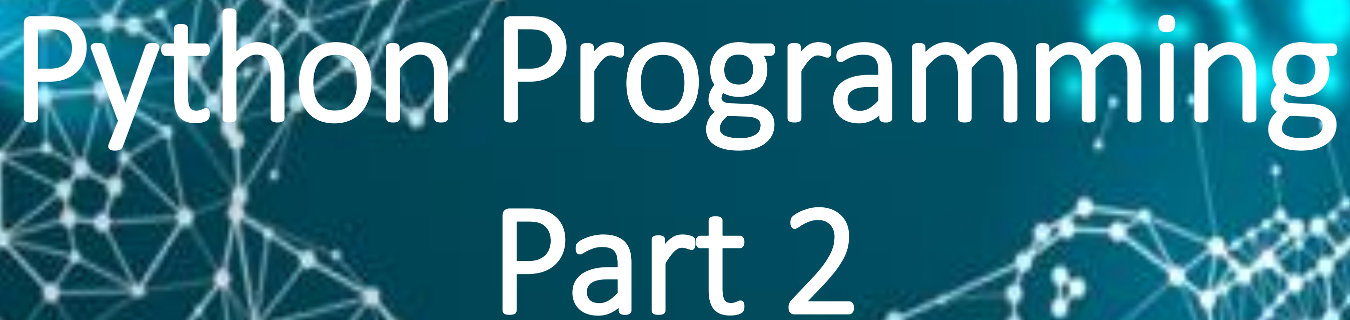
Python Programming Part 2



Instructor: Vision Wang

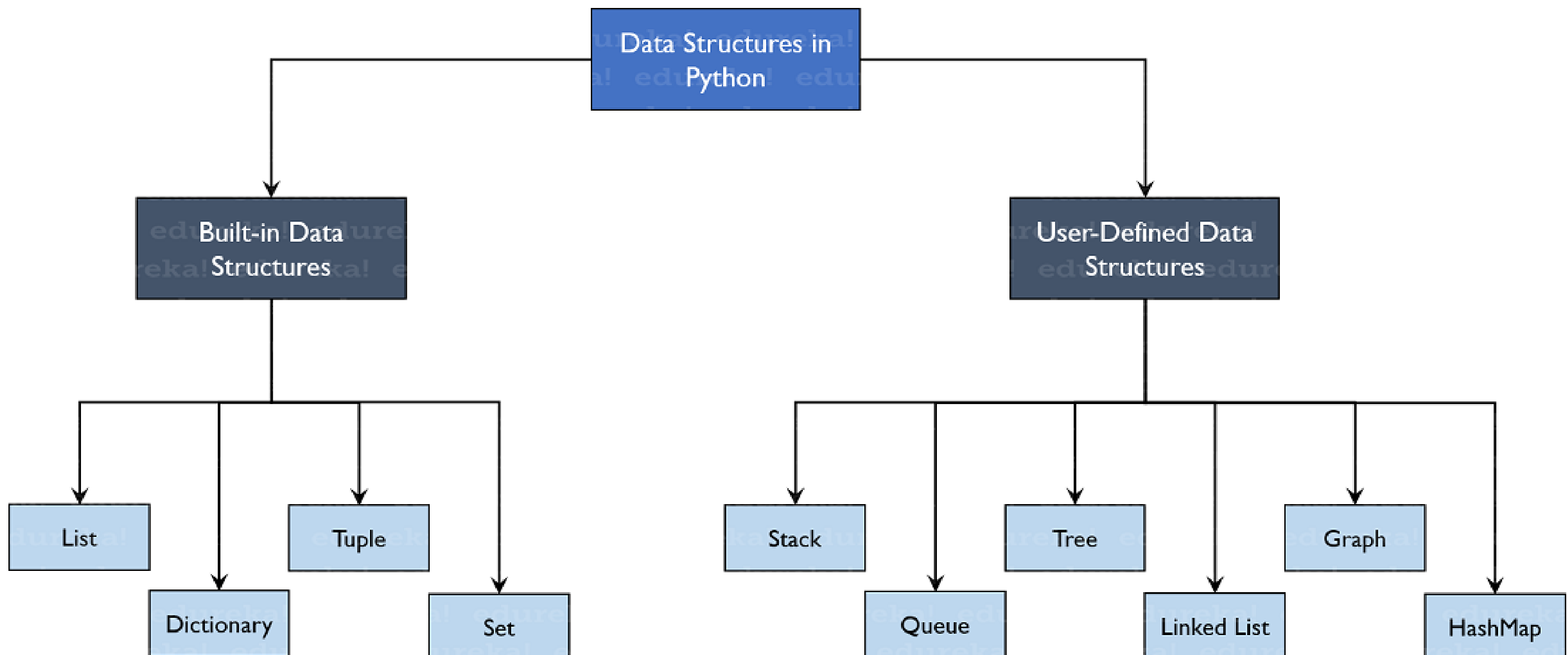
Email: xinwang35314@gmail.com

Part 2

- Data Structures
- Modules
- Errors and exceptions
- Classes

Data Structures

- **Data Structures** allow you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.
- Python has implicit support for Data Structures which enable you to store and access data. These structures are called List, Dictionary, Tuple and Set.



Built-in Data Structures

These Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster.

Lists

- **Lists** are used to store data of different data types in a sequential manner.
- Index – the addresses assigned to every element of the list.
- **Positive indexing** – starts from 1 and goes on until the last element.
- **Negative indexing** – starts from -1 enabling you to access elements from the last to first.

```
>>> my_list = [1,2,3,"apple","banana",2.33]
>>> print(my_list[2])
3
>>> print(my_list[-2])
banana
```

Dictionary

- Dictionaries are used to store key-value pairs.

What is key-value pair (KVP)?

- A **key-value pair (KVP)** is a set of two linked data items.
- A **key** is a unique identifier for some item of data.
- A **value** is the data that is.
- Key-value pairs are frequently used in tables.

Creating a Dictionary

```
>>> my_dict =
{"firstName":"Bugs","lastName":"Bunny","location":
"Earth"}
>>> print(my_dict)
{'lastName': 'Bunny', 'firstName': 'Bugs', 'location':
'Earth'}
```

key	value
firstName	Bugs
lastName	Bunny
location	Earth

Dictionary

Changing the adding key, value pairs

```
>>> my_dict["location"] = "USA"
>>> print(my_dict)
{'lastName': 'Bunny', 'firstName':
'Bugs', 'location': 'USA'}
>>> my_dict["Phone"] = 1234567
>>> print(my_dict)
{'lastName': 'Bunny', 'firstName':
'Bugs', 'location': 'USA', 'Phone':
1234567}
```

Accessing Elements

- Using the keys only.
- Using the **get()** function

```
>>> print(my_dict['lastName'])
Bunny
>>> print(my_dict.get("lastName"))
Bunny
```

Deleting key, value pairs

- Delete the values using the **pop()** function.
- Clear the entire dictionary using **clear()** function.

```
>>> a = my_dict.pop("Phone")
>>> print(a)
1234567
>>> print(my_dict)
{'lastName': 'Bunny', 'firstName':
'Bugs', 'location': 'USA'}
>>> my_dict.clear()
>>> print(my_dict)
{}
```

Dictionary

- Other functions: **keys()**, **values()**, **items()**

```
>>> my_dict =
{"firstName":"Bugs","lastName":"Bunny","location":"Earth"}
>>> print(my_dict.keys())
dict_keys(['lastName', 'firstName', 'location'])
>>> print(my_dict.values())
dict_values(['Bunny', 'Bugs', 'Earth'])
>>> print(my_dict.items())
dict_items([('lastName', 'Bunny'), ('firstName', 'Bugs'), ('location', 'Earth')])
```

Tuple

- Tuples are the same as lists with the exception that the data once entered into the tuple cannot be changed no matter what.
- The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed.

```
>>> mylist = ["apple","banana","pear"]
>>> mylist[0] = "bird"
>>> print(mylist)
['bird', 'banana', 'pear']
>>> mytuple = ("apple","banana","pear")
>>> mytuple[0] = "bird"
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    mytuple[0] = "bird"
TypeError: 'tuple' object does not support item assignment
```

Tuple

Creating a Tuple

```
>>> mytuple = ("apple","banana",2)
>>> print(mytuple)
('apple', 'banana', 2)
```

Accessing Elements

```
>>> print(mytuple[0])
apple
>>> print(mytuple[0][1])
p
```

Appending Elements

```
>>> mytuple = mytuple + (2 , 3,
"sky")
>>> print(mytuple)
('apple', 'banana', 2, 2, 3, 'sky')
```

Q: Think about what result you can print out according to the code below?

```
>>>my_tuple = (1, 2, 3, ['hindi',
'python'])
>>>my_tuple[3][0] = 'english'
>>>print(my_tuple)
>>>print(my_tuple.count(2))
>>>print(my_tuple.index(['english'
, 'python']))
```

Set

- Sets are a collection of unordered elements that are unique.

Creating a set

```
>>>my_set = {1, 2, 3, 4, 5, 5, 5}
>>>print(my_set)
{1, 2, 3, 4, 5}
```

Adding Elements

- Use **add()** function.

```
>>>my_set = {1, 2, 3}
>>>my_set.add(4)
>>>print(my_set)
{1, 2, 3, 4}
```

Operations in sets

- **union()** function – combines the data in both sets.
- **intersection()** function – finds the data present in both sets only.
- **difference()** function – deletes the data present in both and outputs data present only in the set passed.
- **symmetric_difference()** function – same as the difference() function but return the remaining in both sets.

Set

Q: What’s the return for the code below? Think about it and try to see if it’s same as what you think.

```
>>>my_set = {1, 2, 3, 4}
>>>my_set_2 = {3, 4, 5, 6}
>>>print(my_set.union(my_set_2))
>>>print(my_set.intersection(my_set_2))
>>>print(my_set.difference(my_set_2))
>>>print(my_set.symmetric_difference(my_set_2))
>>>my_set.clear()
>>>print(my_set)
```

Modules

- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module can define functions, classes and variables.

An example of a simple module. This module file is named aname.py .

```
def print_func(par):
    print("Hello : ", par)
    return

print(print_func())
```

Import Statement

You can use any Python source file as a module by executing an *import* statement in some other Python source file. The import syntax:

import module_name

```
# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

from…import Statement

from statement lets you import specific attributes from a module into the current namespace. The syntax:

from module_name import name1

from…import * Statement

Import all names from a module into the current namespace by using the following syntax:

from module_name import *

Q: Could you create a module for *Fibonacci numbers*?

- The function the module can fulfill is to return all the Fibonacci numbers less than or equal to a certain number, which is an input number.

Errors and Exceptions

There are two distinguishable kinds of errors: syntax errors and exceptions.

Syntax Errors

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Exceptions

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Classes

- **Class** – A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Creating classes

The *class* statement creates a new class definition. The name of the class immediately follows the keyword *class* followed by a colon as follows

```
class ClassName:
   'Optional class documentation string'
   class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.__doc__*.
- The *class_suite* consists of all the component statements defining class members, data attributes and functions.

Class Example

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print("Total Employee %d" % Employee.empCount)

   def displayEmployee(self):
      print("Name : ", self.name,  ", Salary: ", self.salary)

#This would create first object of Employee class
emp1 = Employee("Zara", 2000)
#This would create second object of Employee class
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Try and see, what will be printed out as result?

Classes

Creating Instance Objects

- To create instances of a class, you call the class using class name and pass in whatever arguments its ___*init*___ method accepts.

```
#This would create first object of Employee class emp1 =
Employee("Zara", 2000)
#This would create second object of Employee class
emp2 = Employee("Manni", 5000)
```

Accessing Attributes

- Access the object's attributes using the dot operator with object.

```
>>>emp1.displayEmployee()
>>>emp2.displayEmployee()
>>>print("Total Employee %d" % Employee.empCount)
```