CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - III

Sub:	Python Programming							Code:	16MCA21
Date:	/05/2017	Duration:	90 mins	Max Marks:	50	Sem:	II	Branch:	MCA

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1(a)	Write steps of Object Oriented development phases	[6]	CO1	L2
(b)	Explain in detail about IsInstance function with suitable examples	[4]	CO1	L2
2(a)	Explain about Class Object attributes	[6]	CO1	L2

(b)	Draw the memory model for the following:
	`>>> class Book:`
	`...  """Information about a book."""`
	`...`
	`>>> ruby_book = Book()`
	`>>> ruby_book.title = 'Programming Ruby'`
	`>>> ruby_book.authors = ['Thomas', 'Fowler', 'Hunt']`
	`>>> ruby_book.title`
	`'Programming Ruby'`
	`>>> ruby_book.authors`
	`['Thomas', 'Fowler', 'Hunt']`
		[4]	CO2	L1

3(a)	Write and Explain Book class with sufficient details	[6]	CO1	L2
(b)	Explain about __init__ method in a class	[4]	CO1	L2
4(a)	Write code to compare to two book objects based on ISBN	[4]	CO1	L2
(b)	Explain in detail about Encapsulation,Inheritance and Polymorphism	[6]	CO1	L2
5(a)	Explain short notes on GUI and event driven programming	[6]	CO2	L2
(b)	Explain about TKinter module and write a basic GUI program and explain line by line	[4]	CO2	L2
6(a)	Explain about various tkinter widgets	[4]	CO2	L2
(b)	Explain tkinter based python program for creating a GUI that has a label,Entry and a button. The values given in Entry field should be updated in Label on click of the button	[6]	CO2	L4
7(a)	Explain about Special class Attributes	[4]	CO1	L2
(b)	Write python code for creating Member and Faculty and student class such that Faculty and students are subclass of Member	[6]	CO1	L2
8a)	Write python code for inverting a dictionary	[6]	CO2	L4

b)	What is the output of the following code ?
	`1. class tester:`
	`2.  def __init__(self, id):`
	`3.  self.id = str(id)`
	`4.  id="224"`
	`5.`
	`6. >>>temp = tester(12)`
	`7. >>>print temp.id`
		[4]	CO1	L4

Subject :Python Programming
Subject Code:16MCA21

IA3 -Solution

1 a) Write steps of Object Oriented development phases:

Object-oriented programming revolves around defining and using new types.a class is how Python represents a type. Object-oriented programming involvesat least these phases:

1. *Understanding the problem domain.* This step is crucial: you need to know what your customer wants (your boss, perhaps a friend or business contact,perhaps yourself) before you can write a program that does what the customer wants.
2. *Figuring out what type(s) you might want.* A good starting point is to readthe description of the problem domain and look for the main nouns and noun phrases.
3. *Figuring out what features you want your type to have.* Here you should write some code that *uses* the type you're thinking about, much like wedid with the Book code at the beginning of this chapter. This is a lot like the Examples step in the function design recipe, where you decide what the code that you're about to write should do.
4. *Writing a class that represents this type.* You now need to tell Python aboutyour type. To do this, you will write a class, including a set of methods inside that class. (You will use the function design recipe as you designand implement each of your methods.
5. *Testing your code.* Your methods will have been tested separately as you followed the function design recipe, but it's important to think about how the various methods will interact.

b) Explain in detail about IsInstance function with suitable examples

Function isinstance reports whether an object is an *instance* of a class—that is, whether an object has a particular type:

```
>>> isinstance('abc', str)
True
>>> isinstance(55.2, str)
False
```

'abc' is an instance of str, but 55.2 is not.

Python has a class called object. Every other class is based on it:

```
>>> help(object)
Help on class object in module builtins:
class object
| The most base type
```

Function isinstance reports that both 'abc' and 55.2 are instances of class object:

```
>>> isinstance(55.2, object)
True
>>> isinstance('abc', object)
True
```

Even classes and functions are instances of object:

```
>>> isinstance(str, object)
True
>>> isinstance(max, object)
True
```

Every class in Python is *derived* from class object, and so every instance of every class is an object.

2 a) Explain about Class Object attributes:

Attributes are variables inside a class that refer to methods, functions, variables, or even other classes.

Class object has the following *attributes:*

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Every class in Python, including ones that you define, automatically *inherits*these attributes from class object: they are automatically part of every class.More generally, every subclass inherits the features of its superclass. This isa powerful tool: it helps avoid a lot of duplicate code and makes interactions between related types consistent.

For the following class:

```
>>> class Book:
... """Information about a book."""
...
```

Subject :Python Programming
Subject Code:16MCA21

Just as keyword def tells Python that we're defining a new function, keyword class signals that we're defining a new type.

Much like str is a type, Book is a type:

```
>>> type(str)
<class 'type'>
>>> type(Book)
<class 'type'>
```

Our Book class isn't empty, either, because it has inherited all the attributes of class object:

```
>>> dir(Book)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
```

There are four extra attributes in class Book; every subclass of class object automatically has these attributes in addition to the inherited ones:

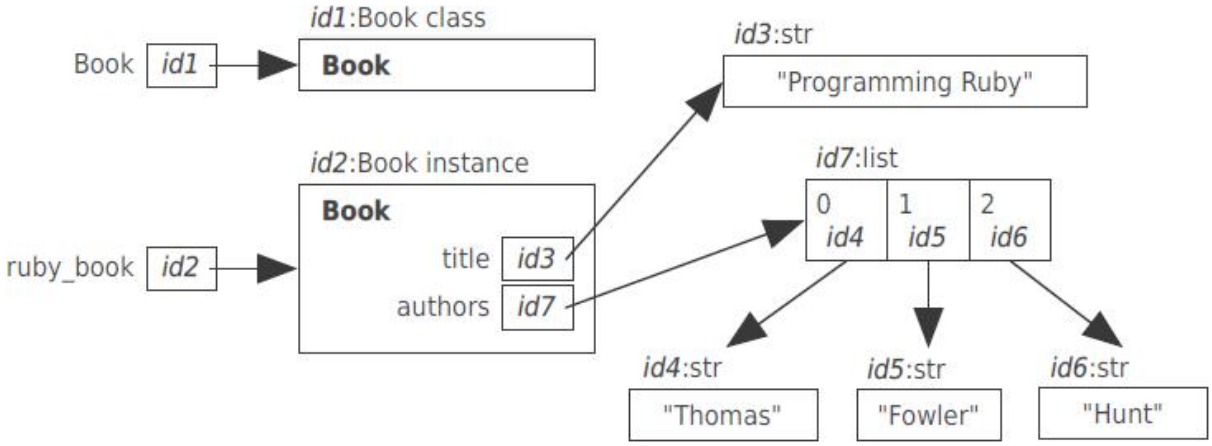
'__dict__', '__module__', '__qualname__', '__weakref__'

b) Draw the memory model for the following:

```
>>> class Book:
...     """Information about a book."""
...
 >>> ruby_book = Book()
 >>> ruby_book.title = 'Programming Ruby'
 >>> ruby_book.authors = ['Thomas', 'Fowler', 'Hunt']

 >>> ruby_book.title
 'Programming Ruby'
 >>> ruby_book.authors
 ['Thomas', 'Fowler', 'Hunt']
```

Memory model:



3 a) Write and Explain Book class with sufficient details:

class Book:

"""Information about a book, including title, list of authors publisher, ISBN, and price.

"""

def __init__(self, title, authors, publisher, isbn, price):

""" (Book, str, list of str, str, str, number) -> NoneType

Create a new book entitled title, written by the people in authors,published by publisher, with ISBN isbn and costing price dollars.

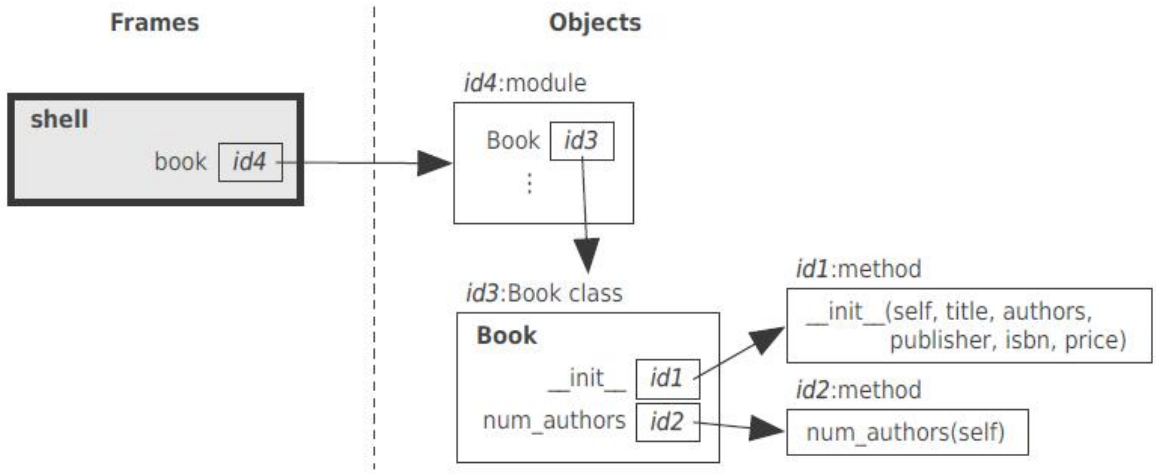
>>> python_book = Book(\'Practical Programming', \['Campbell', 'Gries', 'Montojo'], \'Pragmatic Bookshelf', \'978-1-93778-545-1', \ 25.0)

>>> python_book.title

'Practical Programming'

```
>>> python_book.authors
['Campbell', 'Gries', 'Montojo']
>>> python_book.publisher
'Pragmatic Bookshelf'
>>> python_book.ISBN
'978-1-93778-545-1'
>>> python_book.price
25.0
"""
self.title = title
# Copy the authors list in case the caller modifies that list later.
self.authors = authors[:]
self.publisher = publisher
self.ISBN = isbn
self.price = price
def num_authors(self):
""" (Book) -> int
Return the number of authors of this book.
>>> python_book = Book( \
'Practical Programming', \
['Campbell', 'Gries', 'Montojo'], \
'Pragmatic Bookshelf', \
'978-1-93778-545-1', \
25.0)
>>> python_book.num_authors()
3
"""
return len(self.authors)
```

The class definition. When Python executes this module, it creates a class object and assigns it to variable Book:



b) Explain about __init__ method in a class

Method __init__ is called whenever a Book object is created. Its purpose is to initialize the new object; this method is sometimes called a *constructor*. Here are the steps that Python follows when creating an object:
1. It creates an object at a particular memory address.
2. It calls method __init__, passing in the new object into the parameter self.
3. It produces that object's memory address.

4) Write code to compare to two book objects based on ISBN.

```
class Book:
"""Information about a book, including title, list of authors,
publisher, ISBN, and price.
"""
def __init__(self, title, authors, publisher, isbn, price):
self.title = title
# Copy the authors list in case the caller modifies that list later.
self.authors = authors[:]
self.publisher = publisher
self.ISBN = isbn
```

```
self.price = price
def num_authors(self):
""" (Book) -> int
Return the number of authors of this book.
>>> python_book = Book( \
'Practical Programming', \
['Campbell', 'Gries', 'Montojo'], \
'Pragmatic Bookshelf', \
'978-1-93778-545-1', \
25.0)
>>> python_book.num_authors()
3
"""
return len(self.authors)
```

b) Explain in detail about Encapsulation,Inheritance and Polymorphism:

Encapsulation

To *encapsulate* something means to enclose it in some kind of container. Inprogramming, *encapsulation* means keeping data and the code that uses it in one place and hiding the details of exactly how they work together. For example, each instance of class file keeps track of what file on the disk it is reading or writing and where it currently is in that file. The class hides the details of how this is done so that programmers can use it without needingto know the details of how it was implemented.

Polymorphism

Polymorphism means “having more than one form.” In programming, it means that an expression involving a variable can do different things depending on the type of the object to which the variable refers. For example, if obj refers to a string, then obj[1:3] produces a two-character string. If obj refers to a list, onthe other hand, the same expression produces a two-element list. Similarly,the expression left + right can produce a number, a string, or a list, depending on the types of left and right.

Polymorphism is used throughout modern programs to cut down on the amount of code programmers need to write and test. It lets us write a generic function to count nonblank lines:

```
def non_blank_lines(thing):
"""Return the number of nonblank lines in thing."""
count = 0
for line in thing:
if line.strip():
count += 1
return count
```

And then we can apply it to a list of strings, a file, a web page on a site

Inheritance

Giving one class the same methods as another is one way to make them polymorphic, but it suffers from the same flaw as initializing an object’s instance variables from outside the object. If a programmer forgets just one line of code, the whole program can fail for reasons that will be difficult to track down. A better approach is to use a third fundamental feature of objectoriented programming called *inheritance*, which allows you to recycle code in yet another way.

Whenever you create a class, you are using inheritance: your new class automatically inherits all of the attributes of class object, much like a

```
class Member:
""" A member of a university. """
def __init__(self, name, address, email):
""" (Member, str, str, str) -> NoneType
Create a new member named name, with home address and email address.
"""
self.name = name
self.address = address
self.email = email
class Faculty(Member):
""" A faculty member at a university. """
def __init__(self, name, address, email, faculty_num):
""" (Member, str, str, str, str) -> NoneType
Create a new faculty named name, with home address, email address,
faculty number faculty_num, and empty list of courses.
"""
super().__init__(name, address, email)
self.faculty_number = faculty_num
self.courses_teaching = []
```

report erratum • discuss

Subject :Python Programming
Subject Code:16MCA21

A Little Bit of OO Theory • 285

```
class Student(Member):
""" A student member at a university. """
def __init__(self, name, address, email, student_num):
""" (Member, str, str, str, str) -> NoneType
Create a new student named name, with home address, email address,
student number student_num, an empty list of courses taken, and an
empty list of current courses.
"""
super().__init__(name, address, email)
self.student_number = student_num
self.courses_taken = []
self.courses_taking = []
```

6 a) Explain short notes on GUI and event driven programming:

GUI Programming

Programs with a graphical user interface (GUI) are quite different from programs that use terminal input and output. In a terminal-based program, the computation is drived by the program: It does its job, sometimes printing some output, sometimes requesting input from the user and pausing to wait for this input.

In a GUI program, the user is much more in control of the program: There are buttons to be pressed, the window can be moved, the menu activated, or an animation may be playing. This requires a different style of programming: Instead of following a fixed sequence of actions, a GUI program reacts to events. There are many different kinds of events: the user may click with the mouse, move the mouse button, press some key, close the window, move or resize the window, and so on.

At first sight, it's therefore a bit difficult to figure out what is happing in a GUI program. There are always two steps: First, we need to set up the windows and configure their contents. Then, the program doesn't do anything actively — it only acts by responding to events. These responses to events have been set up in the first phase.

Event-driven programming:

A style of coding where a program's overall flow of execution is dictated by events.

- The program loads, then waits for user input events.
- As each event occurs, the program runs particular code to respond.
- The overall flow of what code is executed is determined by the series of events that occur
- Contrast with application- or algorithm-driven control where program expects input data in a pre-determined order and timing
  - Typical of large non-GUI applications like web crawling, payroll, batch simulation
- **event**: An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.
- **listener**: An object that waits for events and responds to them.
  - To handle an event, attach a *listener* to a component.
  - The listener will be notified when the event occurs (e.g. button click).

b) Explain about TKinter module and write a basic GUI program and explain line by line

Most modern programs interact with users via a *graphical user interface*, or GUI, which is made up of windows, menus, buttons,and so on. Python uses module called tkinter. A traditionally structured program usually has control over what happens when, but anevent-driven program must be able to respond to input at unpredictable moments.

tkinter is one of several toolkits one can use to build GUIs in Python. It is the only one that comes with a standard Python installation.

Every tkinter program consists of these things:

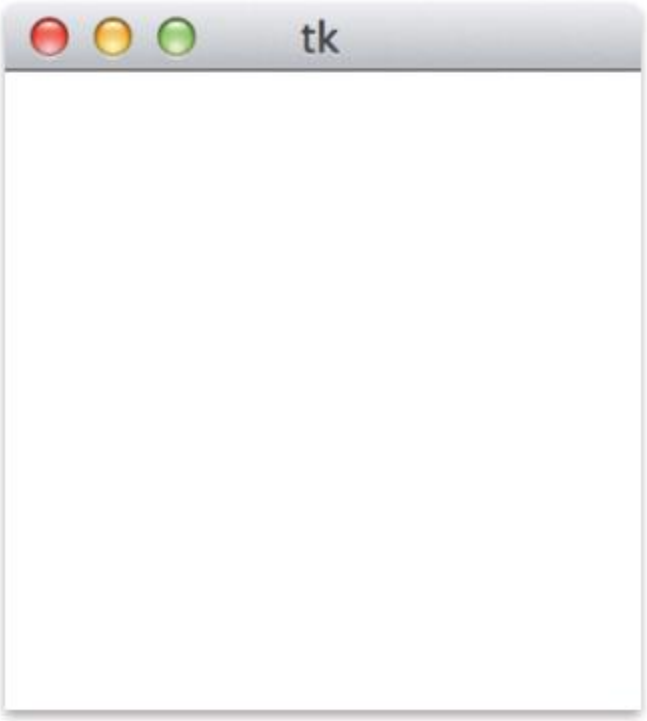
• Windows, buttons, scrollbars, text areas, and other *widgets*—anything that you can see on the computer screen (Generally, the term *widget* means any useful object; in programming, it is short for "window gadget.")

• Modules, functions, and classes that manage the data that is being shown in the GUI—• An event manager that *listens* for events such as mouse clicks and keystrokes and reacts to these events by calling event handler functions

Here is a small but complete tkinter program:

```
import tkinter
window = tkinter.Tk()
window.mainloop()
```

Tk is a class that represents the *root window* of a tkinter GUI. This root window's mainloop method handles all the events for the GUI, so it's important to create only one instance of Tk.



The root window is initially empty. If the window on the screen is closed, the window object is destroyed (though we can create new root window by calling Tk() again). All of the applications we will create have only one root window, but additionalwindows can be created using the TopLevel widget.
The call on method mainloop doesn't exit until the window is destroyed (which happens when you click the appropriate widget in the title bar of the window), so any cde following that call won't be executed until later

6 a) Explain about various tkinter widgets

Widget classes

Tkinter supports 15 core widgets:

Button

A simple button, used to execute a command or other operation.

Canvas

Structured graphics. This widget can be used to draw graphs and plots, create graphics editors, and to implement custom widgets.

Checkbutton

Represents a variable that can have two distinct values. Clicking the button toggles between the values.

Entry

A text entry field.

Frame

A container widget. The frame can have a border and a background, and is used to group other widgets when creating an application or dialog layout.

Label

Displays a text or an image.

Listbox

Displays a list of alternatives. The listbox can be configured to get radiobutton or checklist behavior.

Menu

A menu pane. Used to implement pulldown and popup menus.

Menubutton

A menubutton. Used to implement pulldown menus.

Message

Display a text. Similar to the label widget, but can automatically wrap text to a given width or aspect ratio.

Radiobutton

Represents one value of a variable that can have one of many values. Clicking the button sets the variable to that value, and clears all other radiobuttons associated with the same variable.

Scale

Allows you to set a numerical value by dragging a "slider".

Scrollbar

Standard scrollbars for use with canvas, entry, listbox, and text widgets.

Text

Formatted text display. Allows you to display and edit text with various styles and attributes. Also supports embedded images and windows.

Toplevel

A container widget displayed as a separate, top-level window.

All these widgets provide the Misc and geometry management methods, the configuration management methods, and additional methods defined by the widget itself. In addition, the Toplevel class also provides the window manager interface. This means that a typical widget class provides some 150 methods.

Mixins

The Tkinter module provides classes corresponding to the various widget types in Tk, and a number of mixin and other helper classes (a *mixin* is a class designed to be combined with other classes using multiple inheritance). When you use Tkinter, you should never access the mixin classes directly.

Implementation mixins

The Misc class is used as a mixin by the root window and widget classes. It provides a large number of Tk and window related services, which are thus available for all Tkinter core widgets. This is done by *delegation*; the widget simply forwards the request to the appropriate internal object.

The Wm class is used as a mixin by the root window and Toplevel widget classes. It provides window manager services, also by delegation.

Using delegation like this simplifies your application code: once you have a widget, you can access all parts of Tkinter using methods on the widget instance.

Geometry mixins

The Grid, Pack, and Place classes are used as mixins by the widget classes. They provide access to the various geometry managers, also via delegation.

Grid

The grid geometry manager allows you to create table-like layouts, by organizing the widgets in a 2-dimensional grid. To use this geometry manager, use the grid method.

Pack

The pack geometry manager lets you create a layout by "packing" the widgets into a parent widget, by treating them as rectangular blocks placed in a frame. To use this geometry manager for a widget, use the pack method on that widget to set things up.

Place

The place geometry manager lets you explicitly place a widget in a given position. To use this geometry manager, use the place method.

Widget configuration management

The Widget class mixes the Misc class with the geometry mixins, and adds configuration management through the cget and configure methods, as well as through a partial dictionary interface.

b) Explain tkinter based python program for creating a GUI that has a label,Entry and a button. The values given in Entry field should be updated in Label on click of the button

```
import tkinter

def cross(text,label):
    text.insert(tkinter.INSERT, 'X')

window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
label = tkinter.Label(frame, text='Name')
label.pack(side='left')
entry = tkinter.Entry(frame)
entry.pack(side='left')
button = tkinter.Button(frame, text='Add', command=lambda: copy(text,label))
button.pack()

window.mainloop()

def cross(text):
text.insert(tkinter.INSERT, 'X')
window = tkinter.Tk()
frame = tkinter.Frame(window)
frame.pack()
text = tkinter.Text(frame, height=3, width=10)
text.pack()
button = tkinter.Button(frame, text='Add', command=lambda: cross(text))
button.pack()
window.mainloop()

counter = tkinter.IntVar()
counter.set(0)
```

```
# The views.
frame = tkinter.Frame(window)
frame.pack()
button = tkinter.Button(frame, text='Click', command=click)
button.pack()
label = tkinter.Label(frame, textvariable=counter)
label.pack()
# Start
```

7a) Explain about Special class Attributes

There are four special class attributes:
'__dict__',
'__module__',
'__qualname__',
'__weakref__'

Every class that you have defined contains these four attributes, plus several more.The first one, __dict__, unsurprisingly refers to a dictionary.
Whenever you assign to an instance variable, it changes the contents of the object's dictionary. You can even change it yourself directly,
Variable __module__ refers to the module object in which the class of the object was defined.
Variable __weakref__ is used by Python to manage when the memory for an object can be reused.
Variables __name__ and __qualname__ refer to strings containing the simple and fully qualified names of classes, respectively; their values are usually identical, except when a class is defined inside another class, in which case the fully qualified name contains both the outer class name and the inner class name.
Variable __class__ refers to an object's class object. There are several more special attributes, and they are all used by Python to properly manage information about a program as it executes.

b) Write python code for creating Member and Faculty and student class such that Faculty and students are subclass of Member

```
class Member:
""" A member of a university. """
def __init__(self, name, address, email):
""" (Member, str, str, str) -> NoneType
Create a new member named name, with home address and email address.
"""
self.name = name
self.address = address
self.email = email
class Faculty(Member):
""" A faculty member at a university. """
def __init__(self, name, address, email, faculty_num):
""" (Member, str, str, str, str) -> NoneType
Create a new faculty named name, with home address, email address,
faculty number faculty_num, and empty list of courses.
"""
super().__init__(name, address, email)
self.faculty_number = faculty_num
self.courses_teaching = []
```

report erratum • discuss

A Little Bit of OO Theory • 285

```
class Student(Member):
""" A student member at a university. """
def __init__(self, name, address, email, student_num):
""" (Member, str, str, str, str) -> NoneType
Create a new student named name, with home address, email address,
student number student_num, an empty list of courses taken, and an
empty list of current courses.
"""
super().__init__(name, address, email)
self.student_number = student_num
self.courses_taken = []
self.courses_taking = []
```

8 a) Write python code for inverting a dictionary:

```
>>> bird_to_observations
{'canada goose': 5, 'northern fulmar': 1, 'long-tailed jaeger': 2,
'snow goose': 1}
```

```
>>>
>>> # Invert the dictionary
>>> observations_to_birds_list = {}
>>> for bird, observations in bird_to_observations.items():
... if observations in observations_to_birds_list:
... observations_to_birds_list[observations].append(bird)
... else:
... observations_to_birds_list[observations] = [bird]
...
>>> observations_to_birds_list
{1: ['northern fulmar', 'snow goose'], 2: ['long-tailed jaeger'],
5: ['canada goose']}
```

The program above loops over each key/value pair in the original dictionary, bird_to_observations. If that value is not yet a key in the inverted dictionary, observations_to_birds_list, it is added as a key and its value is a single-item list containing the key associated with it in the original dictionary. On the other hand, if that value is already a key, then the key associated with it in the original dictionary is appended to its list of values.

Now that the dictionary is inverted, you can print each key and all of the items in its value list:

```
>>> # Print the inverted dictionary
... observations_sorted = sorted(observations_to_birds_list.keys())
>>> for observations in observations_sorted:
... print(observations, ':', end=" ")
... for bird in observations_to_birds_list[observations]:
... print(' ', bird, end=" ")
... print()
...
1 : northern fulmar snow goose
2 : long-tailed jaeger
5 : canada goose
```

The outer loop passes over each key in the inverted dictionary, and the inner loop passes over each of the items in the values list associated with that key.

b) What is the output of the following code ?

```
1. class tester:
2.     def __init__(self, id):
3.         self.id = str(id)
4.         id="224"
5.
6. >>>temp = tester(12)
>>>print temp.id
```

Output:12