:                    # Error variable defined.
    <Code block to run when an exception happens>
```

Exception Handling (4/6)

Exception handling with else and finally:

```
try:
    <Code block where an exception may happen>
except:
    <Code block to run when an exception happens>
else:
    <Code block to run when there is no exception>
finally:
    <Whether exception happens or not, run this code block>
```

Exception Handling (5/6)

Exception handling with else and finally:

```
In[1]  : try:
   ... :     result = 123/x                          # An exception can happen when x is 0.
   ... : except ZeroDivisionError as err:
   ... :     print(err)
   ... : else:
   ... :     print(result)
   ... : finally:
   ... :     print('The End')
```

Exception Handling (6/6)

- Exception handling with else and finally:

```
In[1]  : try:
    ... :     result = x.index(1234)                              # An exception can happen if 1234 is not found in x.
    ... : except ValueError as err:
    ... :     print(err)
    ... : else:
    ... :     print(result)
    ... : finally:
    ... :     print('The End')
```

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 2.

Python Programming

UNIT 5.

Python V

Samsung Innovation Campus

UNIT 5.
Python V

What this unit is about:

- You will develop algorithms for problem-solving.
- You will implement data structures such as stacks and queues.
- You will learn how to store complex objects in external files.
- You will learn how to retrieve data from the Excel, Word, PDF documents.

Expected outcome:

- Ability to formulate and apply algorithms for problem-solving.
- Ability to interact with some of the most common document types.

How to check your progress:

- Coding Exercises.
- Quiz.

Chapter 2.

Python Programming

Together for Tomorrow! Enabling People

Education for Future Generations

UNIT 3. Python III

3.1. Control Structures.

3.2. Python Functions.

3.3. Python Input and Output.

Unit 4. Python IV

4.1. Classes and Objects.

4.2. Exception Handling.

Unit 5. Python V

5.1. Algorithms.

5.2. Data Structures.

5.3. Working with Files.

5.4. Working with Excel, Word, PDF Documents.

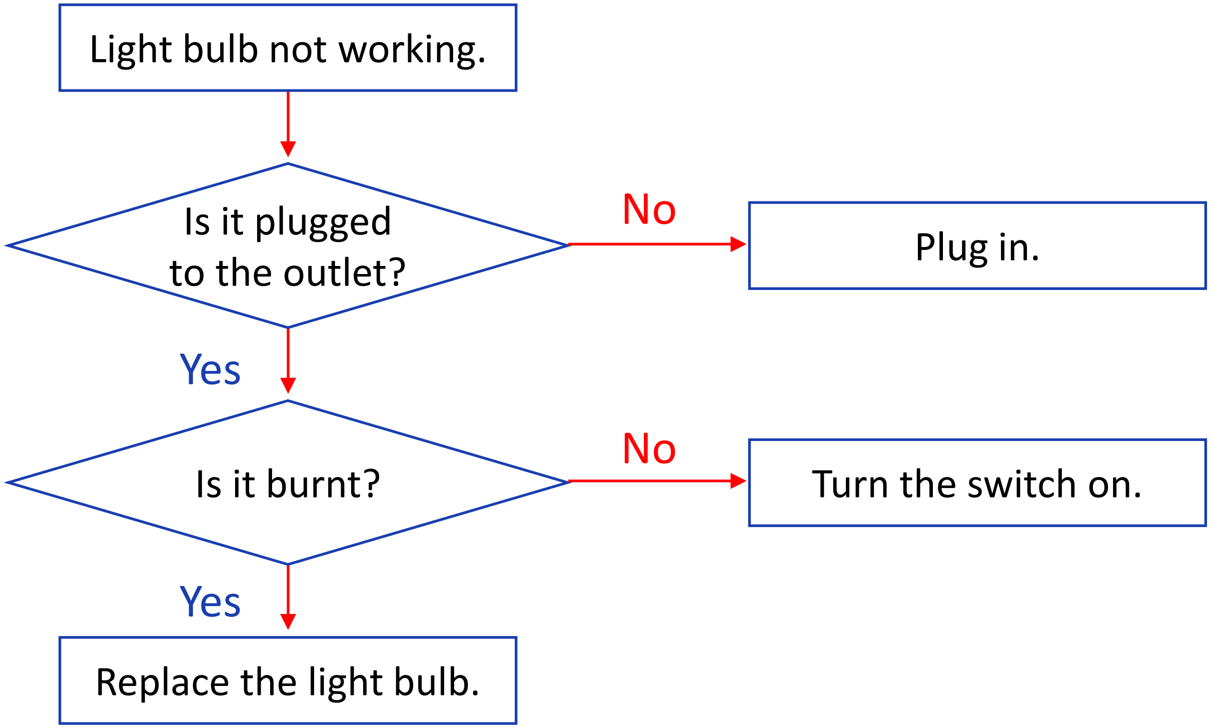
Algorithms (1/14)

What is an algorithm?

- It is a set of procedures and rules for problem solving (implemented in a programming language).
- Each step of an algorithm should be concise and clear.

Algorithms (2/14)

Algorithm to repair a failing light bulb:



Light bulb not working.

Is it plugged to the outlet?

No

Plug in.

Yes

Is it burnt?

No

Turn the switch on.

Yes

Replace the light bulb.

Algorithms (3/14)

Algorithm Example #1: Calculate the absolute value x.

1). Check whether x is positive or negative.

2). If x is a positive number, return x.

3). On the contrary, if x is a negative number, return $-x$.

Algorithms (4/14)

Algorithm Example #1: Calculate the absolute value x.

```
In[1]  :  def ABS(x):
    ... :      if x >= 0:
    ... :         result = x
    ... :      else:
    ... :         result = -x
    ... :      return result

In[2]  :  ABS(-3)
Out[2]: 3

In[3]  :  ABS(4)
Out[3]: 4
```

Algorithms (5/14)

Algorithm Example #2: Find the maximum value from a list.

1). Iterate through the list and get each value in a sequence.

2). Store the first value as the temporary maximum.

3). From the second value and on, compare it with the temporary maximum.

- If the value is larger than the stored one, then this becomes the new temporary maximum.

4). The temporary maximum that remains at the end is the maximum of the list.

Algorithms (6/14)

Algorithm Example #2: Find the maximum value from a list.

```
In[1]  : def MAX(x):
    ... :         n = len(x)
    ... :         my_max = x[0]                      # First value stored as temporary maximum.
    ... :         for i in range(1, n):
    ... :             if x[i] > my_max:              # If the value is larger than the stored maximum,
    ... :                 my_max = x[i]              # replace the temporary maximum.
    ... :         return my_max                      # Return the temporary maximum.
In[2]  :  a = [ 999, 131, -542, 1022, 1021, 45, 77]
In[3]  :  MAX(a)
Out[3]: 1022
```

Algorithms (7/14)

Algorithm Example #2: Find the maximum value from a list.

1). Iterate through the list and get each item in a sequence.

2). At each step, compare the item with the succeeding ones.

- If repetition is detected, store it in a set.

3). Continue until the penultimate item in the last.

4). Output the set.

Algorithms (8/14)

Algorithm Example #3: Find repetitions in a list.

```
In[1]  : def FIND_SAME(x):
    ... :     n = len(x)
    ... :     result = set()                 # A set that will contain the result.
    ... :     for i in range(0, n-1):        # Iterate from 0 to n-2. n-2 is the second index from the last.
    ... :         for j in range(i + 1, n):  # Iterate from i+1 to n-1. n-1 is the last index.
    ... :             if x[i] == x[j]:       # When repetition is detected,
    ... :                 result.add(x[i])   # add it to the result set.
    ... :     return list(result)
In[2]  : a = ['Tom', 'Jerry', 'Mike', 'Sara', 'Tom', 'Sara', 'John']
In[3]  : FIND_SAME(a)
Out[3]: ['Sara', 'Tom']
In[4]  : b = [1,1,1,2,3,4,2,3,5,6,7,8,9,4]
In[5]  : FIND_SAME(b)
Out[5]: [1, 2, 3, 4]
```

Algorithms (9/14)

Algorithm Example #4: Find repetitions in a list (using dictionary).

1). Create an empty dictionary.

2). Iterate through the list and get each item in a sequence.

- If the extracted item is not in the dictionary as a key, then create a pair using this item as key and 1 as value.
- IF the extracted item is already in the dictionary as a key, increase the corresponding value by 1.

3). Output those keys for which values are equal or larger than 2.

Algorithms (10/14)

- Algorithm Example #4: Find repetitions in a list (using dictionary).

```
In[1]  : def FIND_SAME_DICT(x):
   ... :            my_dict = {}                          # An empty dictionary.
   ... :      for name in x:                              # Loop though the items in a list.
   ... :          if name in my_dict:                     # If the item is already included in the dictionary as key,
   ... :             my_dict[name] += 1                   # increase the corresponding value by 1.
   ... :          else:
   ... :             my_dict[name] = 1                    # Include the item as a new key with value equal to 1.
   ... :      result = []
   ... :      for name in my_dict:
   ... :          if my_dict[name] >= 2:                  # If the value is equal or lager than 2,
   ... :             result.append(name)                  # append to the result list.
   ... :      return result
In[2]  : FIND_SAME(a)                                     # You assume that a is a list of names.
Out[2]: ['Sara', 'Tom']
```

Algorithms (11/14)

Scenario: in a thick book we'd like to find the page 618.

- Flipping the pages one by one from 1 until the desired page is found can certainly work but it is very inefficient.
- You would like to find the desired page with less effort.
- You can take advantage of the fact that the page numbers are already ordered (sorted) from the smallest to the largest.

Algorithms (12/14)

Scenario: in a thick book we'd like to find the page 618.

1). Open the somewhere in the middle, the page is, say, 520.

2). As 618 is larger than 520, you can narrow the search range to (520, end).

3). Open again somewhere middle between 520 and the end, the page is, say, 720.

4). As 618 is smaller than 720, you can further narrow our search range to (520, 720).

5). As you repeat the trials, the search range narrows the desired page is reached.

Algorithms (13/14)

Algorithm Example #5: Binary search in a sorted list.

1). Initially set the search range with the lower bound = 0 and the upper bound = the list length.

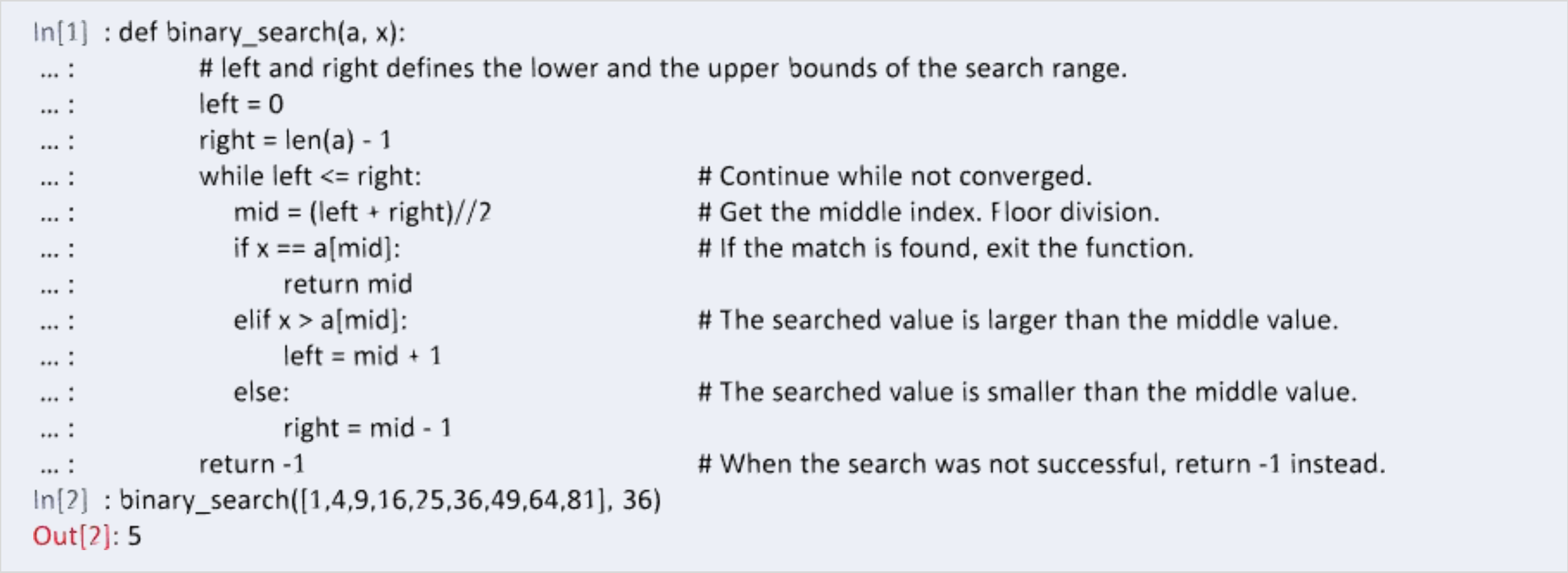
2). Get the middle value from the search range and compare it with the searched value.

- If they match, return the position and end the search.
- If the searched value is larger than the middle value, set this middle value as the new lower bound of the search range.
- If the searched value is smaller than the middle value, set this middle value as the new upper bound of the search range.

3) Repeat from the step 2.

Algorithms (14/14)

Algorithm Example #5: Binary search in a sorted list.



```
In[1]  : def binary_search(a, x):
 ... :          # left and right defines the lower and the upper bounds of the search range.
 ... :          left = 0
 ... :          right = len(a) - 1
 ... :          while left <= right:                 # Continue while not converged.
 ... :              mid = (left + right)//2          # Get the middle index. Floor division.
 ... :              if x == a[mid]:                  # If the match is found, exit the function.
 ... :                  return mid
 ... :              elif x > a[mid]:                 # The searched value is larger than the middle value.
 ... :                  left = mid + 1
 ... :              else:                            # The searched value is smaller than the middle value.
 ... :                  right = mid - 1
 ... :          return -1                            # When the search was not successful, return -1 instead.
In[2]  : binary_search([1,4,9,16,25,36,49,64,81], 36)
Out[2]: 5
```

Coding Exercise #0109

Follow practice steps on 'ex_0109.ipynb'

Chapter 2.

Python Programming

Together for Tomorrow! Enabling People

Education for Future Generations

UNIT 3. Python III

3.1. Control Structures.

3.2. Python Functions.

3.3. Python Input and Output.

Unit 4. Python IV

4.1. Classes and Objects.

4.2. Exception Handling.

Unit 5. Python V

5.1. Algorithms.

5.2. Data Structures.

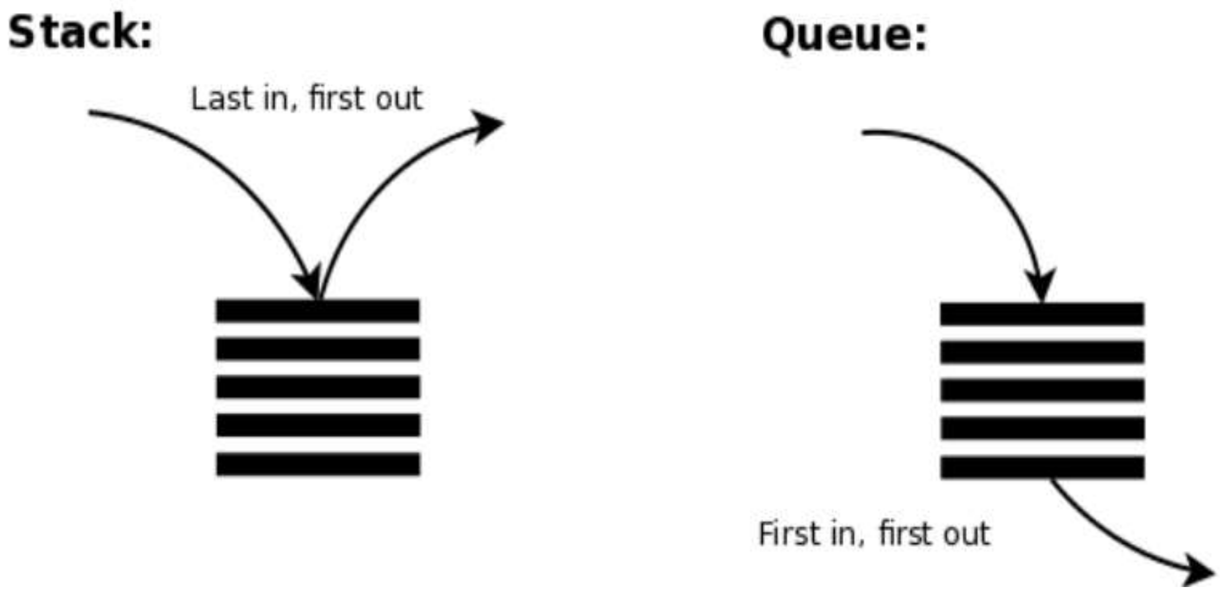
5.3. Working with Files.

5.4. Working with Excel, Word, PDF Documents.

Data Structures (1/4)

Stack and Queue:

- In a stack, the last value in is the first value out (LIFO).
- In a queue, the first value in is the first value out (FIFO).



Data Structures (2/4)

Stack and Queue:

- You can implement stack and queue data structures with Python list.

Data Structure	Action	Code	Explanation
Queue	Initialize	qu = []	Create an empty list.
	Enqueue	qu.append(x)	Append an item at the end. Length increased by 1.
	Dequeue	x = qu.pop(0)	Take an item from the beginning. Length shortened by 1.
Stack	Initialize	st = []	Create an empty list.
	Push	st.append(x)	Append an item at the end. Length increased by 1.
	Pop	x = st.pop()	Take an item from the end. Length shortened by 1.

Data Structures (3/4)

Stack's pop action with Python list:

```
In[1]  : a=['a','b','c','d','e']
In[2]  : while a:
 ... :         print(a.pop())                                          # Pop a value from the stack.
Out[2]:
e
d
c
b
a
```

Data Structures (4/4)

Queue’s dequeue action with Python list:

```
In[1]  : a=['a','b','c','d','e']
In[2]  : while a:
 ... :        print(a.pop(0))                                    # Dequeue a value from the queue.
Out[2]:
a
b
c
d
e
```