ICT162
OBJECT ORIENTED PROGRAMMING

STUDY GUIDE (5CU)

Course Development Team

Head of Programme	:	A/P Paul Wu Horng Jyh
Course Developer(s)	:	Ms Jenny Lim Eng Lian

Educational Technology & Production
Singapore University of Social Sciences
463 Clementi Road
Singapore 599494

Release V1.0

Table of Contents

Course Guide

1. Welcome

Presenter: Ms Lim Eng Lian Jenny

This streaming video requires Internet connection.
Access it via Wi-Fi to avoid incurring data charges on your personal mobile plan.

Click here to watch the video

Welcome to the course *ICT 162 Object-Oriented Programming*, a 5 credit unit (CU) course.

This Study Guide will be your personal learning resource to take you through the course learning journey. The guide is divided into two main sections – the Course Guide and Study Units.

The Course Guide describes the structure for the entire course and provides you with an overview of the Study Units. It serves as a roadmap of the different learning components within the course. This Course Guide contains important information regarding the course learning outcomes, learning materials and resources, assessment breakdown and additional course information.

2. Course Description and Aims

This course covers further concepts in object-oriented programming. It explains the basic building blocks of an object. Students learn how to apply object structure and methods to store and compute tabular information as a system of objects. The principles and reasons of structuring objects in a class hierarchy and association will be explained. A particular class, the container class, is introduced. Students will learn how complex processing mechanisms can be programmed through the container class. These complex mechanisms are then shown to be the building blocks for Graphical User Interface and Event Management, which have become a standard approach in building native software applications.

Course Structure

This course is a 5-credit unit course presented over 6 weeks.

There are six Study Units in this course. The following provides an overview of each Study Unit.

Study Unit 1 – Introduction to Object-Oriented Programming

This study unit introduces the object-oriented programming (OOP) paradigm, the concept of classes and objects and the main pillars of OOP which are encapsulation, inheritance and polymorphism.

The OOP paradigm is a means to structure programs via bundling related pieces of data and operations into a class – a data type with operations defined. Objects are instances of the class. Each object maintains its own set of data and can perform the operations defined in the class.

There are relationships between classes, of which two are elaborated, namely, object composition and inheritance. Object composition is a has-a relationship where objects of a class are composed of other objects. Inheritance is an is-a relationship where objects of one class are specialized objects of some classes. Object composition and inheritance are means of reducing code duplication.

The study unit also covers defining or writing classes using object and inheriting from only the class `object`. Study Unit 2 will focus on inheritance from user defined classes.

As this unit covers chapters 1, 6 and 7 of the textbook, it is estimated that the student will spend about 8 hours to read the textbook chapters in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Study Unit 2 – Inheritance

Classes in Python 3 are new style classes. This means the classes that do not specify a superclass inherit from the class `object` implicitly.

In an inheritance relationship, subclasses inherit both instance and class variables as well as instance and class methods, in particular, the class method `__new__` in the class `object` which is invoked whenever an object is created. However, you will see that inheritance in Python is somewhat different as instance variables are not inherited.

This study unit focuses on defining classes that specify one or more superclasses explicitly. It covers how a subclass can augment its superclass through defining new instance or class variables and methods, and how a subclass can specialise its superclass by redefining the implementation of the methods inherited from the superclass. Methods can be specialised in two ways: overriding by refinement and overriding by replacement. The `super` keyword is introduced and is differentiated from the keyword `self`.

Python allows subclasses to inherit from multiple superclasses, and it applies the Method Resolution Order (MRO) to resolve which methods of superclasses are bounded to a method call. The study unit elaborates on multiple inheritance and illustrates the MRO for Python 3.

This study unit also covers abstract superclass and abstract methods. Abstract methods allow us to apply the template method design pattern, a design pattern that applies polymorphism.

As this unit covers chapters 9 of the textbook as well as other resources, it is estimated that the student will spend about 6 hours to read the textbook chapters and resources, in conjunction with the study notes, to work out the activities and self-assessment

questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Study Unit 3 – Object Composition with Collections

Chapter 1 of this study unit focuses on the management of a mutable collection of objects. The discussion is in 3 parts – when objects in the collection are objects from a single class, when objects in the collection are objects from a concrete superclass and its subclasses and finally, when objects in the collection are objects from subclasses of an abstract superclass.

Each part covers the standard services for collection management and each part builds on demonstrating a variety of services involving the collection.

Chapter 2 covers two design patterns, the template method pattern and strategy pattern, applicable to inheritance and object composition which are two of the pillars of object-oriented programming. In particular, the patterns are demonstrated and differentiated.

This study unit does not cover any chapter of the textbook. However, as there are several online resources to read, it is estimated that the student will spend about 6 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Study Unit 4 – Exception Handling

This study unit focuses on handling runtime errors which occur whenever the runtime system or the interpreter is unable to successfully execute program statements. A runtime error is also called an exception. There are two ways of dealing with exception: preventive or defensive programming and recovery or handling exception after it has been raised. The latter is the focus of this study unit.

To recover from an exception after it has been raised requires special language constructs to route the program execution to statements that handle the exception. In particular, the study unit elaborates on the `try-except-else-finally` construct and the `with` statement provided in the Python language.

The study unit also demonstrates how to implement a user-defined exception class. Having user-defined exception classes allows an application to distinguish between exceptions raised from business rule violation and from semantically invalid program statements.

When a program detects that a business rule is violated, it can raise exceptions. Thus study unit covers the four ways that a program can raise exceptions from user-defined exception classes. The study guide also discusses the various scenarios of handling the exception in an application.

This study unit covers chapter 5, section 5.5 of the textbook. As there are also several online resources to read, it is estimated that the student will spend about 7 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Study Unit 5 – Graphical Interface

The focus of this study unit is graphical user interface (GUI) application using tkinter which is bundled with Python. tkinter is a Python interface to Tk, an open source cross platform widget toolkit.

The different types of widgets that make up a GUI and the various layout managers that position widgets are first discussed. The study unit also demonstrates how the different types of widgets are created and how layout managers are used to position them.

The study unit covers event handling. User actions on the GUI generate events. To be able to respond to these events, the GUI application must include event-handling code, that is, code that will execute under certain user actions. The study unit discusses two ways of binding event handling code to widgets that generate events

The study unit finally discusses how object-oriented programming techniques are applied to GUI application, to modularize the implementation and to separate the various programming concerns. In particular, the study unit illustrates the model-view-controller (MVC) framework, which separates the implementation of the data (model) from the implementation of various views of the data, and allows the data and the view to communicate indirectly through the controller l.

This study unit covers chapter 15 of the textbook. As there are also several online resources to read, it is estimated that the student will spend about 7 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Study Unit 6 – OO Design Principles

Poor object-oriented program designs cause programs to be fragile, rigid and immobile. This study unit covers five software design principles, SOLID to overcome these problems.

The SOLID principles are first described and then illustrated using three design patterns: the template method pattern, the strategy pattern and the model-view-controller pattern.

This study unit does not cover any chapter of the textbook. However, as there are several online resources to read, it is estimated that the student will spend about 6 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

3. Learning Outcomes

Knowledge & Understanding (Theory Component)

By the end of this course, you should be able to:

- Describe the structure of objects: attributes and methods
- Use objects to store and compute tabular information
- Demonstrate how class hierarchy and association can be used to organize information
- Apply the principles of object-oriented programming principles in designing and developing applications

Key Skills (Practical Component)

By the end of this course, you should be able to:

- Construct the class hierarchy and association according to specification
- Develop GUI for an application based on user requirements

4. Learning Materials

The following is a list of the required learning materials to complete this course.

Required Textbook(s)

Goldwasser, M. H., & Letscher, D. (2014). *Object-Oriented Programming in Python*
Upper Saddle River, N.J: Pearson Prentice Hall.

Other recommended study material (Optional)

The following learning materials may be required to complete the learning activities:

Website(s):

NILL

5. Assessment Overview

The overall assessment weighting for this course is as follows:

Assessment	Description	Weight Allocation
Assignment 1	Online Quiz	12%
Assignment 2	Tutor Marked Assignment	18%
Examination	ECA	70%
TOTAL		100%

The following section provides important information regarding Assessments.

Continuous Assessment:

There will be continuous assessment in the form of 1 quiz (QUIZ) and 3 pre-class quizzes (PCQ) totaled 40%. One tutor-marked assignment (TMA) totaled 60%. In total, this continuous assessment will constitute 30% of overall student assessment for this course. The three assignments are compulsory and are non-substitutable. These assignments will test conceptual understanding of both the fundamental and more advanced concepts and applications that underlie object-oriented programming. It is imperative that you read through your Assignment questions and submission instructions before embarking on your Assignment.

Examination:

The final (2-hour) written exam will constitute the other 70 percent of overall student assessment and will test the ability to marketing related concepts, theories and strategies to particular situations commonly faced by marketing managers. All topics covered in the course outline will be examinable. To prepare for the exam, you are advised to review Specimen or Past Year Exam Papers available on Learning Management System.

Passing Mark:

To successfully pass the course, you must obtain a minimum passing mark of 40 percent for both of the QUIZ and TMA components. That is, students must obtain at least a mark of 40 percent for the combined assessments and also at least a mark of 40 percent for the final exam. For detailed information on the Course grading policy, please refer to The Student Handbook ('Award of Grades' section under Assessment and Examination Regulations). The Student Handbook is available from the Student Portal.

Non-graded Learning Activities:

Activities for the purpose of self-learning are present in each study unit. These learning activities are meant to enable you to assess your understanding and achievement of the learning outcomes. The type of activities can be in the form of Quiz, Review Questions, Application-Based Questions or similar. You are expected to complete the suggested activities either independently and/or in groups.

6. Course Schedule

To help monitor your study progress, you should pay special attention to your Course Schedule. It contains study unit related activities including Assignments, Self-assessments, and Examinations. Please refer to the Course Timetable in the Student Portal for the updated Course Schedule.

Note: You should always make it a point to check the Student Portal for any announcements and latest updates.

7. Learning Mode

The learning process for this course is structured along the following lines of learning:

(a) Self-study guided by the study guide units. Independent study will require ***at least 3 hours per week.***

(b) Working on assignments, either individually or in groups.

(c) Classroom Seminar sessions (3 hours each session, 6 sessions in total).

(d) Online Office hours (2 hours each session, 6 sessions in total

iStudyGuide

You may be viewing the iStudyGuide version, which is the mobile version of the Study Guide. The iStudyGuide is developed to enhance your learning experience with interactive learning activities and engaging multimedia. Depending on the reader you are using to view the iStudyGuide, you will be able to personalise your learning with digital bookmarks, note-taking and highlight sections of the guide.

Interaction with Instructor and Fellow Students

Although flexible learning – learning at your own pace, space and time – is a hallmark at SUSS, you are encouraged to engage your instructor and fellow students in online discussion forums. Sharing of ideas through meaningful debates will help broaden your learning and crystallise your thinking.

Academic Integrity

As a student of SUSS it is expected that you adhere to the academic standards stipulated in The Student Handbook, which contains important information regarding academic policies, academic integrity and course administration. It is necessary that you read and understand the information stipulated in the Student Handbook, prior to embarking on the course.

STUDY UNIT 1

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Learning Outcomes

By the end of this unit, you should be able to:

1. Differentiate between class and object, reuse by object composition and by inheritance, class level and object level members
2. Define encapsulation, object composition and inheritance, polymorphism
3. Illustrate encapsulation, object composition and inheritance, polymorphism in OO design
4. Explain the keyword self and the double underscores and discuss the purpose of constructor
5. Implement object level members including constructors, accessors, mutators and str method, class level members including those for generating running numbers and a class adhering to Python convention
6. Apply method ”overloading”
7. List good software practices for defining classes

Overview

This study unit introduces the object-oriented programming (OOP) paradigm, the concept of classes and objects and the main pillars of OOP which are encapsulation, inheritance and polymorphism.

The OOP paradigm is a means to structure programs via bundling related pieces of data and operations into a class – a data type with operations defined. Objects are instances of the class. Each object maintains its own set of data and can perform the operations defined in the class.

There are relationships between classes, of which two are elaborated, namely, object composition and inheritance. Object composition is a has-a relationship where objects of a class are composed of other objects. Inheritance is an is-a relationship where objects of one class are specialized objects of some classes. Object composition and inheritance are means of reducing code duplication.

The study unit also covers defining or writing classes using object and inheriting from only the class `object`. Study Unit 2 will focus on inheritance from user defined classes.

As this unit covers chapters 1, 6 and 7 of the textbook, it is estimated that the student will spend about 8 hours to read the textbook chapters in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 The Object Oriented Paradigm

1.1 Classes, Objects and Encapsulation

A computation is an operation carried out on a computer on some data. Two forms of abstraction related to computation are data abstraction and operation abstraction.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 5-11.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

1.1.1 Data Abstraction and Operation Abstraction

- Data Abstraction

  Data, no matter simple or complex, is first encoded and then stored in memory as a sequence of bits.

  Different encoding schemes use different bit patterns to represent the same data in memory. For example, the bit pattern for a street map is different for different encoding schemes such as gif, jpg or bmp encoding (or format). Each encoding scheme uses a decoding scheme to manipulate the bit sequence when data is processed, such as when a pixel in an image is modified.

  High level languages provide encoding schemes for program data through built-in data types such as `int` for whole numbers or as `str` for a sequence of characters. Data is encoded when stored and decoded when processed.

  High level languages thus take away the low level concerns about how to encode and decode data in memory. Instead of looking at a piece of data as sequence of bits in memory in a certain encoding, a program can view data according to its type. High level languages thus provide data abstraction to programmers.

  Furthermore, many high level languages do not restrict programmers to only those data types built into the language but also allow programmers to define custom data types. The custom data types are called user-defined types and they provide programmers another level of data abstraction. An example of a custom data type is an employee record type. This custom data type is made

up of several fields such as employee number (`str`), name (`str`), performance (`int`) and salary (`float`).

- Operation Abstraction

  Instructions to the CPU are at hardware level, and the set of instructions available to programs are dependent on and limited by the architecture of the underlying machine. Thus, simple operations such as adding two numbers can require several machine instructions including those to set up the data for the CPU to operate on and those for storing data (result) from the CPU back to memory.

  To relieve programmers of the tedium of mapping high level operations to primitive instructions dependent on the machine architecture, high level languages introduce operators such as the assignment operator, and control structures such as decision structures and loop structures, Supporting software such as compilers and interpreters then map the high level constructs into machine instructions.

  Many high level languages do not restrict programmers to only the built-in operators and high level constructs in the language but also allow them to define custom or user-defined operations.

  Such operations are defined as functions and they provide another level of operation abstraction. An example of a user-defined operation is a function `calculateBonus(performance, cur_sal)` which computes the bonus of an employee given his performance and current salary.

Hence, as can be seen, high level languages provide both abstraction of data and abstraction of operations, allowing programmers to focus on the programming problem at hand without concern about the underlying hardware architecture.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 12-16.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

1.1.2 Class

The object-oriented programming is a paradigm shift. Instead of abstracting at the data level via custom data types and abstracting at the operation level via function definition, an object-oriented programming language provides for the abstraction of data and operation levels simultaneously via a **class**.

A **class** abstracts both data and operations. A class is a custom data type with custom operations.

In non-object-oriented programming languages such as C, there is no notion of class, only the notion of data type which is simply data abstraction and the notion of function which is simply operation abstraction.

In some object-oriented programming languages such as Java, there is a distinction between (primitive) data types that are simply data abstractions and classes which are both data and operations abstraction.

In other object-oriented programming languages such as Python, all data types are classes, that is, in Python, all data types are abstraction of both data and operations, and so, in Python, the terms data types and classes are synonymous.

The motivation for object-oriented programming is real world objects and how they interact cooperatively to provide services to one another.

- Real world objects are interdependent, each with responsibilities assigned to them, to provide specific service to other objects.
    - The type of an object determines its responsibilities.

      For example, a cashier can perform the service `checkout`, and a waiter provides the services `takeOrder` and `serveFood`.
    - To request for a service is to request an object to invoke its operation. Standard protocol of requesting for services are set in place, according to the interface of type of objects.

      For example to request for the service `checkout`, the operation `checkout` is invoked on a cashier object and the invocation requires a collection of one or items.
- Real world objects have standard operating procedure for the services
    - No matter how different the cashier objects are (e.g., different name and age), they are equipped with the standard operating procedures for their services.
    - The implementation of the services or the operation that an object uses to provide services is treated as a black box.

      The requester making the service does not depend on the implementation of the service as the implementation may change. Rather, the requester simply invokes the service using the specific protocol, sending the service provider the required input.

Furthermore, the service provider is expected to perform the operation according to the standard operating procedure and to return output to the requester.

- Real world objects own data
    - To perform a service, an object may use data that it owns.

      For example, if we look at our receipt after a checkout session, we will notice that not only are checkout items listed but also are the employee number and name of the cahier providing the service.

In object-oriented programming, a class can model a type of real world objects, concepts or events etc. Objects can be tangible objects such as cashier and items bought or intangible objects such as a purchase or an order.

Classes for objects that maintain data of the application are also called entity classes.

1.1.3 Object

A class specifies a *type of object* in terms of the object's attributes (data it owns) and the object's behaviour (methods or operations it can perform).

The class Employee specifies a type of object (people who work for others) in terms of the Employee object's attributes and the object's behaviour. The class Employee describes

- the attributes that an Employee object has

  Examples of attributes for an Employee object are employee number (`str`), name (`str`), performance (`int`) and salary (`float`). We say the Employee class maintains data for employee number, name, performance and salary.

  Object attributes record data of an individual object. Thus, each Employee object has its own copy of the attributes to record its employee number, name, performance and salary.

  For example, employee Sam has employee number 126, name Sam, performance level 1 and salary $2500. A different employee, say, Ann has a different employee number 123, name Ann, performance level 4 and posbily, the same salary $2500. Updating Ann's salary will change the salary attribute for Ann but does not affect the salary of Sam.

- the behaviour that an Employee object has or the methods it can perform, e.g., the methods `calculateBonus()` and `increaseSalaryBy()`.

  For example, employees Sam and Ann can perform the calculation for their bonuses and salary increment based on their personal attributes such as salary and performance.

Attributes and behaviour are also called **data members** and **member functions** respectively. However, we will use the terms attributes and methods for behaviour.

We *create object* from a class to represent a real world object modelled by that class. Say, Peter is someone who works for others, i.e., Peter is an employee. To represent Peter in our application, an object with the reference `peter` is created from the class Employee.

For example,

```
peter = Employee('Peter')
```

A side note: creating an object from any built-in type such as `list` and referencing the object e.g., by `myList`, has the same syntax as for creating an object from a user-defined type such as `Employee`:

```
>>> myList = list(range(1, 10, 2))
>>> myList
[1, 3, 5, 7, 9]
```

An **object**

- is an occurrence or an **instance** of a class

  The object `peter` is an object or an instance of the class Employee.

- has all the instance attributes (or instance variables, in programming terminology) defined in the class it is created from.

  `peter` has employee number (`str`), name (`str`), performance (`int`) and salary (`float`).

- can perform instance methods defined in the class it is created from.

  `peter` can perform the methods `calculateBonus()` and `increaseSalaryBy()`.

Objects represent real world objects. The attributes of objects store the data owned by real world objects, and the methods of objects enable them to perform their responsibilities as real world objects that interact cooperatively. That is to say:

- Objects defined by a class are interdependent, each with methods to provide service to other objects.
  - The type (or class) of object determines its methods.

    For example, a Cashier object can perform the method `checkout()` if the class Cashier defines the method `checkout()`.
  - To request for a service is to invoke a method.

    To invoke a method, we need to know its **method signature** - the method name, the formal parameters for input data and the returned values.

    For example to request for checkout service, the method `checkout()` is invoked with an actual parameter – a collection of items to checkout.

    Formal and actual parameters are terminologies you saw in function definitions and function calls in ICT133.

- Objects defined by a class have standard procedure for performing the methods

  The implementation of the methods that an object uses to provide services should be treated as a black box as the implementation of the services may change. All the caller should be concerned about is the method signatures.

.

- Objects defined by a class have their own data

  An object may use data that it owns when it executes a method.

Activity 1

A class that might be in a car rental system is the class Customer. List 3 attributes and 3 methods for the class Customer.

1.1.4 Implementing Object-oriented Solutions

To implement object-oriented programming solutions, we first perform object-oriented *analysis* and *design* to identify classes of objects and to discover what methods and attributes they must have.

In the problem analysis phase, we identify the classes by analyzing a problem. Typical objects in a supermarket application include Item objects, Payment objects, Cashier objects and possibly, Sales Promotion objects. This means that correspondingly, the application requires Item, Payment, Cashier and Sales Promotion classes.

In the problem analysis phase, we also identify what attributes objects in each class own.

Activity 2

List 3 other classes that might be in a car rental system.

In the design phase, we propose how the objects will interact by calling for services in the various scenarios of the problem such as during a checkout, an exchange or a refund. We also assign responsibilities the objects have based on their attributes.

The interactions are documented in sequence diagrams, one of such is reproduced from Figure 1.4 in the textbook, shown here as in Figure 1.1.

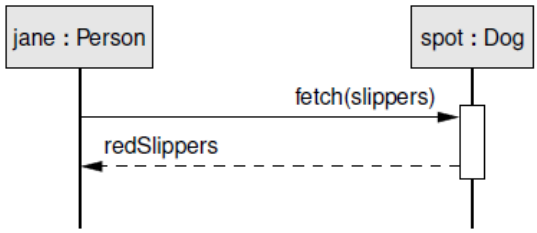


Figure 1.1 Sequence diagram in which jane invokes spot,fetch(slipper)
(Source: Figure 1.4 of textbook)

The sequence diagram shows a sequence of messages that objects send to one another to collaborate so that a function of the application is realised. Each message is a call for service. A message may include actual arguments such as `slipper` for the `fetch` message, and the method may return value such as `redSlippers` in this example.

Based on the service calls in the various sequence diagrams for the various scenarios of the problem, we determine the services that each class provides, and in turn, the methods the class must implement. For example, the class `Dog` provides a `fetch` service so the class `Dog` must implement the method `fetch`.

Next the signature of each method is designed. We determine the input it requires from caller and the output it produces for the caller. The method signatures collectively form the **interface** of the class.

The interface is disclosed to other classes so that service calls can be correctly made, e.g., to call for the service `fetch` to invoke the method `fetch`, the caller needs to supply an actual parameter: what to fetch. The caller can expect to receive an output: the fetched object.

The data required in a method therefore comes from two sources:

- The caller of the service

  The data must be supplied as an actual parameter in the call, e.g., `slipper`.

- The object itself that is performing the service

  The data must be an attribute of the object itself.

  For example, besides being told what to fetch (`slipper` as supplied by caller `jane`), possibly `spot` must also know its current `location`. If `spot`'s `location` is Jane's house, it could fetch Jane's slippers from her room. If `spot`'s `location` is elsewhere, the operation may fail to return slippers.

  Thus, if we had overlooked the attribute `location` at analysis phase, we will uncover it during design phase and this necessitates the refinement of the problem specifications and requirements, after consultation with the end-user.

During the design phase, it is possible that we uncover also attributes that a class must have based on the services it has to provide and the other objects it must request service from to fulfill a service. For example, we will uncover the attribute `pet` for the class `Person` at design time as `jane` needs to request a service from her own `pet`, the `Dog` object, `spot`. That is, the class `Person` will have an attribute, `pet`, a `Dog` object.

Methods with their method signatures are established in the design phase, and the collaborations amongst objects are worked out and documented in sequence diagrams.

In the implementation phase,

- attributes of objects discovered at analysis and design phases are implemented as **instance variables**. We will use the term attribute and instance variable interchangeably.

  During implementation, `name` and `employeeNum` are defined as instance variables of the class `Cashier`, `location` as an instance variable of the class `Dog` and `pet` as an instance variable of the class `Person`.

- methods are implemented with statements to perform the service.

  During implementation, `checkout` and `exchange` are defined as instance methods of the class `Cashier` and `fetch` as instance method of the class `Dog`.

1.1.5 Encapsulation

A class should hide the implementation of both instance variables and methods by indicating them as private, but the class exposes its interface (that is, the collectively method signatures).

Information-hiding or hiding the *implementation details of* instance variables *and methods* from client code is an important feature of object oriented programming as objects should be treated as black boxes with known interfaces.

Information-hiding protects the client code from being impacted by changes in the supplier code. The client code can remain unchanged as long as the supplier‘s interface is unchanged. Therefore, components in the program solution can be changed as long as the interface remains the same. Information-hiding in Python is covered in chapter 2 of this study unit.

Another important concept in object oriented programming is **encapsulation**, a concept frequently confused with information-hiding. While information-hiding is about hiding the implementation details, encapsulation is the result of *abstracting related data and operations simultaneously and applying information-hiding*.

Encapsulation results in class definition; a class defines data and operation, applies information-hiding to wrap up implementation details so that its objects are black boxes. By knowing the interface or the method signatures of methods implemented in a class, the clients or collaborators can call an object of that class to perform a service on the object. The clients or collaborators need not be concerned or know about the implementation of the instance variables and methods.

Hiding implementation details means that the instance variables can be accessed only indirectly via service calls to invoke methods that provide *controlled access* to private instance variables.

These methods are called

- **accessor methods**
  Accessor methods allow inspection of instance variables and

- **mutator methods**
  Mutator methods allow values of instance variables to be changed.

In chapter 2, we use the `@property` decorator, a Pythonic way of implementing accessor and mutator methods.

Once classes are implemented, writing the application becomes a matter of translating series of service calls between objects in the various sequence diagrams for the different scenarios of the problem. Each service call is a method call to an object to execute a method.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 17-19.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

1.2 Relationships between Classes: Object Composition and Inheritance

The classes that we identify during the analysis phase can be related to one another in various ways, of which object composition and inheritance are two.

1.2.1 Has-A and Is-A relationships

Object composition is a *has-a* relationship between two classes.

For example, in a supermarket application, a payment object *has a* cashier object since a payment should record the cashier who facilitates the payment. Payment is

the container class and cashier is the contained class. Therefore, the Payment class has an *instance variable*, a cashier object from the Cashier class.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 20-23.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

Inheritance is an *is-a* relationship between two classes.

For example, in a supermarket application, there are different types of employees such as cashiers and store hands. A cashier *is an* employee. A store hand *is* also *an* employee. We have an is-a relationship between the Cashier class and the Employee class, and another is-a relationship between StoreHand class and Employee class.

Therefore, we define 3 classes of employees:

- The Employee class is the general class of employees.

  This class has instance variables and methods common to all employee e.g., attributes `name` and `employeeNumber` and methods `calculateBonus()` and `increaseSalaryBy()`.

- The Cashier class is a specialized type of the Employee class

  This class has instance variables and methods common to all employees such as `name` and `employeeNumber` and methods `calculateBonus()` and `increaseSalaryBy()`.

  In addition, it has instance variables and methods of cashiers e.g., the method `checkout()`.

- The StoreHand class is a specialized type of the Employee class

  This class has instance variables and methods common to all employees such as `name` and `employeeNumber` and methods `calculateBonus()` and `increaseSalaryBy()`.

  In addition, it has instance variables and methods of store hands e.g., the instance variable `products` to record products assigned to store hands and the method `updateStock()`.

An is-a relationship is modelled as an inheritance relationship. Note that as both cashiers and store hands are employees, albeit specialized, they can be modelled as

subclasses of the Employee class and inherit all instance variables and methods of the Employee class, the **superclass**. A subclass *is a* specialized class of a superclass. The Cashier class and the StoreHand class are subclasses of the Employee class.

Some languages such as Java allow for single inheritance, that is, each subclass can have only one superclass.

Python allows for multiple inheritances. For example, a specialized class of cashier, say CashierTrainer, for cashiers who also train new employees. The class CashierTrainer will have two is-a relationships, one is-a relationship with the Cashier class and another is-a relationship with the Trainer class. This example of multiple inheritance makes the class CashierTrainer is a subclass of both the Cashier class and the Trainer class.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 23-28.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Activity 3

Give one has-a relationship and one is-a relationship that might exist in a car rental system.

1.2.2 Reducing Code Duplication with Object Composition and Inheritance

Designing good classes takes practice. We look at design principles in Study Unit 6, and when implementing classes in Study Unit 2 and 3, design patterns and design principles are mentioned in passing.

Classes provide data and operation abstraction (encapsulation) and help structure the solution to a programming problem. However, incorrect implementations can give code duplication.

Code duplication is a big issue in software development as it increases maintenance effort. As such, programming languages provides many means to reduce code duplication such as:

- Function definition

Duplicated code for an operation is placed in a function definition so that the code now appears only once in the function body. All duplications of the code are replaced by a function call.

This means of reducing code duplication uses operation abstraction.

- Object composition

  When classes have a has-a relationship, the relationship is implemented as object composition. The contained object becomes an instance variable for the container object.

  For example, if a purchase has-a cashier, then an instance variable of the class Purchase will reference an object from the class Cashier.

  The container object can request for services from the contained object, e.g., to inspect or to change value of instance variables of the contained object and for any other services, as long as the services are provided by the contained object.

  By requesting for service, the container object does not need to duplicate statements in the definition of the class for the contained object.

  For example, a purchase object can request to inspect the name and employee number of its cashier object, and so, it does not need to duplicate these attributes nor provide accessor and mutator methods for them.

  This means of reducing code duplication uses object composition.

  However, a has-as relationship can be incorrectly implemented not apply object composition.

  For example,

  - Duplicating the instance variable of the contained class

    Making the name and employee number to be instance variables in the Purchase class instead of just an instance variable to reference the Cashier object.

  - Duplicating the code of the contained class

    Duplicating the accessor methods for name and employee number in the Purchase class instead of requesting for services from a Cashier object.

- Inheritance

When classes have an is-a relationships, the relationship is implemented as inheritance, that is, a subclass-superclass relationship using the notation of the language.

For example, if cashier is-a employee, then cashier should be a subclass of employee, and employee is a superclass of cashier.

A subclass inherits instance variables and methods of the superclass. In Python, the subclass will also inherit class variables and class methods. Class variables and methods are discussed in Section 2.2 of this study unit.

Inheriting instance variables and methods means that the subclass has instance variables and methods that are defined in its superclass, without having to define them. The subclass has them simply by being a subclass of the superclass.

This means of reducing code duplication uses inheritance.

Like a has-as relationship, an is-a relationship can also be incorrectly implemented not to apply inheritance, that is, a subclass-superclass relationship.

For example,

- Duplicating the instance variables of a superclass

  For example, making the name and employee number to be instance variables in the class Cashier instead of applying inheritance to inherit the instance variables, name and employee number, in the class Employee.

- Duplicating the methods of a general class

  Duplicating the methods `calculateBonus()` and `increaseSalaryBy()`in the class Cashier instead of inheriting the methods from the class Employee.

To avoid code duplication, the has-a and is-a relationships must be implemented as object composition and inheritance respectively.

1.3 Polymorphism

Polymorphism means many forms or many interpretations.

For example, consider the statement:

> Molly kicked the bucket

This statement can have many interpretations at different times, depending on who Molly currently is.

- if Molly is a person involved in a fatal accident

  The statement is interpreted as Molly has died.

- if Molly is a cow in a farm

  The statement is interpreted as Molly the cow kicked the bucket (probably while being milked).

Therefore, we must know the class of the object that Molly references, to discover which method should be interpreted as or bounded to the operation "kick the bucket".

In object-oriented programming, a polymorphic expression is a service request in this form:

```
objPerformingMethod.methodName(<parameter-list>)
```

The interpretation of a polymorphic expression depends on the *class* of the object referenced by `objPerformingMethod`.

Polymorphism is an important concept in object-oriented program as a variable in an expression can reference objects from different classes at different times, as you will see in Study Unit 3.

As long as those classes have the required interface or methods as specified in the expression, the Python interpreter will use the method defined in the class of the current object.

Recall in Study Unit 6 of ICT133, we learn that the Python interpreter applies dynamic binding. At runtime, the Python interpreter will know the data type (or class) of the object that receives a message (method call) as each object has an identity, a type and value(s). The Python interpreter then binds the method call to the method of the object's class (or type).

In dynamic binding, the search for the method starts at the class of the object receiving the message. If the method is located in that class, the method is bound as the method to execute. If the method cannot be located, the statement produces an error.

However, if the object is from a subclass, and the method cannot be located in that subclass, then the Python interpreter continues the search for the method in its superclasses and the superclasses of each superclass and so on, as methods and attributes of superclasses are inherited. If the method cannot be located in all of its superclasses, then the expression produces a runtime error.

Class, encapsulation, object composition, inheritance and polymorphism are key concepts that make a programming language object-oriented.

Chapter 2 Defining our Own Classes

2.1 Defining Classes with Object Composition

A class definition in Python has the following structure:

```
class <className> <(superclass)>:
      <body>
```

`class` is a keyword to start a class definition, and it is followed by a class name. The superclass of the class may optionally be specified, and if so, within parentheses.

Note that Python 3 uses new style classes. That means every class that you create inherits from `object` implicitly, unless you specify a different superclass. Subclasses that inherit from a different superclass is covered in Study Unit 2.

The `<body>` of a class consists of statements which form the specification of its objects, in terms of instance variables and methods, as well as class variables and methods which is covered in section 2.2.

The class `Employee` can be defined simply with only the `pass` statement, if it does not have other instance variables and methods other than those inherited from class `object`.

```
class Employee:
    pass
```

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 203-205.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

2.1.1 Constructor

A class definition in Python naturally uses object composition as its instance variables reference values which are objects, either from built-in or user-defined classes.

For example, if the class Employee has a name (a `str` object) and a salary (a `float` object), we say an Employee object is composed of a a `str` object (for name) and a `float` object for salary.

Instance variables referencing objects from user-defined classes is covered in Section 2.1.5.

All instance variables should be initialised in a special method called a constructor. The instance variables begin to exist when the assignment statements execute to initialise their values, consistent with the manner in which local variables come into existence.

A constructor is a special method to initialize the instance variables of a newly created object.

In Python, the constructor is given the name and method signature:

```
__init__(self <,parameter-list>).
```

where `<,parameter-list>` is optional.

A constructor does not return any value, that is, it returns `None`.

Instead of the statement `pass`, we define a constructor in the class `Employee`:

```
class Employee:
    def __init__(self, n, num, salary, performance):
        self.__name = n
        self.__empNum = num
        self.__salary = salary
        self.__performance = performance
```

`self.__name`, `self.__empNum`, `self.__salary` and `self.__performance` are instance variables that references the following objects:

- `self.__name` references an object from the class `str`
- `self.__empNum` references an object from the class `int`
- `self.__salary` references an object from the class `float`
- `self.__performance` references an object from the class `int`

Instance variables must be prefixed with `self.`, using the dot notation. Without the prefix, `__name`, `__empNum`, `__salary` and `__performance` are local variables or formal parameters and local variables.

`self` is a reference to the object that receives the service call and executes the method, in this case, the method **`__init__`**.

Methods that start with double underscore (or dunder), __ are special methods or magic methods that programmers should not call directly. The constructor `__init__(self <,parameter-list>)` is a special or magic method and it should not be called directly.

To create an `Employee` object, we use the class name instead and supply the actual parameters for the constructor:

```
e = Employee("Ann", 123, 1800, 4)
print(e)
```

```
Console
<terminated> employee.py [C:\Program Files\Python36\python.exe]
<__main__.Employee object at 0x000000000296AA20>
```

The following events happen when the statement

`e = Employee("Ann", 123, 1800, 4)` is executed.

- The statement `e = Employee("Ann", 123, 1800, 4)` calls to the class method, `__new__`, inherited from the class `object`.

  As mentioned in Section 1.2.2, a class can have class method. Class method is covered in section 2.2.2.

  The class `object` has a class method `__new__` with this method signature:

  ```
  __new__(cls <,parameter-list >)
  ```

  The first formal parameter of a class method is `cls` or the class that receives the method call.

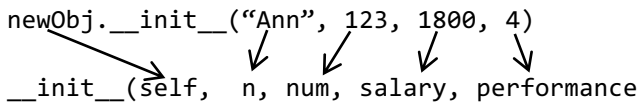
  - `Employee("Ann", 123, 1800, 4)` is mapped to a call to the method `__new__` defined in the class `object` ,
  - `Employee` is mapped to `cls` and
  - `"Ann", 123, 1800, 4` is mapped to `parameter-list`

- The class method `__new__` allocates memory to the `Employee` object being created.
- The class method `__new__` invokes the constructor `__init__` defined in the `Employee` class on the newly created `Employee` object.

`__init__` is an instance method. The first formal parameter of an instance method is `self` or the object that receives the method call. Suppose the newly created `Employee` object is referenced as `newObj`. Then

- `newObj.__init__("Ann", 123, 1800, 4)` is mapped to a call to the method `__init__` defined in the class `Employee` ,
- `newObj` is mapped to `self` and
- `"Ann", 123, 1800, 4` are mapped to the formal parameters `n, num, salary, performance` respectively.

The mapping is shown pictorially here:



- The method `__init__` initializes the instance variables with the values supplied to the formal parameters, and once the method `__init__` completes, control is return to the method `__new__`.
- The method returns the newly initialized object.
- `e` is assigned the reference of the newly initialized object.

Note that

- method call and return are handled in the *same four-step process* for function call and return in ICT133, Study Unit 4.
- `self` and `cls` are not keywords but they are conventionally used as formal parameters to refer to the object and to the class respectively, that receive the method call (a call for service) and execute the corresponding method.
- the class method `__new__` allocates space for the newly created object in the heap memory.

  Recall that in ICT133, Study Unit 6, objects reside in heap memory and unlike local variables and formal parameters, they continue to exist as long as their reference count is not zero.

Notice that we have used names that start with double underscore (or dunder) __ for the instance variables of the class `Employee`.

Instance variables declared with double underscore (or dunder) __ are private. Attempt to access private instance variables will cause a runtime error, e.g., accessing

the name of `Employee` object, `e` using `e.__name` produced a run-time error, as shown in the code fragment below:

```
e = Employee("Ann", 123, 1800, 4)
print(e)
print(e.__name)
```

```
Console
<terminated> employee.py [C:\Program Files\Python36\python.exe]
<__main__.Employee object at 0x00000000028BAA20>
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\sr
    print(e.__name)
AttributeError: 'Employee' object has no attribute '__name'
```

As the instance variable `__name` is private and so, is hidden to clients, the error message will report that the `Employee` object does not have the attribute `__name`.

Instead of using double underscore (or dunder) __ to prefix the names of instance variables and so make them private, the single underscore _ is conventionally used to prefix their names so as to communicate to clients that these instance variables are internal implementations and so should not be directly accessed.

The Python interpreter does not raise any error when such instance variables are accessed by clients. Instead, Python depends on programmers to honour the gentleman's agreement not to access internal implementations directly, but instead, to access such instance variables via accessor (or getter) and mutator (or setter) methods.

Henceforth, following Python convention, we will prefix instance variables with a single underscore _.

Note that each class can have at most one constructor. However, just as in functions, we may give default values to the formal parameters in the constructor, to achieve the effect of "**overloading**".

For example, we can give the default value `None` to `_salary` and the default value `None` to `_performance` if no value is specified to initialise them

```
    def __init__(self, n, num, salary = None, performance = None):
        self._name = n
        self._empNum = num
        self._salary = 2000 if salary is None else salary
        self._performance = 1 if performance is None else performance
```

Subsequently, the instance variables can be initialized to some predefined values if they reference `None`.

With "overloading", we can create `Employee` objects in a variety of ways such as:

```
e1 = Employee("Ann", 123, 1000, 4)
e2 = Employee("Jane", 125, performance = 3)
e3 = Employee("Sam", 126, 2500)
```

Activity 4

A Vehicle object has the following data members: the registration number, the make and model, the year of manufacture, engine capacity in c.c, daily rental rate before discount and whether the vehicle is on discount.

a) Write the constructor for the class Vehicle.
Create a Vehicle and print it.

b) Rewrite the constructor for the class Vehicle such that the default value for the year of manufacture is this year, daily rental rate is 100 and the vehicle is not on discount.

Create two Vehicle objects, one using the default value for the year of manufacture and the other without using default value.

2.1.2 Accessor and Mutator Methods

We will use the class `Employee` to illustrate the Pythonic way of providing controlled access to instance variable via accessor and mutator (setter) methods using the `@property` decorator.

Accessor methods allow inspection of and mutator methods allow update to the instance variables.

To allow controlled access to inspect and to update the instance variable, `_name`:

- Make an accessor method to inspect

  We define a method, `name` and apply the `@property` decorator on it:

```
@property
def name(self):
    return self._name
```

With this accessor method, we can inspect the instance variable, `self._name` via the method `name`. For example, if `e` references an `Employee` object, then

```
print(e.name)
```

will access the instance variable, `self._name` of the employee, `e` and print it.

- Make a mutator method to update

Once the accessor method `name` is defined, we can make a mutator method for it with a `@accessor.setter` decorator where `accessor` is the name of the accessor method.

```
@name.setter
def name(self, value):
    self._name = value
```

With this mutator method, we can update the instance variable, `self._name` via the method `name`. For example, if `e` reference an `Employee` object, then

```
e.name = 'Anne'
```

will update the instance variable, `self._name` of the employee, `e` to `Anne`.

The code fragment shows the accessor method `name` called in the `print` statement to inspect the private instance variable `self._name`, and the setter method `name` called to update the instance variable `self._name` from `Ann` to `Anne`.

```
e = Employee("Ann", 123, 1800, 4)
print(e.name)
e.name = 'Anne'
print(e.name)
```

```
Console
<terminated> employee.py [C:\Program Files\Python36\p
Ann
Anne
```

In most cases of mutator methods, we want to perform a value check before updating the value of a private instance variable, e.g., `self._salary`. For example, the valid value for `self._salary` is between 1200 and 15000, inclusive.

```
@property
def salary(self):
    return self._salary
```

```
    @salary.setter
    def salary(self, value):
        if  value < 1200:
            self._salary = 1200
        elif value > 15000:
            self._salary = 15000
        else:
            self._salary = value
```

Alternatively, the salary setter method can be rewritten to use conditional expressions.

```
    @salary.setter
    def salary(self, value):
        self._salary = 1200  if  value < 1200 else 15000 if value > 15000 else
value
```

To ensure that `self._salary` is within the correct range when the newly created object is initialized we should apply the range check in the method `__init__`.

To do so, we simply call the setter method, `salary`, instead of directly assigning the given value in `sal` to the private instance variable, `_salary`.

```
    def __init__(self, n, num, salary, performance):
        self._name = n
        self._empNum = num
        self.salary = salary
        self._performance = performance
    def __init__(self, n, num, salary = None, performance = None):
        self._name = n
        self._empNum = num
        # replace self._salary with self.salary
        self.salary = 2000 if salary is None else salary
        self._performance = 1 if performance is None else performance
```

The code fragment shows that `self._salary` always store a value within the valid range even the Employee object is created or updated with invalid value for salary.

```
e = Employee("Ann", 123, 1000, 4)
print(e.name, e.empNum, e.salary)
e.salary = 100000
print(e.name, e.empNum, e.salary)
```

```
Console
<terminated> employee.py [C:\Program Files\Python36\python.exe]
Ann 123 1200
Ann 123 15000
```

There are 4 combinations as to whether the getter and setter methods are implemented, and which combination to implement is dependent on whether the class must provide these services to clients.

- When both accessor and mutator services are required
    - define a instance method to return the private instance variable
    - apply the `@property` decorator for the method

      For example,

```
@property
def name(self):
    return self._name
```

    - define an instance method to update the private instance variable
    - apply the *`@accessor.setter`* decorator for the method where *`accessor`* is the name of the accessor method.

      For example,

```
@name.setter
def name(self, value):
    self._name = value
```

- When both inspection and update services are not required

  No implementation of accessor or mutator method for that private instance variable.

- When only the accessor service is required
    - define an instance method for the private instance variable
    - apply the `@property` decorator for the method

      For example, the private instance variable `__empNum` can be inspected but not updated, implement only the getter method.

```
@property
def empNum(self):
    return self._empNum
```

- When only the setter service is required
    - define an instance method to update the private instance variable
    - apply the `property` function for the method with the first argument `None` for the accessor method

For example, the private instance variable `_performance` can be updated but not inspected by clients, so implement only the mutator method.

```
def performance(self, value):
    self._performance = value

performance = property(None, performance)
```

Activity 5

Continue to work on class Vehicle and write accessor and mutator methods using the @property decorator or the property function, where applicable, to allow

- only inspection of the registration number, make, model, year of manufacturer, capacity and whether vehicle is on discount
- both inspection and mutation of the daily rental rate.

  The accessor method returns 10% of the rate if the vehicle is on discount and 100% of the rate otherwise.

  The mutator method validates that the rental rate before discount is between 70 and 300, inclusive. If the provided value is outside this range, set the rental rate to 70 if the provided value is less than 70, and set it to 300 if the provided value is more than 300

Make the change to the constructor to enforce the check for the rental rate when a Vehicle object is created.

2.1.3 Magic Methods

Any method that starts with double underscore (or dunder), __ is a special method or a magic method that client code should not call directly. We have seen two of such methods, `__new__` and `__init__`.

Some methods of interest are for the following operations:

- Comparison operations such as

  `__eq__(self, other)`, `__ne__(self, other)`, `__lt__(self, other)`, `__gt__(self, other)`, `__le__(self, other)` and `__ge__(self, other)`, etc.

  Implement these methods if the class provide comparison services between 2 of its objects using the operators `==`, `!=`, `<`, `>`, `<=` and `>=` respectively.

  Having these methods allow these operators to be applied meaningful on 2 objects in your class.

  For example, the `__lt__(self, other)` can be implemented in the class `Employee` to return `True` either if the `Employee` object, *`self`* has less salary than

the `Employee` object, `other`, or if the salary is the same as the `Employee` object, `other` and the `Employee` object, *`self`* has a lower performance than `other`. Otherwise, the method returns `False`.

```
def __lt__(self, other):
    result = False`
    if  self._salary < other._salary:
        result = True
    elif self._salary == other._salary and \
    self._performance < other._performance:
        result = True
```

The method allows us to compare two `Employee` objects with the `<` operator . In this code fragmenet, the method **`__lt__`** is invoked on `e1` and so, `e1` is `self` and `e2` as `other`:

```
e1 = Employee("Ann", 123, 1000, 4)            # _salary is 1200
e2 = Employee("Jane", 125, performance = 3)   # _salary is 2000
print (e1 < e2)                               # output True
```

Activity 6

Continue to work on class Vehicle and write the following methods which compare the engine capacity:

a) `__eq__(self, other)`
If both Vehicle objects have the same engine capacities, the method returns `True`, otherwise it returns `False`.

b) `__lt__(self, other)`
If `self` has a smaller engine capacity, the method returns `True`, otherwise it returns `False`.

c) `__gt__(self, other)`
If `self` has a larger engine capacity, the method returns `True`, otherwise it returns `False`.

As an exercise, implement this method in terms of calling the methods you implemented in parts a) and b).

d) Test your methods.

- Unary operations such as

`__abs__(self)` and `__neg__(self)`, etc.

As unary operators are not meaningful to `Employee` object, we will not implement them in the class `Employee`. Had these methods been implemented, they would allow us to apply the unary operators `abs()` and `-` on an `Employee` object to invoke the `__abs__(self)` and `__neg__(self)` respectively, as shown here:

```
print (abs(e1), -e1)
```

- Arithmetic operations such as

`__add__(self, other)`, `__sub__(self, other)`,`__mul__(self, other)`, `__div__(self, other)`, `__floordiv__(self, other)`, `__mod__(self, other)` and `__pow__(self, other)`, etc.

These methods implement the operators `+`, `*`, `/`, `//` , `%` and `**` respectively. Implement only meaningful operators to allow operators to be applied meaningfully on 2 objects in your class.

For example, the `__sub__(self, other)`is implemented in the class `Employee` to return the result of subtracting the salary of the `Employee` object, *`self`*, by the salary of the `Employee` object, `other`.

```
def __sub__(self, other):
    return self._salary - other._salary
```

Having the method allows us to subtract the salary of one `Employee` object, `e1` as `self` with the salary of the other `Employee` object, `e2` as `other`, as shown here:

```
e1 = Employee("Ann", 123, 1000, 4)            # _salary is 1200
e2 = Employee("Jane", 125, performance = 3)   # _salary is 2000
print (e1 - e2)
```

- Augmented assignment operations such as

`__iadd__(self, other)`, `__isub__(self, other)`, `__imul__(self, other)`, `__idiv__(self, other)`, `__ifloordiv__(self, other)`, `__mod__(self, other)` and `__ipow__(self, other)`, etc.

These methods implement the operators `+=`, `*=`, `/=`, `//=` , `%=` and `**=` respectively.

As augmented assignment operations are not meaningful to `Employee` object, we will not implement them in the class `Employee`. Had these methods been implemented, they would allow us to apply these operators on two `Employee` objects as shown here:

```
e1 += e2
```

- Type conversion operations such as

  `__int__(self)` and `__float__(self)`, etc

  These methods implement the conversion operation `int()` and `float()` respectively. Implement only meaningful conversion operations to allow operators to be applied meaningfully on an object in your class.

  As type conversion operations are not meaningful to `Employee` object, we will not implement them in the class `Employee`. Had these methods been implemented, they would allow us to apply these operators on an `Employee` object as shown here:

```
float(e1)
```

- Operations that return a representation of an object in the class such as

  `__str__(self)` and `__repr__(self)`, etc.

  These methods implement the operators `str()`and `repr()` respectively. `str()` returns a string representation of an object for the human reader whereas `repr()`returns a string representation of an object, possibly executable Python code.

  For example, the `__str__(self)`is implemented in the class `Employee` such that it return a string with attribute names and values.

```
def __str__(self):
    return ("Emp No: {:3d}  name: {:<10}  performance Level: " \
        + "{:1d}  salary: ${:8.2f}") \
       .format(self._empNum, self._name, self._performance, self._salary)
```

  The `__str__()` method allows us to get a string representation of an `Employee` objects with the `str()` operator as shown here:

```
e1 = Employee("Ann", 123, 1000, 4)
print(e1)
print(str(e1))
```

  Output:

```
Emp No: 123  name: Ann         performance Level: 4  salary: $ 1000.00
Emp No: 123  name: Ann         performance Level: 4  salary: $ 1000.00
```

Activity 7

Continue to work on class Vehicle and write the `__str__(self)` method. The rental rate should be the discounted rate if there is a discount,

2.1.4 Other Methods

In addition, methods that allow an object to perform application-related services that must be coded. For example, suppose an `Employee` object must respond to a service call to calculate his bonus or to update his salary.

For simplicity, assume the bonus of an employee is determined by his performance and current salary, according to the table:

Performance Level	Bonus
5	2 times monthly salary
4	1.2 times monthly salary
3	0.5 times monthly salary
0-2	No bonus

The method implemented to calculate bonus is as follows:

```
def calculateBonus(self):
    '''
    This method returns the bonus which is a factor of the employee's
    monthly salary. The factor is determined by the performance level
    of the employee. '''

    bonus = 0
    if self._performance == 5:
        bonus =  2 * self._salary
    elif self._performance == 4:
        bonus = 1.2 * self._salary
    elif self._performance == 3:
        bonus = 0.5 * self._salary
    return bonus
```

Assume also that the salary is increased by a percentage of the monthly salary plus a base increase.

The method implemented is as follows:

```
def increaseSalaryBy(self, percent = 0, base = 20):
    '''
    This method updates the monthly salary by a percentage of the meploye's
    monthly salary and a base increase '''

    self.salary = self._salary * (1 + percent/100) + base
```

Notice that again, we use the mutator method, **`salary`** to update the salary so that the range check will put salary in the valid range.

Activity 8

Continue to work on class Vehicle and write the `changeOnDicount(self)` method. Set the vehicle to be on discount if it is not, and set it to be not on discount if it is on discount.

Activity 9

Continue to work on class Vehicle.

- Write the method to compute the age of the Vehicle.
- Write the method to compute and return the discount rate, a number between 0 and 1, inclusive, to give if the vehicle is on discount. Give a 5% discount if the age of the vehicle is 2 years and under, 10% if the age of the vehicle is more than 2 years but 5 years or under and 15% otherwise.
- Change the accessor method for rental rate to use the computed discount rate instead of 10%, if the vehicle is on discount.

2.1.5 Object Composition with User-Defined Type

Besides using built-in classes for instance variables, we can also use user-defined classes for instance variables when we apply object composition.

To illustrate object composition using user-defined classes, we include an instance variable that references the supervisor of an Employee object. A supervisor is also an Employee object. Some Employee objects may not have a supervisor.

We modify the constructor as follows:

```
def __init__(self, n, num, salary = None, performance = None, supervisor= None):
      self._name = n
      self._empNum = num
      self.salary = 2000 if salary is None else salary
      self._performance = 1 if performance is None else performance
      self._supervisor = supervisor
```

The accessor and mutator methods for this instance variable are as follows:

```
@property
def supervisor(self):
    return self._supervisor

@supervisor.setter
def supervisor(self, supervisor):
    self._supervisor = supervisor
```

Note that there is no difference in way an instance variable which references an object from a user-defined class is initialized, inspected and updated.

We can also invoke methods on such instance variable. For example, to modify the method **__str__** to include also

- the employee's bonus, we use the expression `self.calculateBonus()`, and
- the supervisor's employee name, we use the expression `self._supervisor.name` where the accessor method `name` is invoked on the instance variable `self._supervisor`.

```
    def __str__(self):
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
              + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} Supervisor: {:}")\
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor.name)
```

We can test the code by creating Employee objects with and without supervisor:

```
e3 = Employee("Sam", 126, 2500)
e2 = Employee("Jane", 125, performance = 3, supervisor = e3)
e1 = Employee("Ann", 123, 1000, 4, e2)
print(e1)
print(e2)
print(e3)
```

Output:

```
Emp No: 123  Name: Ann         Performance Level: 4  Salary: $ 1200.00 Bonus: $ 1440.00 Supervisor: Jane
Emp No: 125  Name: Jane        Performance Level: 3  Salary: $ 2000.00 Bonus: $ 1000.00 Supervisor: Sam
Emp No: 126  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: None
True
-800
```

Here is the complete class for `Employee`:

```
class Employee:
    ''' This class models an employee.
    Each employee has
         a name (str),
         an employee number (int),
         salary (float),
         a performance level and
         optionally, a supervisor
    '''

    def __init__(self, n, num, salary = None, performance = None, supervisor=
None):
        '''
        Initialises an employee object
        Requires name, employee number
        Optionally, salary, performance and supervisor which are given
        default values

        '''
        self._name = n
        self._empNum = num
        self.salary = 2000 if salary is None else salary
        self._performance = 1 if performance is None else performance
        self._supervisor = supervisor

    @property
    def name(self):
        '''getter method for employee name'''
        return self._name

    @name.setter
    def name(self, value):
        '''setter method for employee name'''
        self._name = value

    @property
    def empNum(self):
        '''getter method for employee number'''
        return self._empNum

    @property
    def salary(self):
        '''getter method for employee salary'''
        return self._salary

    @salary.setter
    def salary(self, value):
        '''setter method for employee salary'''
```

```
        self._salary = 1200 if  value < 1200 \
        else Employee.15000 if value > 15000 else value

    def performance(self, value):
        '''setter method for employee salary'''
        self._performance = value

    performance = property(None, performance)

    def __lt__(self, other):
        '''
        This method returns True if the employee has a smaller salary or
        if the employee has the same salary but a lower performance than
        the other employee
        '''
        result = False
        if  self._salary < other._salary:
            result = True
        elif self._salary == other._salary \
        and self._performance < other._performance:
            result = True
        return result

    def __sub__(self, other):
        '''
        This method returns the difference between the salaries of the
        employee and the other employee
        '''
        return self._salary - other._salary

    def calculateBonus(self):
        '''
        This method returns the bonus which is a factor of the meploye's
        monthly salary. The factor is determined by the performance level
        of the employee.
        '''
        bonus = 0
        if self._performance == 5:
            bonus =  2 * self._salary
        elif self._performance == 4:
            bonus = 1.2 * self._salary
        elif self._performance == 3:
            bonus = 0.5 * self._salary
        return bonus


    def increaseSalaryBy(self, percent = 0, base = 20):
        '''
        This method updates the monthly salary by a percentage of the meploye's
        monthly salary and a base increase
        '''
        self.salary = self._salary * (1 + percent) + base

    @property
    def supervisor(self):
        '''getter method for employee's supervisor'''
        return self._supervisor

    @supervisor.setter
```

```
    def supervisor(self, supervisor):
        '''setter method for employee's supervisor'''
        self._supervisor = supervisor

    def __str__(self):
        '''
        This method returns a string representation of an employee
        '''
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
                + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} Supervisor: {:}") \
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor.name)
```

2.2 Defining classes with class level members

The implementation of the class `Employee` can be improved using class variables and methods, the focus of this section. Class variables and class methods are different from instance variables and instance methods. Bear in mind that

- Instance variables that record the data of individual objects

  Each object has its own copy of the instance variables.

- Instance methods that the individual objects can perform.

2.2.1 Class-Level Attributes

Unlike instance variables which belong to the individual objects, class variables belong to the class. There is only one copy of each class variable, regardless of the number of objects in the class.

Class variables are defined outside methods, within the class definition, unlike instance variables which are defined (via initialisation) in the constructor, `__init__`.

For example,

```
class Employee:
    _perfFactor = {5:2, 4:1.2, 3:0.5}
    _nextNumber = 1
```

`_perfFactor` and `_nextNumber` are class attributes. There is one copy of these attributes for the whole class.

Class attributes are useful in these cases:

- maintaining a common piece of data that applies to all objects in that class
- generating a running number for that class

Instance methods can access class variables. For example, the instance method **`calculateBonus`** can access the class variable `_perfFactor`.

```
    def calculateBonus(self):
        '''
        This method returns the bonus which is a factor of the meploye's
        monthly salary. The factor is determined by the performance level
        of the employee. '''
        return  Employee._perfFactor.get(self._performance, 0) * self._salary
```

The constructor **`__init__`** can access and update the class variable `nextNumber`.

```
    def __init__(self, n, salary = None, performance = None, supervisor= None):
        '''
        Initialises an employee object
        Requires name, employee number
        Optionally, salary, performance and supervisor which are given
        default values

        '''
        self._name = n
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1
        self.salary = 2000 if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
```

Instead of supplying an employee number when creating an Employee object, we can get the class `Employee` to generate the employee number.

In this example,

- the employee number takes the current value of the class variable, `_nextNumber`.
- `_nextNumber` is then incremented to ready it for the next employee

The code fragment tests the updated `__init__` method:

```
e3 = Employee("Sam", 2500)
e2 = Employee("Jane", performance = 3, supervisor = e3)
print(e3)
print(e2)
```

Output:

```
Emp No:   1  name: Sam         performance Level: 1  salary: $ 2500.00 Bonus:
$    0.00 Supervisor: None
Emp No:    2   name: Jane          performance Level: 3   salary: $ 2000.00 Bonus:
$ 1000.00 Supervisor: Sam
```

To eliminate hardcoding values in our program code, we introduce 4 more class variables:

```
_defSal = 2000
_defPerf = 1
_maxSalary = 15000
_minSalary = 1200
```

The constructor is modified to remove hardcoded values related to default salary and performance level:

```
def __init__(self, n, salary = None, performance = None, supervisor= None):
    '''
    Initialises an employee object
    Requires name, employee number
    Optionally, salary, performance and supervisor which are given
    default values

    '''
    self._name = n
    self._empNum = Employee._nextNumber
    Employee._nextNumber += 1
    self.salary = Employee._defSal if salary is None else salary
    self._performance = Employee._defPerf if performance is None \
    else performance
    self._supervisor = supervisor
```

Likewise, we will modify the mutator method for the instance variable `_salary`.

```
@salary.setter
def salary(self, value):
    '''setter method for employee salary'''
    self._salary = \
    Employee._minSalary if  value < Employee._minSalary \
    else Employee._maxSalary if value > Employee._maxSalary \
    else value
```

Research

Updating the __init__ method to access the class variable ncxtNumber using the self reference to:

```
    def __init__(self, n, salary = None, performance = None,
    supervisor= None):
        self._name = n
        self._empNum = self._nextNumber
        Employee._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
```

and using the type(self) reference to:

```
    def __init__(self, n, salary = None, performance = None,
    supervisor= None):
        self._name = n
        self._empNum = type(self)._nextNumber
        type(self)._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
```

have no impact on the last output.

`However, updating the __init__ method to update the class variable ncxtNumber using the self reference

```
    def __init__(self, n, salary = None, performance = None,
    supervisor= None):
        self._name = n
        self._empNum = Employee._nextNumber
        self._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
```

produces an incorrect employee number for Jane:

```
Emp No:   1  name: Sam         performance Level: 1  salary: $ 2500.00 Bonus:
$    0.00 Supervisor: None
Emp No:   1  name: Jane        performance Level: 3  salary: $ 2000.00 Bonus:
$ 1000.00 Supervisor: Sam
```

Find out the difference between

- self and type(self) and
- type(self) and the class name

2.2.2 Class-Level Methods

Class methods are defined with the decorator *@classmethod*. While the first argument of an object level class method is `self`, the first argument of a class level class method is `cls`.

Class methods are methods that implement the services of a class, e.g., services to add or remove the factors for calculating bonus as follows:

```
@classmethod
def addPerfFactor(cls, performanceLevel, factor):
    cls._perfFactor[performanceLevel] = factor

@classmethod
def removePerfFactor(cls, performanceLevel):
    return cls._perfFactor.pop(performanceLevel, 0)
```

We may also add other class methods to inspect and update the class variables for default values for salary and performance and the minimum and maximum salaries.

2.2.3 Static Methods

A class may contain 3 types of methods:

- instance methods

  These methods implement the services provided by objects.

- class methods

  These methods implement the services provided by class.

- static methods

  These methods are utilities and ideally, they should be implemented as module level functions (ICT133, Study Unit 4). When defined within a class, static methods require the decorator *@staticmethod*.

  For example:

  ```
  @staticmethod
  def belongs(obj, aClass):
      return type(obj) == aClass
  ```

Static methods, like functions, are called with its module name, in this case, Employee, e.g.,

```
print(Employee.belongs(e3, type(e2)))
```

Activity 10

Modify the Employee class to implement the following requirements.

- Add three class variables for storing the maximum, minimum and default salaries. Initialise them with the values 15000, 1200 and 2000 respectively.
- Modify these methods to use the newly defined class vaariables:
  - `__init__`
  - the setter method `salary`

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 206-230.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

2.3 Good Software Practice

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 237-268.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Good practices should adopt when developing classes as they make programs easier to maintain. Some good software practices are as follows:

- Naming convention
  - Class name should begin with uppercase letter. The class name is in the singular form.

- Function and method names should begin with lowercase letter. The function and method names should reflect the service/task performed by the function or method.

- Variables (instance, class and local) and formal parameters should begin with lowercase letter.

  The names should be meaningful and reflect the data that the variable is for.

  The name is singular if the variable stores a single value and the name should be in plural form or indicate a collection if the variable stores a collection of value.

- Instance and class variables are private and so should begin with an underscore.

- Formal Documentation using docstring

  - Document each class and its purpose. Each class should model only one type of object.

  - Document the instance and class variables, both their purpose and their data types

  - Document each method and its purpose/responsibility/task. Each method should model one purpose/responsibility/task. Document the formal parameters, the expected data types, whether their values are required or optional, and the output.

  - Document each function similarly.

- Apply encapsulation to reduce coupling of software components. Instance variables should be attributes of the object and methods should be responsibilities of objects. Hide implementation details.

- Think reuse rather than duplicate code. Don't repeat yourself (DRY) as far as possible.

- Use functions and classes to structure your program.

  Use function to manage application logic and classes to organise data and related functions (methods) .

- Test your code as your write them, to allow erroneous code to be quickly located.

These software practices is illustrated in the class Employee.

```
class Employee: # upper case, singular
    # Formal Documentation using docstring
    '''
    This class models an employee.
  The class has 6 class variables:
        the default, the maximum and minimum salaries of employees
        the default performance level
        a collection of performance level and the respective bonus factor
        the next employee number

   Each employee has
        a name (str),
        an employee number (int),
        salary (float),
        a performance level (int) and
        optionally, a supervisor (Employee)
    '''

    _defSal = 2000  # lower case, private and begin with an underscore
    _maxSalary = 15000
    _minSalary = 1200
    _defPerf = 2
    _perfFactor = {5:2, 4:1.2, 3:0.5}
    _nextNumber = 1

    @classmethod
    def addPerfFactor(cls, performanceLevel, factor):
            # Formal Documentation using docstring
        '''class method to add/update a performance level and its corresponding
        bonus factor.
        '''
        cls._perfFactor[performanceLevel] = factor

    @classmethod
        def removePerfFactor(cls, performanceLevel):
        '''class method to remove a performance level and its corresponding
        Bonus factor.
        '''
        return cls._perfFactor.pop(performanceLevel, 0)

   def __init__(self, n, salary = None, performance = None, supervisor= None):
        '''
        Initialises an employee object
        Requires name (str)
        Optionally salary (float), performance (int) and supervisor (Employee)
        which are given default values

        '''
        self._name = n
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
```

```
    @property
    def name(self):
        '''getter method for employee name'''
        return self._name

    @name.setter
    def name(self, value):
        '''setter method for employee name'''
        self._name = value

    @property
    def empNum(self):
        '''getter method for employee number'''
        return self._empNum

    @property
    def salary(self):
        '''getter method for employee salary'''
        return self._salary

    @salary.setter
    def salary(self, value):
        '''setter method for employee salary'''
        self._salary = \
        Employee._minSalary if  value < Employee._minSalary \
        else Employee._maxSalary if value > Employee._maxSalary \
        else value

    def performance(self, value):
        '''setter method for employee salary'''
        self._performance = value

    performance = property(None, performance)

    def calculateBonus(self):
        '''
        This method returns the bonus which is a factor of the meploye's
        monthly salary. The factor is determined by the performance level
        of the employee.
        '''
        return  Employee._perfFactor.get(self._performance , 0)  *
self._salary

    def increaseSalaryBy(self, percent = 0, base = 20):
        '''
        This method updates the monthly salary by a percentage of the
        employee's monthly salary and a base increase
        '''
        self.salary = self._salary * (1 + percent/100) + base

    @property
    def supervisor(self):
        '''getter method for employee's supervisor'''
        return self._supervisor

    @supervisor.setter
    def supervisor(self, supervisor):
        '''setter method for employee's supervisor'''
```

```
        self._supervisor = supervisor

    def __str__(self):
        '''
        This method returns a string representation of an employee
        '''
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
                + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} Supervisor: {:}") \
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor._name)
```

Summary

Object-oriented programming is a paradigm shift. Instead of abstracting just data through custom data types and abstracting just operations through function definition, object-oriented programming provides abstraction of data and operations simultaneously via a class. A class abstracts both data and operations to make a custom data type with custom operations.

A class specifies a *type of object* in terms of the object's attributes (or instance variables) and the object's behaviour (or methods).

The three main pillars of object-oriented programming are encapsulation, inheritance and polymorphism.

A class hides the implementation of both its variables and methods by making them private, but exposes the interface (or method signatures collectively). Encapsulation is the result of *abstracting data and operations simultaneously and performing information hiding*.

Inheritance is an *is-a* relationship between two classes. Inheriting attributes and methods means that the subclass has the attributes and methods of its superclass without having to define them. The subclass gets them simply by being a subclass of the superclass.

Object composition is a *has-a* relationship between two classes. When objects have a has-a relationship, the relationship is implemented as object composition. The contained object becomes an attribute for the container object.

Polymorphism means many forms or many interpretations. The interpretation of a polymorphic expression depends on the *class* of the *object* that a variable in the expression references.

`class` is the keyword to start a class definition. A constructor is a special method to initialize the instance variables of a newly created object. The Pythonic way of providing accessor and mutator methods is using the `@property` decorator. Accessor methods allow inspection whereas mutator methods allow update of the instance variables.

A class may have class variables and class methods. There is only one copy of each class variable, regardless of the number of objects in the class. Class methods are defined with the decorator *`@classmethod`*. While the first formal parameter of an instance method is `self`, the first formal parameter of a class method is `cls`. Static methods are similar to functions in modules.

Good practices are essential when developing classes as they make programs easier to maintain.

References

Goldwasser, M. H., & Letscher, D. (2014). *Object-Oriented Programming in Python [pdf version]. Retrieved from* http://cs.slu.edu/~goldwamh/oopp/

Formative Assessment

1. The object-oriented programming concept described by this phrase: "abstraction of data and operations" is _____.

A) object
@ Incorrect. A class is an abstraction of both data and operations. Refer to Section 1.1.3 Object, Study Unit 1.

*B) class
@ Correct. A class is an abstraction of both data and operations. Refer to Section 1.1.2 Class as Data and Operation Abstraction, Study Unit 1.

C) object composition
@ Incorrect. It is an has-a relationship between classes. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

D) inheritance
@ Incorrect. It is an is-a relationship between classes. Refer to Section 1.2.Has-A and Is-A relationships, Study Unit 1.

2. The object-oriented programming concept described by this phrase: "an instance of a class" is _____.

*A) object
@ Correct. A class is an abstraction of both data and operations. Refer to Section 1.1.3 Object, Study Unit 1.

B) class
@ Incorrect. A class is an abstraction of both data and operations. Refer to Section 1.1.2 Class as Data and Operation Abstraction, Study Unit 1.

C) object composition
@ Incorrect. It is an has-a relationship between classes. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

D) inheritance
@ Incorrect. It is an is-a relationship between classes. Refer to Section 1.2.Has-A and Is-A relationships, Study Unit 1.

3. Which pair of classes shows an incorrect inheritance relationship?

A) Manager is a staff
@ Incorrect. This is-a relationship is correct. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

B) Square is a shape

@ Incorrect. This is-a relationship is correct. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

*C) Peter is a manager
@ Correct. This is not an is-a relationship. Peter is an occurrence or a Manager object. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

D) Dog is an animal.
@ Incorrect. This is-a relationship is correct. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

4. Which pair of classes shows an incorrect object composition relationship?

A) Student has a name
@ Incorrect. This is a has-a relationship or an object composition relationship. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

B) Student is enrolled in a degree program
@ Incorrect. Enrolled is modelled as a has-a relationship or an object composition relationship. A student has a degree program in which he is enrolled. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

C) Student must belong to a tutorial group.
@ Incorrect. This is-a relationship is incorrect. Peter is an occurrence or a Manager object. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

*D) Student can be a student club president.
@ Correct. The can be relationship is an is-a relationship in which a student club president is a student. An is-a relationship is not an object composition relationship. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 1.

5. What makes a programming statement polymorphic?

*A) When an object references can refer to objects from different classes and these classes have the same interface.
@ Correct. The different implementations of the same interface produce different interpretation to a programming statement, thus, making it polymorphic. Refer to Section 1.3 Polymorphism, Study Unit 1.

B) When the statement is written using an object-oriented programming language.
@ Incorrect. This is not a correct criterion. Refer to Section 1.3 Polymorphism, Study Unit 1.

C) When the program uses object composition.
@ Incorrect. This is not a correct criterion. Refer to Section 1.3 Polymorphism, Study Unit 1.

D) When the method is not implemented in the subclass but is implemented in the superclass.
@ Incorrect. This is not a correct criterion. Refer to Section 1.3 Polymorphism, Study Unit 1

6. Give the name of type of method that inspects an instance variable and specify the decorator used for that method, if applicable.

*A) accessor and @property
@ Correct. Accessor methods allow inspection and the decorator used is @property. Refer to Section 2.1.2 Accessor and Mutator Methods, Study Unit 1.

B) mutator and @property
@ Incorrect. Mutator methods allow update.. Refer to Section 2.1.2 Accessor and Mutator Methods, Study Unit 1.

C) class method and @classmethod
@ Incorrect. A class method is a method that belongs to a class and has cls as the first argument. cls refers to the class that the method is invoked on. Refer to Section 2.2.2 Class-Level Methods, Study Unit 1.

D) magic methods without decorator
@ Incorrect. A magic method starts with double underscore (or dunder), ___ and is a special method that code outside the class definition should not call directly. Refer to Section 2.1.3 Magic Methods, Study Unit 1.

7. Suppose _cv is defined as a class variable in the class Test and assigned the value 0. Which is a correct statement to include in an instance method to increment _cv by 1 whenever that instance method is called so that _cv can count the number of times that method is called?

A) `_cv += 1`
@ Incorrect. _cv is a local variable. Refer to Section 2.2.1 Class-Level Attributes, Study Unit 1.

B) `self_cv += 1`
@ Incorrect. This statement introduces a new instance variable, _cv and does not update the class variable. Refer to Section 2.2.1 Class-Level Attributes, Study Unit 1.

C) `cls_cv += 1`
@ Incorrect. The first formal parameter for an instance method is conventionally `self`. Even the first formal parameter is changed to `cls`, it will reference to the object for which the method is invoked on. Refer to Section 2.2.2 Class-Level Methods, Study Unit 1.

*D) `type(self)_cv += 1`
@ Correct. `type(self)` refers to the class. Refer to Section 2.2.2 Class-Level Methods, Study Unit 1.

8. Which guideline is not a good software practice?

A) A class name should begin with uppercase letter. The name is in the singular form.
@ Incorrect. This guideline to follow naming convention is a good software practice Refer to Section 2.3 Good Software Practice, Study Unit 1.

B) Instance and class variables are private and so should begin with an underscore.
@ Incorrect. This guideline to follow naming convention is a good software practice Refer to Section 2.3 Good Software Practice, Study Unit 1.

*C) Choose inheritance over object composition.
@ Correct. This guideline is not a good software practice Refer to Section 2.3 Good Software Practice, Study Unit 1.

D) Avoid code duplication.
@ Incorrect. This guideline is a good software practice Refer to Section 2.3 Good Software Practice, Study Unit 1.

View the answers at the end of this study unit.

Solutions or Suggested Answers

Activity 1

Attributes: Customer Id, name, address, contact, date of birth
Methods: compute age, reveal contact number, reveal customer id

Activity 2

Vehicle, car, van, truck
Rental
Staff
Maintenance record etc

Activity 3

A rental has a customer and a Vehicle
Rental company has customers, has vehicles, has rentals
IndividualCustomer is a Customer, CorporateCustomer is a Customer
Car is a Vehicle, Truck is a Vehicle, Van is a Vehicle etc

Activity 4

a)

```
import datetime
class Vehicle:
    def __init__(self, regNum, make, model, cap , yearMfg, rate, onDiscount):
        self._registrationNumber = regNum
        self._make = make
        self._model = model
        self._capacity = cap
        self._yearOfManufacture = yearMfg
        self._rate = rate
        self._onDiscount = onDiscount

if __name__ == '__main__':
    v1 = Vehicle("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
   print(v1)
```

b)

```
import datetime
class Vehicle:
    def __init__(self, regNum, make, model, cap , yearMfg=
datetime.datetime.now().year, rate = 100, onDiscount = False):
        self._registrationNumber = regNum
        self._make = make
```

```
        self._model = model
        self._capacity = cap
        self._yearOfManufacture = yearMfg
        self.rate = rate
        self._onDiscount = onDiscount

if __name__ == '__main__':
    v1 = Vehicle("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
    v2 = Vehicle("SY1235X", 'Citroen', 'C3 AirCross', 1199)
    print(v1)
    print(v2)
```

Activity 5

```
    @property
    def registrationNumber(self):
        return self._registrationNumber

    @property
    def make(self):
        return self._make

    @property
    def model(self):
        return self._model

    @property
    def yearOfManufacture(self):
        return self._yearOfManufacture

    @property
    def capacity(self):
        return self._capacity

    @property
    def rate(self):
        return self._rate *  0.9 if self._onDiscount else self._rate

    @rate.setter
    def rate(self, value):
        self._rate = 70 if value < 70 else 300 if value > 300 else value


  def __init__(self, regNum, make, model, cap , yearMfg=
datetime.datetime.now().year, rate = 100, onDiscount = False):
        self._registrationNumber = regNum
        self._make = make
        self._model = model
        self._capacity = cap
        self._yearOfManufacture = yearMfg
        self.rate = rate
        self._onDiscount = onDiscount
```

Activity 6

```
    def __eq__(self, other):
        return self._capacity == other._capacity

    def __lt__(self, other):
        return self._capacity < other._capacity

    def __gt__(self, other):
            return not(self == other or  self < other)

if __name__ == '__main__':
    v1 = Vehicle("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
    v2 = Vehicle("SY1235X", 'Citroen', 'C3 AirCross', 1199)
    print(v1)
    print(v2)
    print (v1 < v2)
    print (v1 == v2)
    print (v1 > v2)
```

Activity 7

```
  def __str__(self):
    return ("Registration No: {:9s} Make: {:15s} Model:  {:10s} \n" \
        + "Engine Capacity: {:4d}   Year of manufacture: {:4d}   Rental Rate: " \
        + "$ {:5.2f} On Discount: {!r}\n"). \
        format(self._registrationNumber, self._make, self._model, self._capacity,
            self._yearOfManufacture, self.rate, self._onDiscount
```

Activity 8

```
    def changeOnDiscount(self):
        self._onDiscount = not(self._onDiscount)
```

Activity 9

```
    def age(self):
        return datetime.datetime.now().year - self._yearOfManufacture

    def computeDiscountRate(self):
        age =  self.age()
        return 0.05 if age <= 2 else 0.10 if age <= 5 else 0.15

    @property
    def rate(self):
        return self._rate * (1 - self.computeDiscountRate()) if self._onDiscount
else self._rate
```

Activity 10

```
class Employee:
    _perfFactor = {5:2, 4:1.2, 3:0.5}
    _nextNumber = 1
    _maxSalary = 15000
```

```
_minSalary = 1200
_defaultSalary = 2000

def __init__(self, n, sal = _defaultSalary, perf = 1, supervisor= None):
    self._name = n
    self._empNum = Employee._nextNumber
    Employee._nextNumber += 1
    self.salary = sal
    self._performance = perf
    self._supervisor = supervisor

@salary.setter
def salary(self, value):
    self._salary = Employee._minSalary  if  value < Employee._minSalary \
    else Employee._maxSalary if value > Employee._maxSalary else value
```

STUDY UNIT 2
INHERITANCE

Learning Outcomes

By the end of this unit, you should be able to:

1. Differentiate between subclass and superclass
2. Explain the keyword super
3. Implement a subclass with object level members
4. Define method overriding and differentiate the two types of method overriding
5. Describe multiple inheritance and implement it
6. Explain how common methods in super classes are resolved

Overview

Classes in Python 3 are new style classes. This means the classes that do not specify a superclass inherit from the class `object` implicitly.

In an inheritance relationship, subclasses inherit both instance and class variables as well as instance and class methods, in particular, the class method `__new__` in the class `object` which is invoked whenever an object is created. However, you will see that inheritance in Python is somewhat different as instance variables are not inherited.

This study unit focuses on defining classes that specify one or more superclasses explicitly. It covers how a subclass can augment its superclass through defining new instance or class variables and methods, and how a subclass can specialise its superclass by redefining the implementation of the methods inherited from the superclass. Methods can be specialised in two ways: overriding by refinement and overriding by replacement. The `super` keyword is introduced and is differentiated from the keyword `self`.

Python allows subclasses to inherit from multiple superclasses, and it applies the Method Resolution Order (MRO) to resolve which methods of superclasses are bounded to a method call. The study unit elaborates on multiple inheritance and illustrates the MRO for Python 3.

This study unit also covers abstract superclass and abstract methods. Abstract methods allow us to apply the template method design pattern, a design pattern that applies polymorphism.

As this unit covers chapters 9 of the textbook as well as other resources, it is estimated that the student will spend about 6 hours to read the textbook chapters and resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 Defining subclasses

1.1 Python 3 Classes

All Python 3 classes are new style classes and they implicitly inherit from the class `object`, the superclass for classes that are defined in Study Unit 1.

Recall that

- inheritance is an is-a relationship. The subclass is a specialised class of a general class, the superclass.
- Python allows for multiple inheritance.
- in an inheritance relationship, a subclass inherits the instance and class variables as well as the instance and class methods from the superclass.
- the class `object` is the only superclass that you have encountered so far although, a superclass can be a user-defined class.

Inheritance is particularly useful when the objects in a problem specification are variations of a general type. In such a case, the class for the general type is first defined. After which, the classes for the variations are defined as subclasses. Each subclass inherits from the superclass, and so, only the variations need to be implemented in the subclasses.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], page 299.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

Consider a Bank application. There are a variety of bank account types. Therefore, we first define a basic bank account class, say Savings account, assuming it is a basic bank account type in the application.

Once the Savings account class is defined (as a subclass of the class `object`), we will then define classes for the variations of bank account types in the problem specification, e.g., Junior account and a Senior account etc., as subclasses of the class Saving account. We depict inheritance relationship with a no-filled triangle arrow.

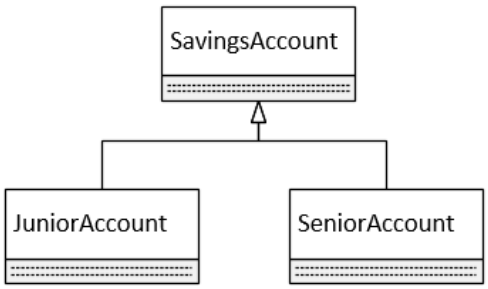


Figure 2.1 Inheritance relationship with SavingsAccount as superclass, and JuniorAccount and SeniorAccount as subclass of SavingsAccount.

Activity 1

Identify the superclass and subclasses for each problem statement:

a) A module is either a core or an elective.
b) Teaching staff has regular teaching evaluations. Management staff has a portfolio. Besides teaching and management staff, there are other types of staff.
c) A student is either a full-time or a part-time student. A student is also either a local student or a foreign student.

The class **`SavingsAccount`** is defined here:

```
class SavingsAccount: # implicitly a subclass of object
    ''' This class models a savings account.
    The class has 3 class variables:
         the default initial balance,
         the nextAccountNumber and
         the prevailing interest rate for savings account objects
    Each saving account object has
         an account number (int) qnd
         balance (float)
    '''

    _defBalance = 500
    _nextAccountNumber = 1
    _interestRate = 0.01
    _serviceCharge = 5

    @classmethod
    def minBalance(cls, bal):
        ''' update the default minimum balance
        '''
        cls._defBalance = bal
```

```
    @classmethod
    def getMinBalance(cls):
        ''' return the current default value of minimum balance
        '''
        return cls._defBalance

    @classmethod
    def interestRate(cls, rate):
        '''update the default interest rate
        '''
        cls._interestRate = rate

    @classmethod
    def getInterestRate(cls):
        ''' return the current default value of interest rate
        '''
        return cls._interestRate

    def __init__(self, balance = None):
        '''
        Initialise a saving account object
            account number is auto-generated and
            balance is optional and is given default value when not provided
        '''
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        if  balance is None:
            self._balance = type(self)._defBalance
        else:
            self._balance = balance

    @property
    def balance(self):
        '''return balance of saving account object
        '''
        return self._balance

    @property
    def accountNumber(self):
        '''return account number of saving account object
        '''
        return self._accountNumber

    def deposit(self, amount):
        '''
        Add amount to balance if amount > 0 and return True.
        Otherwise return False
        '''
        if amount <= 0:
            return False
        self._balance += amount
        return True

    def withdraw(self, amount):
        '''
        Subtract amount from account balance if account balance >= amount
        and return True. Otherwise return False
```

```
        '''
        if self._balance < amount:
            return False
        self._balance -= amount
        return True

    def addInterest(self):
        '''
        Add the interest based on the current interest rate and balance
        '''
        self._balance += type(self)._interestRate * self._balance


    def deductServiceCharge(self):
        '''
        Deduct service charge from the balance if the balance falls under
        the required balance which is the default balance.
        Balance can be negative after the deduction
        '''
        if self._balance < type(self)._defBalance:
            self._balance -= type(self)._serviceCharge

    def __str__(self):
        '''
        Return a string representation of a savings account
        '''
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumer, self._balance)
```

We next define a class **`JuniorAccount`** as a subclass of `SavingsAccount`.

```
class JuniorAccount(SavingsAccount):
    pass
```

With just the statement `pass`, **`JuniorAccount`** inherits

- all the instance variables: `_accountNumber` and `_balance` from `SavingsAccount` including those instance variables that `SavingsAccount` inherits from `object`.
- all the class variables: `_defBalance`, `_nextAccountNumber`, `_interestRate` and `_serviceCharge` from `SavingsAccount` including those class variables that `SavingsAccount` inherits from `object`.
- all the instance methods such as the constructor `__init__`, the accessor methods `balance`, `accountNumber`, the other methods such as `deposit`, `withdraw`, `addInterest`, `deductServiceCharge`, and the magic method `__str__` from `SavingsAccount` including those instance methods that `SavingsAccount` inherits from `object`.
- all the class methods such as `minBalance`, `getMinBalance`, `interestRate`, and `getInterestRate` from `SavingsAccount` including those class methods that `SavingsAccount` inherits from `object` such as `__new__`.

Thus, a `JuniorAccount` object can behave as a `SavingsAccount` object, having all the required variables and methods. A `JuniorAccount` object can substitute a `SavingsAccount` object.

Executing the code fragment:

```
ja = JuniorAccount()
print(ja)
print(ja.deposit(100))
print(ja)
```

produces this output:

```
Account No:   1 Balance: $  500.00
True
Account No:   1 Balance: $  600.00
```

Activity 2

a) Define the class `SeniorAccount` which is a subclass of `SavingsAccount`.
b) Create two instances of `SeniorAccount`, the first instance is created without specifying the initial balance and the second with an initial balance of 700.
c) Print both instances.
d) Deposit $100 to the first instance and withdraw $55 from the second instance.
Print both instances again.

Research

Inheritance in Python is somewhat different from the conventional inheritance relationship in OOP. Although we say that a subclass inherits the instance variables, this is not true for Python inheritance because instance variables begin to exist only when they are assigned values.

How and when does `JuniorAccount` object get the `_accountNumber` and `_balance` defined in the class `SavinsAccount` ?

A subclass object may

- augment the superclass

  A subclass can define new instance or class variables and new methods, thus a subclass can have additional variables and methods, or more functionality than a superclass.

- specialising the superclass

  A subclass can redefine the implementation of the methods inherited from the superclass. So, although a subclass has the same interface as the superclass, it can have different implementations of methods, allowing it to respond differently from the superclass for the same method calls.

  Thus, a subclass specialising the superclass is what that makes method calls polymorphic.

1.2 Augmentation

Augmentation adds variables and methods on top of those attributes and methods inherited from the superclasses.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 300-302.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Suppose a `JuniorAccount` object has an additional instance variable, `guardian`. We will augment the `JuniorAccount` class with the accessor methods for `guardian`.

```
@property
 def guardian(self):
     '''getter method for guardian of junior account'''
     return self._guardian

 @guardian.setter
 def guardian(self, g):
     '''setter method for employee name'''
     self._guardian = g
```

Recall that a variable begins to exist only when it is assigned some value. Thus, when the mutator method for `_guardian` is invoked on the `JuniorAccount` object, the `JuniorAccount` object starts to have the instance variable, `_guardian`.

These statements:

```
ja = JuniorAccount(600)
ja.guardian = 'Alice' # invoke mutator method first to avoid runtime error
print(ja,'Guardian:', ja.guardian)
```

```
print(ja.withdraw(100))
print(ja,'Guardian:', ja.guardian)
```

produce the output:

```
Account No:   1 Balance: $  600.00 Guardian: Alice
True
Account No:   1 Balance: $  500.00 Guardian: Alice
```

If the method `print` been executed before the mutator method `guardian` is called:

```
ja = JuniorAccount (600)
print(ja,'Guardian:', ja.guardian)
ja.guardian = 'Alice'
```

`ja` will not have the instance variable `_guardian`, and so the executing the method `print` produces an error that there is no attribute (or instance variable) `_guardian`:

```
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\Sl
    print(ja,'Guardian:', ja.guardian)
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\Sl
    return self._guardian
AttributeError: 'JuniorAccount' object has no attribute '_guardian'
```

It is important to note that in Python, a variable begins to exist only when it is assigned some value. Therefore, the reason `JuniorAccount` "inherits" `_accountNumber` and `_balance` defined in the class `SavinsAccount` is

- `JuniorAccount` inherits `__init__` defined in the class `SavinsAccount` and
- `__init__` defined in the class `SavinsAccount` gets executed when the statement `ja = JuniorAccount (600)` is executed.

We can also add class variables and methods to subclasses. For example, `JuniorAccount` objects enjoy additional interest rate of 1 percent. Class variables are defined outside methods.

```
_additionalInterestRate = 0.01

@classmethod
def additionalInterestRate(cls, rate):
    '''class method to update the additional interest rate
    '''
    cls._additionalInterestRate = rate

@classmethod
def getAdditionalInterestRate(cls):
    '''class method to returns additional the interest rate
    '''
    return cls._additionalInterestRate
```

We may continue to add as many as necessary, instance and class variables and instance and class methods that describe a `JuniorAccount` object. For example, the `JuniorAccount` object class can have its own class variable for the default balance.

```
    _defBalance = 200
```

Interestingly, there is no need to define the class methods for this new class variable `_defBalance` as `JuniorAccount` class inherits the class methods **`minBalance`** and **`getMinBalance`**`(cls)` which act as accessor and mutator methods respectively for `_defBalance`.

Furthermore, the expression `type(self)._defBalance` in the statement found in the method `__init__` for `SavingsAccount`:

```
        if  balance is None:
            self._balance = type(self)._defBalance
```

will access the correct class variable `_defBalance` as

- `type(self)` returns `JuniorAccount` if `self` is a `JuniorAccount` object and
- `type(self)` returns `SavingsAccount` if `self` is a `SavingsAccount` object

The following code fragments show two ways to inspect and update the class variables in the classes `SavingsAccount` and `JuniorAccount`.

Using the class names:

```
    print(JuniorAccount.getMinBalance())
    print(SavingsAccount.getMinBalance())
    JuniorAccount.minBalance(250)
    SavingsAccount.minBalance(600)
```

Using the object types:

```
    ja = JuniorAccount('Alice')
    sa = SavingsAccount(300)
    print(type(ja).getMinBalance())
    print(type(sa).getMinBalance())
    type(ja).minBalance(500)
    type(sa).minBalance(700)
```

Research

The execution of this code fragment produces different output in 2 scenarios:

```
sa = SavingsAccount(300)
print(sa)
ja = JuniorAccount('Alice')
print(ja)
sa = SavingsAccount(100)
print(sa)
ja = JuniorAccount('Jane')
print(ja)
sa = SavingsAccount(100)
print(sa)
ja = JuniorAccount('Peter')
print(ja)
```

Scenario 1

The constructor in SavingsAccount references the class variable as `SavingsAccount._nextAccountNumber`:

```
def __init__(self, balance = None):
    self._accountNumer = SavingsAccount._nextAccountNumber
    SavingsAccount._nextAccountNumber += 1
    if  balance is None:
        self._balance = type(self)._defBalance
    else:
        self._balance = balance
```

The code fragment produces this output:

```
Account No:   1 Balance: $  300.00
Account No:   2 Guardian: Alice  Balance: $  200.00
Account No:   3 Balance: $  100.00
Account No:   4 Guardian: Jane  Balance: $  200.00
Account No:   5 Balance: $  100.00
Account No:   6 Guardian: Peter  Balance: $  200.00
```

Scenario 2

The constructor in SavingsAccount references the class variable as `type(self)._nextAccountNumber`:

```
def __init__(self, balance = None):
    self._accountNumer = type(self)._nextAccountNumber
    type(self)._nextAccountNumber += 1
    if  balance is None:
        self._balance = type(self)._defBalance
    else:
        self._balance = balance
```

The code fragment produces this output:

```
Account No:   1 Balance: $  300.00
Account No:   2 Guardian: Alice  Balance: $  200.00
Account No:   2 Balance: $  100.00
Account No:   3 Guardian: Jane  Balance: $  200.00
Account No:   3 Balance: $  100.00
Account No:   4 Guardian: Peter  Balance: $  200.00
```

Find the reason for the difference in output.

Activity 3

a) Define a class variable `_additionalInterestRate` in the class `SeniorAccount`. Assign it the value 0.015.

b) Define a class variable `_maxOverdraft` in the class `SeniorAccount`. Assign it the value 500.
Define two class methods `getMaxOverdraft` and `setMaxOverdraft` in the class `SeniorAccount`. to inspect and update the class variable `_maxOverdraft`.

c) Define accessor method for an instance variable, `_currentOverDraft` in the class `SeniorAccount`.

1.2 Specialisation

Specialisation allows the subclass to override the implementation of the inherited methods by redefining them in the subclass.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 303-307.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

For example, a `JuniorAccount` object must be initialised with a specified `guardian`. Thus, the `JuniorAccount` class must override the inherited constructor so that its has have 3 formal parameters excluding `self` – `accountNumber`, `balance` and `guardian`.

As the constructor in `SavingAccount` initialises the instance variables `_accountNumber` and `_balance`, we should not duplicate the assignment statements in the constructor of the `JuniorAccount` class. Instead, the constructor of the `JuniorAccount` class will invoke the constructor of its superclass, that is, `SavingAccount` class.

```
def __init__(self, guardian, balance = None):
    '''
    Initialise an junior account object
       guardian is required
       balance is optional and is given default value
    '''
    super().__init__(balance)
    self._guardian = guardian
```

`super().__init__(balance)` is a call to invoke the constructor of the superclass on `self` where `self` is the newly created `JuniorAccount` object.

The qualifier `super()` triggers a search for the specified method `__init__` from the superclass instead of from the class of the object itself, *`self`*.

There are 2 cautions when writing the method `__init__` in subclasses:

1. If the statement had been coded as `self.__init__`, then the method is searched from class of `self`, a reference to the object itself. The constructor in subclass will execute infinitely as it is called repeatedly, resulting in runtime error.

   For example, if the statement had been coded as `self.__init__(balance)` in subclass `JuniorAccount` class, invoking it produces a runtime error:

```
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\
    ja = JuniorAccount('Alice')
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\
    self.__init__(balance)
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\
    self.__init__(balance)
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\
    self.__init__(balance)
  [Previous line repeated 992 more times]
RecursionError: maximum recursion depth exceeded
```

2. If the statement `super().__init__` is left out in the `__init__`method for subclass, then the subclass objects will not have the instance variables of the superclass.

   For example, if the statement `super().__init__(balance)` is removed from the `__init__`method for `JuniorAccount` class, then `JuniorAccount` objects will not have the instance variables `_accountNumber` and `_balance`.

   Executing the statements:

```
ja = JuniorAccount('Alice')
print(ja.accountNumber, ja.balance)
```

   produces this error:

```
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU2\bank
    print(ja.accountNumber, ja.balance)
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU2\bank
    return self._accountNumber
AttributeError: 'JuniorAccount' object has no attribute '_accountNumber'
```

A `JuniorAccount` class may wish to have a different implementation for the `withdraw` method. For example, a bank may limit withdrawal amount to not more than $50, unless the guardian is present, in which case, there is no imposed limit.

The `withdraw` method in `JuniorAccount` class must include the 2 cases:

- when the guardian is not present, the withdraw amount is limited and
- when the guardian is present, there is no limit (as per the `withdraw` method in `SavingsAccount`).

```
def withdraw(self, amount, guardian = None):
    '''
    Limit the withdraw amount to 50 in the absence of the guardian.
    Return True if withdraw is successful. Otherwise return False
    '''
    if guardian is None or self._guardian != guardian:
        if amount > 50:
            return False`
    return super().withdraw(amount) # as per superclass SavingsAccount
```

Finally, the `__str__` method must be overridden to include `_guardian`. The `__str__` method in the `JuniorAccount` class will invoke method inherited from the `SavingsAccount` class using `super().__str__` to get the string representation for the part of the `JuniorAccount` object described by `SavingsAccount` class.

```
def __str__(self):
        '''
        Return a string representation of a junior account
        '''
        return super().__str__() + (" Guardian: {:10s}") \
                .format(self._guardian)
```

The examples of overriding so far can be described as **overriding by refinement**. In overriding by refinement, the overriding method in the subclass makes a call to the overridden method in the superclass. We should attempt to override by refinement wherever possible so that we do not repeat the code in the superclass,

When the overriding method in the subclass need not make a call to the overridden method in the superclass, then we say that **overriding is by replacement**.

An example of applying overriding by replacement is when, suppose, we wish to return a string representation of `JuniorAccount` in the order where the guardian appears before the balance. That is,

`accountNumber` , `_guardian` and `_balance` instead of
`accountNumber`, `_balance` and `guardian`.

In this case, we should not call `super().__str__()` as calling the method `__str__` in the superclass returns a string consisting of `_accountNumber` followed by `_balance`.

Instead, we use the accessor methods to get the `accountNumber` and `balance`.

```
    def __str__(self):
        '''
        Return a string representation of a junior account
        '''
        return  ("Account No: {:3d} Guardian: {:10s}  Balance: ${:8.2f}") \
                .format(self.accountNumber, self._guardian, self.balance)
```

Another example of overriding by replacement is the method `deductServiceCharge` in `JuniorAccount` class. As `JuniorAccount` objects are not subjected to service charge, we replace the implementation by the statement **`pass`**.

```
    def deductServiceCharge(self):
        '''
        Does not incur service charge
        '''
        pass
```

Finally, because `JuniorAccount` objects enjoy additional interest rate of 1 percent, `JuniorAccount` must override the **`addInterest`** method.

```
    def addInterest(self):
        '''
        Add the interest based on the interest rate and current balance
        '''
        self.deposit((type(self).getInterestRate()\
                      + type(self)._additionalInterestRate) \
                      * self.balance)
```

Note that

- `type(self)` returns `JuniorAccount`, the class of the object `self`.
- The search for the method `getInterestRate` starts from the `JuniorAccount` class.
- As the method `getInterestRate` is not defined in the `JuniorAccount` class, the Python interpreter searches its superclass, `SavingsAccount`.
- The Python interpreter binds the `getInterestRate` method in `SavingsAccount` to the method call `type(self).getInterestRate()`
- The Python interpreter executes the `getInterestRate` method in `SavingsAccount`

Likewise, the method `deposit` and the accessor method `balance`, inherited from the class `SavingsAccount`, are bounded to the method calls in this program statement as these methods are not defined in the class `SavingsAccount`:

```
        self.deposit((type(self).getInterestRate()\
                      + type(self)._additionalInterestRate) \
                      * self.balance)
```

The search for the class variable `_additionalInterestRate` starts from the `JuniorAccount` class. As the variable is defined in the `JuniorAccount` class, the Python interpreter uses its to compute the amount to deposit.

Activity 4

a) Redefine the method `__init__` in the class `SeniorAccount` to initialise the instance variable `_currentOverDraft` to 0.
b) Refine the method `deposit` so that the overdraft is paid before any part of the deposit amount is added to `_balance` in the `SeniorAccount` object that the method `deposit` is invoked on.
c) Redefine the method `withdraw` so that the method `withdraw` can use an overdraft. The method `withdraw` should be unsuccessful if the withdrawal causes the overdraft amount thus far to exceed the maximum allowable overdraft for `SeniorAccount` objects.
d) Redefine the method `__str__` in the class `SeniorAccount` to return the value of the instance variable `_currentOverDraft`.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 308-317.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

1.3 Multiple Inheritance

Python allows subclasses to inherit from multiple superclasses. Multiple inheritance is useful when there are several optional variables and methods, resulting in several general classes to inherit from, depending on which options are applicable.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 318-320.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Consider this example of multiple inheritance where **`A`** and **`B`** as subclasses of the class `object`, and **`C`** is a subclass of both **`A`** and **`B`**.

```
class A(object):
    def __init__(self):
        super().__init__()
        self._a = 'a'
        print('a')

    def __str__(self):
        return  super().__str__() + self._a

class B(object):
    def __init__(self):
        super().__init__()
        self._b = 'b'
        print('b')

    def __str__(self):
        return super().__str__() + self._b
```

We then define a subclass, **C** of both classes **A** and **B**.

```
class C(A, B):
    def __init__(self):
        super().__init__()
        self._c = 'c'
        print('c')

    def __str__(self):
        return super().__str__() + self._c
```

An object from the class **C** has 3 variables, `_a`, `_b` and `_c`. The subclass **C** overrides the constructor `__init__` and the method `__str__` inherited from both its superclasses, **A** and **B**.

Consider the execution of these 2 statements:

```
c = C()
print(c)
```

The steps listed explain how an object from class `C` that inherits from 2 superclass is constructed.

1. `C()` invokes the method **`__new__`** in the class **`object`** which then calls constructor, **`__init__`** in class **C**.

2. **`__init__`** in class **C** calls its superclass constructor via `super().__init__()`.

   Which superclass's constructor `__init__` should be bounded to this call `super().__init__()`?

   Python 3 uses the Method Resolution Order (MRO) for new style classes which resolves method from left to right, depth-first.

Since class `C` is defined as:

```
class C(A, B)
```

the superclass `A` is rightmost followed by superclass `B`, and so, `super().__init__()` invokes the constructor `__init__` in class `A` and so, `__init__` in class `A` executes next.

Additional Resources

Refer to the following for Method Resolution Order in Multiple Inheritance:

http://python-history.blogspot.sg/2010/06/method-resolution-order.html
https://docs.python.org/3/library/functions.html#super

3. `__init__` in class `A` calls its superclass constructor via `super().__init__()`.

   Note that `super().__init__` does **NOT** call the constructor of superclass of A which is `object`.

   According to the MRO, `super()` in the method `__init__` in class `A` does not refer to the superclass of `A` but to the next superclass of `C` which is class `B`. Thus, the constructor `__init__` in class `B`, the next superclass of `C`, is executed.

4. **`__init__`** in class `B` calls the constructor of the next superclass of class `C` via `super().__init__()`. The next superclass of `C` is `object`.

5. After the constructor **`__init__`** in class `object` completes execution, it returns control of the CPU to its caller, **`__init__`** in class `B`.

6. After the constructor in class `B` prints out `b`, it returns control of the CPU to its caller, **`__init__`** in class `A`.

7. After the constructor in class `A` prints out `a`, it returns control of the CPU to its caller, **`__init__`** in class `C`.

8. **`__init__`** in class C prints out `c`.

Thus, we get the output:

```
b
a
c
```

We can print the MRO of a class using this statement:

```
print(C.__mro__)
```

Output:

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

The same MRO applies to the statement `print(c)`.

1. `print(c)` invokes the method `__str__` in the class `C` which calls its superclass method via `super().__str__()`, and according to the MRO, `super().__str__()` invokes the `__str__` method in class `A` which is bound to the call.

2. `__ str __` in class `A` calls its superclass constructor via `super(). __str__()`. This time, `__str__` in class `B` is bound to the call.

3. `__ str __` in class `B` calls its superclass constructor via `super(). __str__()`. The `__str__` method in class `object` is bound to the call.

4. `__str__()` in class `object` prints out `<__main__.C object at 0x000000002C671D0>` and it returns control of the CPU to its caller, **`__str__`** in class `B`.

5. `__str__()` in class `B` prints out `b` and it returns control of the CPU to its caller, **`__str__`** in class `A`.

6. `__str__()` in class `A` prints out `a` and it returns control of the CPU to its caller, **`__str__`** in class `C`.

7. `__str__()` in class `C` prints out `c`.

Problem specifications requiring multiple inheritances are not as common as single inheritance.

Activity 5

Work out the output of running this module.

```
class A(object):
    def __init__(self):
        print('A')
        super().__init__()
        self._a = 'A'

    def __str__(self):
        return  super().__str__() + self._a

class B(object):
    def __init__(self):
        print('B')
        super().__init__()
        self._b = 'B'


    def __str__(self):
        return super().__str__() + self._b

class C(A, B):
    def __init__(self):
        print('C')
        super().__init__()
        self._c = 'C'


    def __str__(self):
        return super().__str__() + self._c

class D(object):
    def __init__(self):
        print('D')
        super().__init__()
        self._d = 'D'


    def __str__(self):
        return super().__str__() + self._d

class E(C, D):
    def __init__(self):
        print('E')
        super().__init__()
        self._e = 'E'


    def __str__(self):
        return super().__str__() + self._e

e = E()
print(e)
print(E.__mro__)
```

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 321-327.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Chapter 2 Defining Abstract (super)classes

2.1 Programming by contract

In chapter 1 of this Study Unit, we focused on the inheritance relationship, defining first the superclass and then its subclass to reuse the implementations of the superclass.

The inheritance relationship is useful when the problem specification describes a variety of objects which are variations of a general type. Chapter 1 focused on superclasses which themselves can be used to create objects.

For example, the general type `SavingsAccount` class will create `SavingsAccount` objects for customers between ages 18 and 60, the subclass `JuniorAccount` class will create `JuniorAccount` objects for customers under age 18 and the subclass `SeniorAccount` class will create `SeniorAccount` objects for customers above age 60.

This chapter focuses on the case when a superclass identified in problem specification is not meant to create for objects for the application; the superclass simply defines a generic type to contain the common variables and methods in the variety of objects.

For example, a `Vehicle` class is a general type to encapsulate the variables and methods of a vehicle. However, the objects in the applications are specific types of `Vehicle`, such as cars, mini-vans and trucks. In this case, we make the `Vehicle` class as an abstract class to prevent vehicles from be created. Rather, only cars, mini-vans and trucks should be created.

An abstract class has the following features:

- It inherits from the class `ABC` from the `abc` module where `ABC` is the acronym for abstract base class.

  Alternatively, we can specify its metaclass to be `ABCMeta`. We will not use the metaclass option in this course.

- The Python interpreter does not allow objects to be created from an abstract class.

  Attempt to create objects from an abstract class produces a runtime error.

- Abstract methods or methods with no or partial implementation can be defined in an abstract class.

  A subclass that inherits from an abstract superclass promises to override the abstract methods in its superclass. If the subclass does not override the abstract methods in its superclass, then it also becomes an abstract class.

Abstract methods in an abstract superclass are a means to impose that its subclasses override them either by an implementation or by completing the partial implementation. We say the subclasses are bounded contractually to implement the abstract methods.

Activity 6

Give reason whether the superclass(es) for each problem statement should be made abstract:

a) A module is either a core or an elective.
b) Teaching staff has regular teaching evaluations. Management staff has a portfolio. Besides teaching and management staff, there are other types of staff.
c) A student is either a full-time or a part-time student and either a local student or a foreign student.

2.2 Defining abstract classes

To define an abstract class,

- import the class `ABC` and the decorator `abstractmethod`,
- make `ABC` the superclass of the abstract class being defined
- define abstract methods, if any.
    - Abstract methods are defined with the decorator `abstractmethod`
    - Provide a body to the abstract method or if empty, use the `pass` statement

  Note that expressions can include method calls to abstract methods. Methods that contain method calls to abstract methods are designed with the template method pattern which will be illustrated in this section.
- implement the constructor, instance and class variables and their methods as usual.

Additional Resources

Refer to the following for Abstract Base Class:

https://www.python-course.eu/python3_abstract_classes.php
https://docs.python.org/3/library/abc.html

The Vehicle class is defined here:

```
import datetime
from abc import ABC, abstractmethod # added

class Vehicle(ABC): # Vehicle is subclass of ABC

    def __init__(self, regNum, make, model, cap , yearMfg= None, \
                 rate = None, onDiscount = None):
        self._registrationNumber = regNum
        self._make = make
        self._model = model
        self._capacity = cap
        self._yearOfManufacture = datetime.datetime.now().year \
        if yearMfg is None else yearMfg
        self.rate = 100 if rate is None else rate
        self._onDiscount = False if onDiscount is None else onDiscount

    @property
    def registrationNumber(self):
        return self._registrationNumber

    @property
    def make(self):
        return self._make

    @property
    def model(self):
        return self._model

    @property
    def yearOfManufacture(self):
        return self._yearOfManufacture

    def age(self):
        return datetime.datetime.now().year - self._yearOfManufacture

    @property
    def capacity(self):
        return self._capacity

    @abstractmethod # abstract method
    def computeDiscountRate(self):
        pass

    @property
    def rate(self): # abstract method computeDiscountRate can be invoked as if
```

```
                    # as if it is non abstract
        return self._rate * (1 - self.computeDiscountRate() ) if self._onDiscount
else self._rate

    @rate.setter
    def rate(self, value):
        self._rate = 70 if value < 70 else 300 if value > 300 else value

    @property
    def onDiscount(self):
        return self._onDiscount

    def changeOnDiscount(self):
        self._onDiscount = not self._onDiscount

    def __str__(self):
        return ("Registration No: {:9s} Make: {:15s} Model:  {:10s} \n" \
                + "Engine Capacity: {:4d}   Year of manufacture: {:4d} " \
                + "Rental Rate: $ {:5.2f} On Discount: {!r}\n"). \
                format(self._registrationNumber, self._make, self._model,
                        self._capacity,
                        self._yearOfManufacture, self.rate, self._onDiscount)
```

The method

```
    def rate(self): # abstract method computeDiscountRate can be invoked
                    # as if it is non abstract
        return self._rate * (1 - self.computeDiscountRate() ) if self._onDiscount
else self._rate
```

is an example of the template method pattern. This pattern allows different subclasses to provide an implementation for an abstract method (`computeDiscountRate`, in the case) without changing the structure of the algorithm in the method itself (the accessor method, **`rate`**, in the case).

Activity 7

a) Define an abstract class `Employee` with
   - a class variable: `_currentId` which is initialised to 0
   - instance variables:
     - `_id` (auto-generated, a running number from 1, using the class variable `_currentId`),
     - `_name` – name of employee, cannot be `None`,
     - `_dateJoined` – initialised to today's date if no input is provided when the object is created.

b) Define the method `__init__` with formal parameters, `name` and `dateJoined` (optional)

c) The accessor methods for name and data joined

d) Define two abstract methods
   - `salary` without formal parameter
   - `incrementSalary` with a formal parameter, `percent`

e) Define an instance method `yearsOfService` with no formal parameter and returns the number of years of service, computed from today's date and `_dateJoined`.

f) Define the method `_str__` that returns a string representation of the object, including `salary` and `yearsOfService`

2.3 Subclass of An Abstract Class

A subclass of an abstract class is defined in the same manner as how a subclass of a non-abstract superclass is defined, except that the subclass is required to override all abstract methods in the superclass.

We define a subclass, **`Car`** of the abstract superclass, `Vehicle`:

```
from SU2.veh import Vehicle

class Car(Vehicle):
    pass


if __name__ == '__main__':
    v1 = Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
```

Executing the statement causes an error as the abstract method **`computeDiscountRate`** has not been overridden. This resulted in **`Car`** becoming an abstract class itself.

```
<terminated> veh.py [C:\Program Files\Python36\python.exe]
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU2\veh.py", line 69, in <module>
    v1 = Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
TypeError: Can't instantiate abstract class Car with abstract methods computeDiscountRate
```

The abstract method must be overridden as every subclass intended for object creation promises to implement all abstract methods in its superclass.

The abstract method **computeDiscountRate** is implemented in the class **Car** here:

```
class Car(Vehicle):

    def computeDiscountRate(self):
        age =  self.age()
        return 0.05 if age <= 2 else 0.10 if age <= 5 else 0.15


if __name__ == '__main__':
    v1 = Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 520.0, True)
    print(v1)
```

Executing the statements now produces this output:

```
<terminated> car.py [C:\Program Files\Python36\python.exe]
Registration No: SX1234Y   Make: Toyota          Model:  Prius
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 285.00 On Discount: True
```

From the output, note that:

- The **Car** object created has rate 300 (upper limit) since 520 exceeds 300.
- As the **Car** object is on discount, and the **Car** object is 2 years old, **computeDiscountRate** returns 0.05 (or 5%). Hence, the rental rate is given a 5% discount and so the method **rate** returns 285.

Suppose we have 2 other subclasses of **Vehicle**:

```
class MiniVan(Vehicle):
    def computeDiscountRate(self):
        return 0.1

class Truck(Vehicle):
    def computeDiscountRate(self):
        return 0.1 if self.capacity <= 3000 else 0.05
```

Running the statements:

```
    v2 = MiniVan("PC2222F", 'Ford', 'Transit', 3700, 2017, 180, True)
    print(v2)
    v3 = Truck("QA7777G", 'Isuzu', 'NPR-HD', 5200, 2017, 300, True)
    print(v3)
```

produces this output:

```
Registration No: PC2222F   Make: Ford           Model:  Transit
Engine Capacity: 3700   Year of manufacture: 2017   Rental Rate: $ 162.00 On Discount: True

Registration No: QA7777G   Make: Isuzu          Model:  NPR-HD
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 285.00 On Discount: True
```

Notice that the computation of the rental rate uses the **`computeDiscountRate`** method in the class of the object. Therefore, as `v2` references a `MiniVan` object, the statement `print(v2)` resulted in the execution of the **`computeDiscountRate`** method in the `MiniVan` class. As `v3` references a `Truck` object, the statement `print(v3)` resulted in the execution of the **`computeDiscountRate`** method in the `Truck` class.

The template method design pattern allows different subclasses to define the implementation for an abstract method (`computeDiscountRate`, in the case) without changing the structure of the algorithm in the method itself (the accessor method, **`rate`**, in the case).

Furthermore, the template method pattern applies polymorphism as the meaning of the expression, *`self`*`.computeDiscountRate()` takes different meanings, depending on which subclass object the reference `self` refers to at the time of execution.

Activity 8

a) Define a subclass PartTimeEmployee of the abstract class `Employee` with
   - two class variables:
     - `_standardRate` which is initialised to 20 and
     - `_baseSalary` which is initialised to 500
   - instance variables:
     - `_hours` - initialised to 0,
     - `_rate` - initialised to today's date if,
     - `_dateJoined` - initialised to the class variable `_standardRate` if no input is given when the object is created

b) Define the method `__init__` with formal parameters, `name`, `dateJoined` (optional) and `rate` (optional).

c) Define two methods to update the instance variable `_hours`
   - `addHours` with a formal parameter, `hours` to add to the instance variable `_hours`
   - `resetHours` without formal parameter, to set the instance variable `_hours` to 0

d) Implement the two abstract methods
   - `salary` – the salary of a parttime employee has 2 components:
     - `_baseSalary - multiply` by 1.2 if `_hours` is at least 20.
     - the product of `_hours` and `_rate`
   - `incrementSalary` with a formal parameter, `percent`. Increase `_rate` by the given percentage, `percent`.

e) Define the method `_str__` that returns a string representation of the object, including `_rate` and `_hours`.

Test your classes with this code fragment:

```
p1 = PartTimeEmployee('Jane', datetime.date(2017, 6, 15))
print(p1)
p1.addHours(10)
print(p1)
p1.addHours(10)
print(p1)
p1.incrementSalary(10)
print(p1)
p1.resetHours()
print(p1)
```

Output:

```
ID:  1 Name: Jane Joined: 15 Dec 2017 Year  Service:  0 Salary: $ 500.00 Rate: 20.0 Hours:  0.0
ID:  1 Name: Jane Joined: 15 Dec 2017 Year  Service:  0 Salary: $ 700.00 Rate: 20.0 Hours: 10.0
ID:  1 Name: Jane Joined: 15 Dec 2017 Year  Service:  0 Salary: $1000.00 Rate: 20.0 Hours: 20.0
ID:  1 Name: Jane Joined: 15 Dec 2017 Year  Service:  0 Salary: $1040.00 Rate: 22.0 Hours: 20.0
ID:  1 Name: Jane Joined: 15 Dec 2017 Year  Service:  0 Salary: $ 500.00 Rate: 22.0 Hours:  0.0
```

Summary

A superclass can be a user-defined class when, in a problem specification, a variety of objects are identified as variations of a general type. In such a case, the class for the general type is first defined. After which, the classes for the variations are defined as its subclasses.

A subclass may either augment the superclass by defining new variables or new methods, or specialise the superclass by redefining the implementation of the methods inherited from the superclass. Specialising the superclass allows us to write polymorphic statements

Python allows subclasses to inherit from multiple superclasses. Python applies Method Resolution Order (MRO) to resolve which methods of superclasses are bounded to a method call.

An abstract class is a generic class that includes the common variables and methods in the variety of objects in problem specification. However, an abstract class is not meant to create for objects required in the application. Attempt to do so will produce a runtime error.

Abstract methods allow us to write methods that apply the template method design pattern. This design pattern allows different subclasses to define an implementation for an abstract method without changing the structure of the algorithm in a method that invokes the abstract method.

References

Goldwasser, M. H., & Letscher, D. (2014). *Object-Oriented Programming in Python [pdf version]. Retrieved from* http://cs.slu.edu/~goldwamh/oopp/

Formative Assessment

1. Identify the correct pair of subclass and superclass in this problem statement: The supermarket stocks 2 types of items: perishable items and non-perishable items.

A) subclass: item, superclass: stock
@ Incorrect. item is not a specialized stock. Refer to Section 1.1 Python 3 Classes, Study Unit 2.

B) subclass: item, superclass: supermarket
@ Incorrect. item is not a specialized supermarket. Refer to Section 1.1 Python 3 Classes, Study Unit 2.

*C) subclass: non-perishable, superclass: item
@ Correct. non-perishable is a specialized item. Refer to Section 1.1 Python 3 Classes, Study Unit 2.

D) subclass: perishable, superclass: non-perishable
@ Incorrect. perishable is not a specialized non-perishable. Refer to Section 1.1 Python 3 Classes, Study Unit 2.

2. Which following example is NOT augmentation?

A) Adding a new instance variable
@ Incorrect. This is an example of augmentation. Refer to Section 1.2 Augmentation, Study Unit 2.

B) Adding a new class variable
@ Incorrect. This is an example of augmentation. Refer to Section 1.2 Augmentation, Study Unit 2.

C) Adding a new instance method
@ Incorrect. This is an example of augmentation. Refer to Section 1.2 Augmentation, Study Unit 1.

*D) Refining an inherited instance method
@ Correct. This is not an example of augmentation. Refer to Section 1.2 Augmentation, Study Unit 2.

3. Which following example is NOT specialisation?

A) Overriding by replacement
@ Incorrect. This is an example of specialisation. Refer to Section 1.2 Specialisation, Study Unit 2.

B) Overriding by refinement
@ Incorrect. This is an example of specialisation. Refer to Section 1.2 Specialisation, Study Unit 2.

*C) Accessing a class variable with `type(self)`.
@ Correct. This is not an example of specialization. Refer to Section 1.2 Specialisation, Study Unit 2.

D) Giving an inherited method a different implementation
@ Incorrect. This is an example of specialisation. Refer to Section 1.2 Specialisation, Study Unit 2.

4. State the correct method resolution order for class `D`, given the class definitions:

```
class A(object):
    pass

class B(object):
    pass

class C(B):
    pass
):
class D(C, A):
    pass
```

A) `D, C, A, B, object`
@ Incorrect. Python 3 uses the Method Resolution Order (MRO) for new style classes which resolves method from left to right, depth-first. Refer to 1.3 Multiple Inheritance, Study Unit 2.

*B) `D, C, B, A, object`
@ Correct. Python 3 uses the Method Resolution Order (MRO) for new style classes which resolves method from left to right, depth-first. Refer to 1.3 Multiple Inheritance, Study Unit 2.

C) `object, A, B, C, D`
@ Incorrect. This is-a relationship is incorrect. Peter is an occurrence or a Manager object. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 2.

D) `object, B, A, C, D`
@ Incorrect. This is-a relationship is incorrect. Peter is an occurrence or a Manager object. Refer to Section 1.2.1 Has-A and Is-A relationships, Study Unit 2.

5. Which characteristic does not belong to a Python abstract class?

*A) A Python abstract class inherits from the class `ABCMeta` from the `abc` module.
@ Correct. This characteristic does not belong to a Python abstract class. Refer to Section 2.1 Programming by contract, Study Unit 2.

B) Attempt to create objects from an abstract class produces a runtime error.
@ Incorrect. This characteristic belongs to a Python abstract class. Refer to Section 2.1 Programming by contract, Study Unit 2.

C) Abstract methods in an abstract superclass are a means to impose that its subclasses override them
@ Incorrect. This characteristic belongs to a Python abstract class. Refer to Section 2.1 Programming by contract, Study Unit 2.

D) The constructor, instance and class variables and their methods of a Python abstract class are implemented in the usual manner.
@ Incorrect. This characteristic belongs to a Python abstract class. Refer to Section 2.2 Defining abstract classes, Study Unit 2.

6. Which statement is false about abstract method?

A) Having an abstract method makes a class abstract.
@ Incorrect. The statement is true. Refer to Section 2.1 Programming by contract, Study Unit 2.

B) Not implementing an abstract method defined in a superclass makes a class abstract.
@ Incorrect. The statement is true. Refer to Section 2.1 Programming by contract, Study Unit 2.

*C) Every abstract class has one or more abstract method.
@ Correct. The statement is false. Refer to Section 2.1 Programming by contract, Study Unit 2.

D) No object can be created from a class that has abstract methods.
@ Incorrect. The statement is true. Refer to Section 2.1 Programming by contract, Study Unit 2.

7. What is NOT a purpose of the template method design pattern?

A) It allows subclass to give a different interpretation of an algorithm
@ Incorrect. This is a purpose of the template method design pattern. Refer to Section 2.3 Subclass of An Abstract Class, Study Unit 2.

*B) Ensures that the algorithm for the method is never changed
@ Correct. This is NOT a purpose of the template method design pattern. Refer to Section 2.3 Subclass of An Abstract Class, Study Unit 2.

C) It allows polymorphic expression to be written.
@ Incorrect. This is a purpose of the template method design pattern. Refer to Section 2.3 Subclass of An Abstract Class, Study Unit 2.

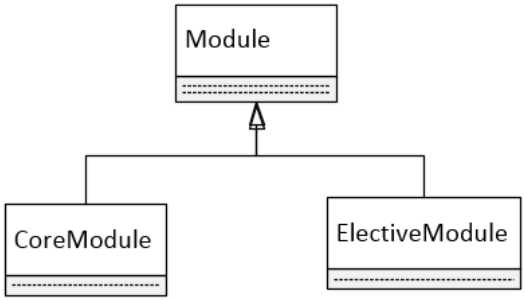
D) It allows the superclass to define the structure of an algorithm.
@ Incorrect. This is a purpose of the template method design pattern. Refer to Section 2.3 Subclass of An Abstract Class, Study Unit 2.

View the answers at the end of this study unit.

Solutions or Suggested Answers

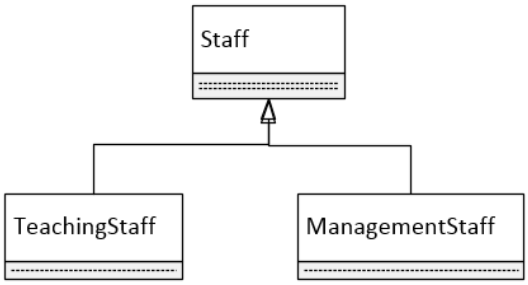
Activity 1

a) Superclass - Module
Subclasses – CoreModule, Elective.Module

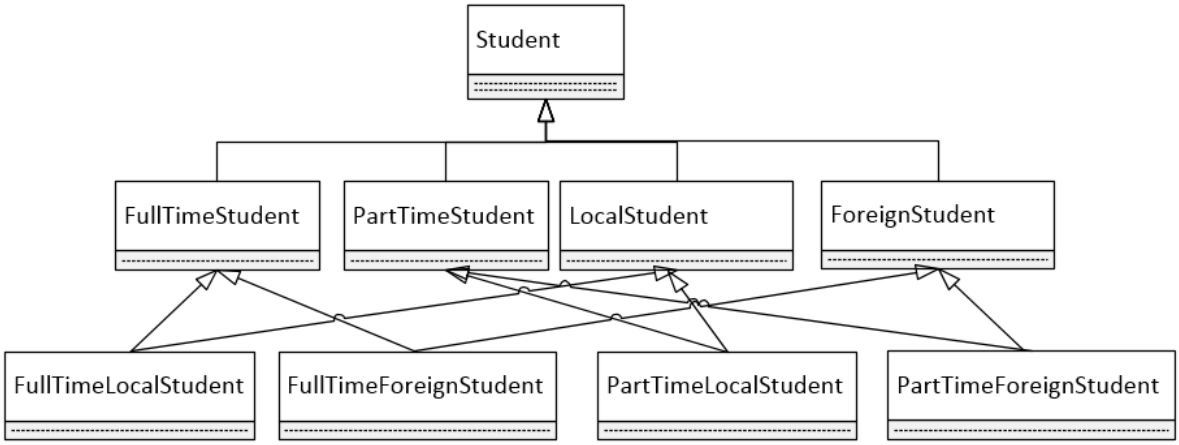


b) Superclass – Staff

Subclasses – TeachingStaff, ManagementStaff



c) Superclass: Student
Subclasses – FullTimeStudent, PartTimeStudent , LocalStudent, ForeignStudent
Subclases: FullTimeLocalStudent, FullTimeForeignStudent, PartTimeLocalStudent, PartTimeForeignStudent,



Activity 2

```
#a)
class SeniorAccount(SavingsAccount):
    pass

#b)
sa1 = SeniorAccount()
sa2 = SeniorAccount(700)

#c)
print(sa1)
print(sa2)

#d)
sa1.deposit(100)
sa2.withdraw(55)
print(sa1)
print(sa2)
```

Activity 3

```
#a)
    _additionalInterestRate = 0.015

#b)
    _maxOverdraft = 500

    @classmethod
    def setMaxOverdraft (cls, amt):
        cls._maxOverdraft = amt

    @classmethod
    def getMaxOverdraft (cls):
        return cls._maxOverdraft
```

```
#c)
    @property
    def currentOverDraft(self):
        return self._currentOverDraft
```

Activity 4

```
#a)
  def __init__(self, balance = None):
        super().__init__(balance)
        self._currentOverDraft = 0


#b)
    def deposit(self, amt):
        if amt > 0:
            if super().deposit (amt - self._currentOverDraft) \
            or amt - self._currentOverDraft == 0:
                self._currentOverDraft = 0
            else:
                self._currentOverDraft -= amt
            return True
        return False

#c)
    def withdraw(self, amt):
        if self._balance \
        + type(self)._maxOverdraft \
        - self._currentOverDraft \
        - amt < 0:
                return False
        if not super().withdraw(amt):
            self._currentOverDraft += amt - self._balance
            self._balance = 0
        return True

#d)
    def __str__(self):
        return super().__str__() + (" Current Overdraft: ${:6.2f}") \
                .format(self._currentOverDraft)
```

Activity 5

```
E
C
A
B
D
<__main__.E object at 0x0000000002CC07B8>DBACE
(<class '__main__.E'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class '__main__.D'>, <class 'object'>)
```

Activity 6

a) Superclass - Module (Abstract) as a Module object is an object from either of the subclasses.

b) Superclass – Staff (Non abstract) as there are other staff who are not teaching or management staff

c) Superclass: Student (Abstract)
Subclasses – FullTimeStudent, PartTimeStudent , LocalStudent, ForeignStudent (all Abstract)
As the problem statement states student is either PartTime or FullTime and either a local or a foreign student at the same time.

Activity 7

```
import datetime
from abc import ABC, abstractmethod

class Employee(ABC):

#a)
    _currentId = 0

#b)

    def __init__(self, name, dateJoined = None):
        Employee._currentId += 1
        self._id = Employee._currentId
        self._name  = name
        self._dateJoined  = \
        datetime.date.today() if dateJoined is None\
        else dateJoined

#c)
    @property
    def name(self):
        return self._name

    @property
    def dateJoined(self):
        return self._dateJoined

#d)
    @abstractmethod
    def salary(self):
        pass

    @abstractmethod
    def incrementSalary (self, percent):
        pass
```

```
#e)
    def yearsOfService(self):
        today = datetime.date.today()
        currentDateJoined = datetime.date(today.year, \
                                          self._dateJoined.month, \
                                          self._dateJoined.day )
        return (today.year - self._dateJoined.year) \
            - (1 if today < currentDateJoined else 0)

#f)
    def __str__(self):
        return ("ID: {:2d} Name: {:} "\
                + "Joined: {:%#d %b %Y} "\
                + ("Years " if self.yearsOfService() > 1 else "Year  ")\
                + "Service: {:2d} Salary: ${:7.2f} ").\
                format(self._id, self._name, self._dateJoined, \
                       self.yearsOfService(), self.salary())
```

Activity 8

```
  class PartTimeEmployee(Employee):

#a)
    _standardRate = 20
    _baseSalary = 500

#b)
    def __init__(self, name, dateJoined = None, rate = None):
        super().__init__(name, dateJoined)
        self._hours = 0
        self._rate  = \
        PartTimeEmployee._standardRate if rate is None else rate

#c)
    def addHours(self, hours):
        self._hours += hours

    def resetHours(self):
        self._hours = 0

#d)
    def salary(self):
        factor = 1 if self._hours < 20 else 1.2
        return type(self)._baseSalary * factor + self._hours * self._rate

    def incrementSalary (self, percent):
        self._rate *= (1 + percent/100)

#e)
    def __str__(self):
        return super().__str__() + ("Rate: {:4.1f} Hours: {:4.1f} ").\
                format(self._rate, self._hours)
```

STUDY UNIT 3

OBJECT COMPOSITION WITH COLLECTIONS

Learning Outcomes

By the end of this unit, you should be able to:

1. Implement a class that manages collection of objects from a single class
2. Implement a class that manages collection of objects from subclasses and a concrete superclass
3. Implement a class that manages collection of objects from subclasses and an abstract superclass
4. Apply polymorphism in the management of collection
5. Demonstrate the Template Method Pattern in Inheritance
6. Demonstrate the Strategy Pattern in Object Composition

Overview

Chapter 1 of this study unit focuses on the management of a mutable collection of objects. The discussion is in 3 parts – when objects in the collection are objects from a single class, when objects in the collection are objects from a concrete superclass and its subclasses and finally, when objects in the collection are objects from subclasses of an abstract superclass.

Each part covers the standard services for collection management and each part builds on demonstrating a variety of services involving the collection.

Chapter 2 covers two design patterns, the template method pattern and strategy pattern, applicable to inheritance and object composition which are two of the pillars of object-oriented programming. In particular, the patterns are demonstrated and differentiated.

This study unit does not cover any chapter of the textbook. However, as there are several online resources to read, it is estimated that the student will spend about 6 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 Managing a Collection of User Objects:

In object composition, an attribute of an object can be a collection, e.g., the class `Company` *has a* collection of `Employee` objects.

If a collection is mutable, a `list` or a `dict` is used to implement the collection.

There are standard services a class that manages mutable collections must provide e.g., to retrieve a specific object in the collection, to add or remove an object from the collection and to list the objects in the collection.

If the collection is immutable, that is, the collection does not get updated once it is created, an immutable type such as `tuple` is used to implement the collection. A class that manages immutable collections must provide only service for inspection, thus, it implements a subset of methods of compared to classes with mutable collections.

This study unit focuses on mutable collection.

1.1 Managing a Collection of Objects from a Single Class

We define a class `Company` to illustrate how a class manages a collection.

A `Company` object has a mutable collection of `Employee` objects, implemented as a `list`. A `Company` object must respond to the service calls to maintain the collection of `Employee` objects and so will respond to the following service calls:

1. Locate an `Employee` object by his employee number and return it. The employee number is an identifier.

   The method should return `None` if there is no such employee in the `list`

2. Add an `Employee` object to the `list`.

   The method returns `True` if the `Employee` object is added. No two employees can have the same employee number, The method returns `False` otherwise.

3. Remove an `Employee` object with a given employee number from the `list`

   The method returns `True` if the identified `Employee` object is removed. The method returns `False` otherwise.

4. Return a string representation of `Employee` objects in the `list`

The class `Employee` defined in Study Unit 1 is reproduced here:

```
class Employee:
    _defSal = 2000
    _maxSalary = 15000
    _minSalary = 1200
    _defPerf = 1
    _perfFactor = {5:2, 4:1.2, 3:0.5}
    _nextNumber = 1

    @classmethod
    def addPerfFactor(cls, performanceLevel, factor):
        '''
        cls._perfFactor[performanceLevel] = factor

    @classmethod
    def removePerfFactor(cls, performanceLevel):
        '''
        return cls._perfFactor.pop(performanceLevel, 0)

    def __init__(self, n, salary = None, performance = None, supervisor= None):
        self._name = n
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def empNum(self):
        return self._empNum

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, value):
        self._salary = \
        Employee._minSalary if  value < Employee._minSalary \
        else Employee._maxSalary if value > Employee._maxSalary \
        else value

    def performance(self, value):
        self._performance = value

    performance = property(None, performance)
```

```
    def calculateBonus(self):
        return  Employee._perfFactor.get(self._performance , 0)  * self._salary

    def increaseSalaryBy(self, percent = 0, base = 20):
        self.salary = self._salary * (1 + percent) + base

    @property
    def supervisor(self):
        return self._supervisor

    @supervisor.setter
    def supervisor(self, supervisor):
        self._supervisor = supervisor

    def __str__(self):
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
                + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} Supervisor: {:}") \
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor._name)
```

For simplicity, the class `Company` has just 2 attributes:

- company name, `_name`
- collection of employees, `_employees`

The constructor for the class `Company` requires one formal parameter to initialise the name of the `Company` object. The collection of employees will be an empty `list`.

```
class Company (object):
    def __init__(self, n):
        self._name = n
        self._employees = []
```

The class `Company` allows for the inspection and update to the company name via the decorator *`@property`*. However, the collection of employees is managed with collection methods.

```
    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, n):
        self._name = n
```

The class `Company` implements these collection methods to respond to the 4 services listed earlier:

1. Locate an `Employee` object in `_employees`

   Locating an object in a collection is a standard collection method to access one element in a collection, whether mutable or immutable. Locating an object can be compared with the accessor method for attribute of scalar type. Since there are many objects in the collection, this method requires a formal argument, the identifier of the object.

   The identifier to locate an `Employee` object is the employee number, a number that is unique for each employee.

   The method inspects each `Employee` objects in the collection, compares employee number of each inspected object with the given employee number. If the numbers match, that `Employee` object is returned. If there is no match in the whole collection, the method returns None.

```
def locate(self, num):
    '''
    Locate an Employee object by his employee number and
    return iti f the employee number matches num.
    Return None if there is no such employee in the collection.
    '''
    for e in self._employees:
        if e.empNum == num:
            return e
    return None
```

2. Add an `Employee` object to `_employees`

   Adding an object into a collection is another standard collection method to update a mutable collection. Adding an object into a collection can be compared with the mutator method for scalar type.

   However, unlike the mutator method which replaces the previous value with a new value, as a a collection can store many objects, the previous value (objects) does not get over-written. Instead, the new value (an object) is appended or added to the collection.

   As an employee number is unique, the method uses the method `locate` to determine whether the employee number of the `Employee` object to be added to the collection is a duplicate employee number. If the employee number

already belongs to an existing `Employee` object in `_employees`, then that `Employee` object should not be added to the collection.

If the employee number is not a duplicate, the `Employee` object is added and the method returns True. Otherwise, the method returns False.

```
def add(self, e):
    '''
    Add an Employee object to the list if the employee number
    is not an employee number of an existing Employee object.
    Return True if the Employee object is added and return
    False otherwise.
    '''
    if self.locate(e.empNum) is None:
        self._employees.append(e)
        return True
    else:
        return False
```

3. Remove an `Employee` object from `_employees`

Removing an object from a collection is another standard collection method to update a mutable collection. Removing an object from a collection can be compared with the mutator method for scalar type.

However, unlike the mutator method which replaces a value with a new value, a collection can store many objects, and removing one of objects does not cause the removal of the other objects in the collection.

The method uses the method `locate` with an employee number to identify `Employee` objectin the collection to be removed.

If the employee number belongs to an existing `Employee` object in the list, the method next checks whether the `Employee` object is a supervisor.

Removing a supervisor affects the `Employee` objects being supervised by the (supervisor) `Employee` object.

If the `Employee` object is not a supervisor, it is removed and the method returns True. Otherwise, the method returns False.

```
def remove(self, num):
    '''
    Remove an Employee object with a given employee number from the
    collection.Employee object must not be a supervisor.
    Return True if the Employee object is removed.
    Return False otherwise.
    '''
    s = self.locate(num)
    if s is None:
        return False
```

```
        for e in self._employees: # check if employee is a supervisor
            if e.supervisor == s:
                return False
        self._employees.remove(s)
        return True
```

Good software principles such as the Single Responsibility Principle (SRP) which is discussed in Study Unit 6 propose that each class should model only one thing (e.g, a company) and each method should have a single responsibility. Therefore, the check for whether an `Employee` object is a supervisor should be coded as as another method.

```
    def isSupervisor(self, s):
        for e in self._employees:
            if e.supervisor == s:
                return True
        return False
```

The method `remove` then calls the method `isSupervisor`.

```
    def remove(self, num):
        '''
        Remove an Employee object with a given employee number from the
        Collection. The Employee object must not be a supervisor.
        Return True if the Employee object is removed.
        Return False otherwise.

        '''
        s = self.locate(num)
        if s is None:
            return False
        if self.isSupervisor(s):
            return False
        self._employees.remove(s)
        return True
```

4. Return a string representation of the list of `Employee` objects in the collection, `_employees`

Returning a string representation of the objects in a collection is another standard collection method to inspect the collection, whether the collection is mutable or immutable. The method can be compared with the accessor method for scalar type.

The method iterates through the `Employee` objects in the collection, concatenating the string representation of each `Employee` object to the output string. If the collection is emoty, the method returns the string `No employee`.

```
def listAll(self):
    '''
    Return the string representation of Employee objects in
    the collection
    '''
    s = ''
    for i, emp in enumerate(self._employees, start = 1):
        s += str(i) + ') ' + str(emp) +  '\n'
    return s or 'No employee\n'
```

The built-in method `enumerate` returns a tuple for each iteration of the `for` loop. Each tuple returned is a pair of values – a count (in this case, starting from 1 since `start = 1`) and the corresponding `Employee` object in the list, `_employees`.

Finally, we define the standard magic method `__str__` which returns the string representation of a `Company` object.

The method `__str__` returns the company name and the details of `Employee` objects in its `list`.

```
def __str__(self):
    '''
    Return the string representation of the Company
    '''
    return "\nCompany Name: " + self.name + "\n" + self.listAll()
```

Note that when a class manages a collection of objects from a single class, the collection class will view the objects as having the functionality defined by the interface of the class.

We perform unit testing on the class `Company` to test the provided services.

To start testing, create a `Company` object.

```
C = Company('SUSS')
```

Testing the methods `listAll()` and `__str__`:

```
print(c)
```

Output

```
Company Name: ABC
No employee
```

Testing the method `add`:

```
e1 = Employee("Peter", 3200)
e2 = Employee("Sam", 2500, supervisor = e1)
e3 = Employee("Jane", performance = 3, supervisor = e2)
print('Add Peter', c.add(e1))
print('Add Sam', c.add(e2))
print('Add Jane', c.add(e3))
print(c)

e4 = Employee("Ann",  1000, 4)
print('Add employee number 4', c.add(e4))
print('Add employee number 4 again', c.add(e4))
print(c)
```

Output

```
Company Name: ABC
1) Emp No:   1  Name: Peter       Performance Level: 1  Salary: $ 3200.00 Bonus: $    0.00 Supervisor: None
2) Emp No:   2  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: Peter
3) Emp No:   3  Name: Jane        Performance Level: 3  Salary: $ 2000.00 Bonus: $ 1000.00 Supervisor: Sam

Add employee number 4 True
Add employee number 4 again False

Company Name: ABC
1) Emp No:   1  Name: Peter       Performance Level: 1  Salary: $ 3200.00 Bonus: $    0.00 Supervisor: None
2) Emp No:   2  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: Peter
3) Emp No:   3  Name: Jane        Performance Level: 3  Salary: $ 2000.00 Bonus: $ 1000.00 Supervisor: Sam
4) Emp No:   4  Name: Ann         Performance Level: 4  Salary: $ 1200.00 Bonus: $ 1440.00 Supervisor: None
```

Testing the method `locate`:

```
print('locate 7', c.locate(7))
print('locate 2', c.locate(2))
```

Output

```
locate 7 None
locate 2 Emp No:   2  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: Peter
```

Testing the method `remove`:

```
print('Remove Sam is supervisor', c.remove(2))
print(c)
print('Remove Jane is not supervisor', c.remove(3))
print(c)
```

Output

```
Remove Sam is supervisor False

Company Name: ABC
1) Emp No:   1  Name: Peter       Performance Level: 1  Salary: $ 3200.00 Bonus: $    0.00 Supervisor: None
2) Emp No:   2  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: Peter
3) Emp No:   3  Name: Jane        Performance Level: 3  Salary: $ 2000.00 Bonus: $ 1000.00 Supervisor: Sam
4) Emp No:   4  Name: Ann         Performance Level: 4  Salary: $ 1200.00 Bonus: $ 1440.00 Supervisor: None

Remove Jane is not supervisor True

Company Name: ABC
1) Emp No:   1  Name: Peter       Performance Level: 1  Salary: $ 3200.00 Bonus: $    0.00 Supervisor: None
2) Emp No:   2  Name: Sam         Performance Level: 1  Salary: $ 2500.00 Bonus: $    0.00 Supervisor: Peter
3) Emp No:   4  Name: Ann         Performance Level: 4  Salary: $ 1200.00 Bonus: $ 1440.00 Supervisor: None
```

Once we have tested the classes `Employee` and `Company`, we can write applications which use these classes, such as to manage the employee list of a company as shown here:

```
from SU3.emp import Employee
from SU3.company import Company

def displayMenu():
    '''
    Display menu and read user's option, repeatedly until user enters
    a valid option.
    The method returns the user's valid option.
    '''
    while True:
        print('Menu\n' + \
              '1. Hire Employee\n'+\
              '2. Terminate Employee\n' + \
              '3. List Employees\n' +\
              '0. Exit')

        option = input('Enter option:')
        if '0' <= option<= '3':
            return int(option)
        print('Enter 0 to 3 only')

def pickSupervisor(company):
    '''
    Display the list of employees in company if the collection is not empty.
    Allow user to select an employee as supervisor.
    Return the selected employee or None if no employee is selected.
    '''
    output = company.listAll() or None
    if output is None:
        print('No employee yet. Can't choose a supervisor')
        return None
    print('Choose supervisor or just <enter> if no supervisor\n' + output)
    s = input('Enter supervisor employee number:') or None
    if s is None:
        return None
    return company.locate(int(s))
```

```
def addEmployee(company):
    '''
    Allow user to enter the details of a new employee.
    Employee name is required and the rest optional.
    Default values will be given if data is not supplied for the rest.
    Create employee object with given data and add to collection.
    Return True if an employee is successfully added and False otherwise.
    '''
    name = input('Enter employee name:') or None
    if name is None:
        print('Invalid Name')
        return False
    salary = input('Enter employee salary:') or None
    if not salary is None:
        salary = float(salary)
    perf = input('Enter employee performance:') or None
    if not perf is None:
        perf = int(perf)
    supervisor = pickSupervisor(company)
    company.add(Employee(name, salary, perf, supervisor))
    return True

def removeEmployee(company):
    '''
    Allow user to enter the employee number of the employee to be removed.
    Return True if an employee is successfully removed and False otherwise.
    '''
    num = input('Enter employee number to terminate:') or None
    if num is None:
        print('Invalid Employee Number)
        return False
    return company.remove(int(num))


def listEmployees(company):
    '''
    Print the employees of the company if the collection is non-empty.
    Print the message 'No Employee' otherwise.
    '''
    print(company.listAll())


def main():
    '''
    Create a new company.
    Repeatedly display the menu to allow user to choose option until user chooses
    to exit the application.
    Otherwise, call the appropriate functions and print the result of the chosen
    operation, wherever appropriate.
    '''
    company = Company('SUSS')

    while True:
        option = displayMenu();
        if option == 1:
            if addEmployee(company):
                print("Employee successfully added")
            else:
                print("ADD Employee Error")
```

```
        elif option == 2:
            if removeEmployee(company):
                print("Employee successfully removed")
            else:
                print("REMOVE Employee Error")
        elif option == 3:
            listEmployees(company)
        elif option == 0:
            break;
    print('Application exiting')
```

From this example application, note the each class has a single responsibility:

- The responsibility of the class `Employee` is to model an employee in terms of his data such as name and employee number and services such as computing bonus.
- The responsibility of the class `Company` is to model a company in terms of its data as company name and the collection of employees it has and services such as adding and removing employees.

The responsibility of the module or program, `companyApplication` is to model the business logic of a company. The program has a `Company` object and it implements methods that can define specific company's business polices and practices such as how hiring and terminating employees.

Activity 1

Given the class `Claim`:

```
class Claim:
    def __init__(self, amount,  dateClaimed):
        self._amount = amount
        self._dateClaimed = date

    @property
    def amount(self):
        return self._amount

    @property
    def dateCalimed(self):
        return self._dateClaimed

    def __str__(self):
        return ("Claim:  Amount: ${:6.2f} Claimed On: {:%#d %b %Y}
"). \
                format(self._amount,  self._dateClaimed )
```

Modify the class `Employee` defined in Study Unit 1 is reproduced in this section, to manage a list of claims made under an employee.

a. Modify the constructor for the `Employee` class to add a new instance variable, `_claims` and set it to empty list.
b. Write or modify the following methods:
   - `locate` that searches for a `Claim` object given a date of claim, `datelaimed`.
     The method returns a list of `Claim` objects made on that day. If there is no `Claim` object for on that day, the method returns an empty list.
   - `add` that adds a `Claim` object to the list. .
   - `remove` that removes a `Claim` object given its claimed date and amount. If there are more than one `Claim` object with that date and amount, for simplicity, remove the first matching `Claim` object.
     The method returns True if remove is successful and False otherwise.
   - `listAll` that return a string representation of the `Claim` objects in the list.
   - `__str__` that return a string representation of an `Employee` object.

c. Write code to test the `Employee` class.

1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass

A collection can store objects not only from a single class such as `Employee` objects but also from a variety of classes that have the same interface or from classes in a class hierarchy such as a concrete superclass `SavingsAccount` and its subclass `JuniorAccount`.

For simplicity again, we implement a class `Bank` with two attributes:

- `_name`, the name of the `Bank` object, and
- `_accounts`, a collection of `SavingsAccount` and `JuniorAccount` objects. The collection is mutable and is to be implemented as a `list`.

To be able to make comparisons between the class `Bank` and the class `Company` in Section 1.1, we will make the class `Bank` provide similar service, namely, to maintain the list, `_accounts`, of `SavingsAccount` and `JuniorAccount` objects:

1. Locate an account which is either a `SavingsAccount` or a `JuniorAccount` object by the identifier, the account number and return it.

   The method should return `None` if there is no such account in the `list`

2. Add an account which is either a `SavingsAccount` or a `JuniorAccount` object, to `_accounts`.

   The method returns `True` if the account is added. The method returns `False` otherwise.

3. Remove an account which is either a `SavingsAccount` or a `JuniorAccount` object with an identifying account number, from `_accounts`

   The method returns `True` if the account is removed. The method returns `False` otherwise.

4. Return a string representation of the list of accounts in `_accounts`.

The magic method `__str__` returns the string representation of the `Bank` object which includes its name and the details of account objects in `_accounts`.

In addition, we will expand the behaviour of this collection class, `Bank` to include methods to make service calls to the objects in `_accounts`:

- a method to add the interest earned to each account in `_accounts`.
- a method to deduct a service charge from each account in `_accounts`.

The classes `SavingsAccount` and `JuniorAccount` defined in Study Unit 2 are reproduced here:

```
class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1
    _interestRate = 0.01
    _serviceCharge = 5

    @classmethod
    def interestRate(cls, rate):
        cls._interestRate = rate

    @classmethod
    def getInterestRate(cls):
        return cls._interestRate

    def __init__(self, balance = None):
        self._accountNumer = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        if  balance is None:
            self._balance = SavingsAccount._defBalance
        else:
            self._balance = balance

    @property
    def balance(self):
        return self._balance

    @property
    def accountNumber(self):
        return self._accountNumer

    def deposit(self, amt):
        if amt <= 0:
            return False
        self._balance += amt
        return True

    def withdraw(self, amt):
        if self._balance < amt:
            return False
        self._balance -= amt
        return True

    def addInterest(self):
        self._balance += SavingsAccount._interestRate * self._balance

    def deductServiceCharge(self):
        if self._balance < SavingsAccount._defBalance:
            self._balance -= SavingsAccount._serviceCharge
```

```
    def __str__(self):
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumer, self._balance)


class JuniorAccount(SavingsAccount):
    _additionalInterestRate = 0.01

    @classmethod
    def additionalInterestRate(cls, rate):
        cls._additionalInterestRate = rate

    @classmethod
    def getAdditionalInterestRate(cls):
        return cls._additionalInterestRate

    def __init__(self, guardian, balance = None):
        super().__init__(balance)
        self._guardian = guardian

    @property
    def guardian(self):
        return self._guardian

    @guardian.setter
    def guardian(self, g):
        self._guardian = g


    def withdraw(self, amt, guardian = None):
        if guardian is None or not self._guardian == guardian:
            if amt > 50:
                return False
        return super().withdraw(amt)

    def addInterest(self):
        self.deposit((type(self).getInterestRate() \
                      + self._additionalInterestRate) \
                      * self.balance)

    def deductServiceCharge(self):
        pass

    def __str__(self):
        return  ("Account No: {:3d} Guardian: {:}  Balance: ${:8.2f}") \
                .format(self.accountNumber, self._guardian, self.balance)
```

The constructor for the class `Bank` requires one formal parameter to initialise the name of the `Bank` object. The newly created `Bank` object has an empty accounts list.

```
    def __init__(self, name):
        self._name = name
        self._accounts = []
```

The class `Bank` allows for the inspection and update via the decorator *`@property`* to the bank.

```
@property
def name(self):
    return self._name

@name.setter
def name(self, n):
    self._name = n
```

To manage the collection, the class `Bank` implements methods to respond to the four services listed earlier:

1. Locate an account which is either a `SavingsAccount` or a `JuniorAccount` object by the identifier, the account number and return it.

```
def locate(self, accountNumber):
    '''
    Locate a bank account object by the account number
    and return it.
    Return None if there is no such account in the list
    '''
    for account in self._accounts:
        if account.accountNumber == accountNumber:
            return account
    return None
```

   Notice that the structure of the `for` loop in the method is the same as that for the example method `for` in Section 1.1. The method structure is the same, whether the collection contains objects from a single class or the collection contains objects from several classes.

   `_accounts` stores objects from both the superclass `SavingsAccount` and the subclass `JuniorAccount`. Each object in `_accounts` has the accessor method `accountNumber`, either defined as in the case of the superclass `SavingsAccount` or inherited as in the case of the subclass `JuniorAccount`.

   The account object, either a `SavingsAccount` or a `JuniorAccount` object, that has the matching account number is thus returned.

2. Add an account which is either a `SavingsAccount` or a `JuniorAccount` object, to `_accounts`.

```
def add(self, account):
    '''
    Add a bank account object to the list if the account number
    is not already in the list of bank accounts.
```

```
        Return True if the bank account object is added.
        Return False otherwise.
        '''
        if self.locate(account.accountNumber) is None:
            self._accounts.append(account)
            return True
        else:
            return False
```

Similarly, notice that the method structure is also the same as the method structure of the method `add` in Section 1.1, whether the collection contains objects from a single class or the collection contains objects from several classes.

However, note that the formal parameter, `account` gets a reference that is either a `SavingsAccount` or a `JuniorAccount` object.

The result is `_accounts` stores both `SavingsAccount` and `JuniorAccount` objects.

3. Remove an account which is either a `SavingsAccount` or a `JuniorAccount` object with an identifying account number, from `_accounts`.

```
    def remove(self, num):
        '''
        Remove an bank account object with a given account number from the
        list. The method returns True if the  bank account object is
        removed.The method returns False otherwise.

        '''
        acct = self.locate(num)
        if acct is None:
            return False
        self._accounts.remove(acct)
        return True
```

Again, notice that the method structure is also the same as the method structure of the method `remove` in Section 1.1, whether the collection contains objects from a single class or the collection contains objects from several classes.

However, note that the program statement `self.locate(num)` returns a reference that is either a `SavingsAccount` or a `JuniorAccount` object. Thus, the removed object is either a `SavingsAccount` or a `JuniorAccount` object.

4. Return a string representation of the list of accounts in `_accounts`.

```
def listAll(self):
    '''
    Return the string representation of the list of accounts in
    the list
    '''
    outputString = ''
    for i, account in enumerate(self._accounts, start = 1):
        outputString += str(i) + ') ' + str(account) +  '\n'
    return outputString or 'No Account\n'
```

Again, notice that the method structure is also the same as the method structure of the method `remove` in Section 1.1, whether the collection contains objects from a single class or the collection contains objects from several classes.

However, note that the tuple returned at each iteration of the `for` loop is either a reference to a `SavingsAccount` or a `JuniorAccount` object. Thus, the `__str__` method bounded to `str(account)` is either the `__str__` method in the `SavingsAccount` or in the `JuniorAccount` class, depending on what class of object `account` references. The expression `str(account)` is polymorphic.

This implementation of the method `listAll` also follows the template method pattern which allows subclass to have different implementation of the variant steps of an algorithm. The template method pattern adheres to the Polymorphic Open-Closed Principle which is covered in Study Unit 6. The Polymorphic Open-Closed Principle states that software entities such as classes and methods should be open for extension but closed to modification. The method `listAll` is open for extension resulting from the polymorphic expression but closed to modification as the algorithm structure is fixed.

Program implementations that apply the Polymorphic Open-Closed Principle are easier to maintain. The collection class needs not be modified by changes in specific subclass or when new subclasses are defined.

We next define following methods:

- the method `__str__` returns the string representation of the `Bank` object.

  The method `__str__` returns the bank name and the details of accounts, `SavingsAccount` and `JuniorAccount` objects in its `_accounts`.

```
def __str__(self):
    '''
    Return the string representation of the Bank object
    '''
    return "\nBank Name: " + self.name + "\n" + self.listAll()
```

Again, notice that the method structure is also the same as the method structure of the method `listAll` in Section 1.1, whether the collection contains objects from a single class or the collection contains objects from several classes.

- a method to add interest earned to each account in the `_accounts`.

  The method iterates through the list, `_accounts`, picks up an account which is either a `SavingsAccount` or a `JuniorAccount` object in each iteration, and requests a service call to the account to add interest through the expression `acct.addInterest()`.

```
def addInterest(self):
    '''
    Add interest to every account in the list
    '''
    for acct in self._accounts:
        acct.addInterest()
```

  The `addInterest` method in the expression `acct.addInterest()` is bounded to either the `addInterest` method in the `SavingsAccount` or the specialized in the `JuniorAccount` class, depending on what class of object `account` references. The expression `acct.addInterest()` is polymorphic.

  This implementation also adheres to the Polymorphic Open/Closed Principle.

- a method to deduct service charges from each account in `_accounts`, when applicable.

  The method iterates through the list, `_accounts`, picks up an account which is either a `SavingsAccount` or a `JuniorAccount` object in each iteration, and requests a service call to the account to deduct the service charge through the expression `acct.deductServiceCharge()`.

```
def deductServiceCharge(self):
    '''
    Add interest to every account in the list
    '''
    for acct in self._accounts:
        acct.deductServiceCharge()
```

  Again, the `deductServiceCharge` method in the expression `acct.deductServiceCharge()` is bounded to weither the `deductServiceCharge` method in the `SavingsAccount` or in the `JuniorAccount` class, depending on what class of object `account` references. The expression `acct.deductServiceCharge()` is polymorphic.

  This implementation also adhere to the Polymorphic Open/Closed Principle.

Thus, when a class manages a collection of objects from a class hierarchy with a concrete superclass, the class should view the objects as having the functionality defined by the (common) interface of the concrete superclass so that the methods observe the Polymorphic Open/Closed Principle.

When objects from subclasses and the concrete superclass in the collection are viewed with a common interface, they can be processed using the same program statements as their classes have either inherited or specialised the required methods. The program statements are polymorphic.

The methods bounded to polymorphic statement depend on the object currently being processed, thus allowing for new (specialised) behaviour without modifying the program statements that process the collection. The class with the collection (the client code) need not be modified by changes in specific subclass or when new subclasses are defined.

We perform unit testing on the class `Bank` to test the provided services.

To start off, create a `Bank` object.

```
b = Bank('My Bank')
```

Testing the methods `listAll()` and `__str__`:

```
print(b)
```

Output

```
Bank Name: My Bank
No Account
```

Testing the method `add`:

```
ja = JuniorAccount('Alice')
sa = SavingsAccount(300)
print('Add Junior Account 1:', b.add( ja))
print('Add Savings Account 2:', b.add(sa))
print(b)

print('Add Junior Account 1 again:', b.add( ja))
print(b)
```

Output

```
Add Junior Account 1: True
Add Savings Account 2: True

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  500.00
2) Account No:   2 Balance: $  300.00

Add Junior Account 1 again: False

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  500.00
2) Account No:   2 Balance: $  300.00
```

Testing the method `locate`:

```
print('locate 2:', b.locate(2))
print('locate 3:', b.locate(3))
```

Output

```
locate 2: Account No:   2 Balance: $  300.00
locate 3: None
```

Testing the method `remove`:

```
print('remove 2:', b.remove(2))
print('remove 3:', b.remove(3))
print(b)
```

Output

```
remove 2: True
remove 3: False

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  500.00
```

Testing the method `addInterest` (but first adding more accounts):

Note that `SavingsAccount` objects get 1% interest while `JuniorAccount` objects get an addition 1% interest.

```
print('Add Savings Account 3:', b.add(SavingsAccount(700)))
print('Add Savings Account 4:', b.add(JuniorAccount( 'Sam', 300)))
print('Add Savings Account 5:', b.add(SavingsAccount(200)))
print(b)
b.addInterest()
print('after add interest\n', b)
```

Output

```
Add Savings Account 3: True
Add Savings Account 4: True
Add Savings Account 5: True

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  500.00
2) Account No:   3 Balance: $  700.00
3) Account No:   4 Guardian: Sam  Balance: $  300.00
4) Account No:   5 Balance: $  200.00

after add interest

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  510.00
2) Account No:   3 Balance: $  707.00
3) Account No:   4 Guardian: Sam  Balance: $  306.00
4) Account No:   5 Balance: $  202.00
```

Testing the method `addInterest` (but first adding more accounts):

Note that `SavingsAccount` objects occur a service charge of $5 if the balance falls below the default balance of $500 while `JuniorAccount` objects do not occur any service charge.

```
b.deductServiceCharge()
print('after deduct service charge\n', b)
```

Output

```
after deduct service charge

Bank Name: My Bank
1) Account No:   1 Guardian: Alice  Balance: $  510.00
2) Account No:   3 Balance: $  707.00
3) Account No:   4 Guardian: Sam  Balance: $  306.00
4) Account No:   5 Balance: $  197.00
```

Once we have tested the class `Bank`, we can write an application which uses the classes `SavingsAccount`, `JuniorAccount` and `Bank`, to manage the account list of a bank to

- open and close accounts,
- to list the accounts,
- deposit and withdrawal to an account,
- add interest and to deduct service charges to the accounts in the list

as implemented here, shown without comments:

```
from SU3.bankAccounts import SavingsAccount, JuniorAccount
from SU3.bank import Bank

def displayMenu():
    while True:
        print('Menu\n' + \
              '1. Open Account\n'+\
              '2. Close Account\n' + \
              '3. List Account\n' +\
              '4. Make Deposit\n' +\
              '5. Make Withdrawal\n' +\
              '6. Add Interest\n' +\
              '7. Deduct for Service Charges\n' +\
              '0. Exit')

        option = input('Enter option:')
        if '0' <= option<= '5':
            return int(option)
        print('Enter 0 to 5 only')

def openAccount(bank):
    accountType = int(input("Type of Account - 1. Junior  2. Savings. Enter 1 or
2:"))
    if not (accountType == 1 or accountType == 2):
        print(accountType, ' is invalid account type')
        return False

    if accountType == 1:
        guardian = input('Enter guardian name:') or None
        if guardian is None:
            print('Junior Account needs a guardian')
            return False

    balance = input('Enter amount to start account:') or None
    if not balance is None:
        balance = int(balance)

    if accountType == 1:
        bank.add(JuniorAccount(guardian, balance))
    else:
        bank.add(SavingsAccount(balance))
    return True

def closeAccount(bank):
    num = input('Enter account number to remove:') or None
    if num is None:
        print('Invalid account number')
        return False
    return bank.remove(int(num))

def getAccount(bank):
    num = input('Enter account number:') or None
    if num is None:
        print('Invalid account number')
        return None
    account = bank.locate(int(num))
    if account is None:
        print('Account number does not exist')
```

```
        return None
    return account

def getAmount():
    amount = input('Enter amount:') or None
    if amount is None:
        print('Invalid amount')
        return None
    amount = int(amount)
    if amount <= 0:
        print('Amount must be positive')
        return None
    return amount


def doTransact(bank, transact):
    if not(transact == 'deposit' or 'withdraw'):
        return False
    account = getAccount(bank)
    if account is None:
        return False
    amount = getAmount()
    if amount is None:
        return False
    print('Current balance :$', account.balance)
    if transact == 'deposit':
        result = account.deposit(amount)
    else:
        if type(account) == JuniorAccount and amount > 50:
            guardian = input('Enter guardian name:') or None
            result = account.withdraw(amount, guardian)
        else:
            result =  account.withdraw(amount)
    print('New balance :$', account.balance)
    return result

def main():
    bank = Bank('SUSS Bank')

    while True:
        option = displayMenu();
        if option == 1:
            if openAccount(bank):
                print("Account successfully opened")
            else:
                print("Open Account Error")
        elif option == 2:
            if closeAccount(bank):
                print("Account successfully closed")
            else:
                print("Close Account Error")
        elif option == 3:
            print(bank.listAll())
        elif option == 4:
            print('Deposit successful:', doTransact(bank, 'deposit'))
        elif option == 5:
            print('Withdraw successful:', doTransact(bank, 'withdraw'))
        elif option == 6:
            bank.addInterest()
```

```
        elif option == 7:
            bank.deductServiceCharge()
        elif option == 0:
            break;
    print('Application exiting')
```

From this example application, note again the each class has a single responsibility:

- The responsibility of the class `SavingsAccount` is to model a savings account in terms of its data such as account number and balance and its behaviour such as deposit and withdraw.

- The responsibility of the class `JuniorAccount` is to model a junior account in terms of its data such as account number, guardian and balance and its behaviour such as deposit and withdraw.

- The responsibility of the class `Bank` is to model a bank in terms of the data as a bank name and the collection of accounts, both savings and junior accounts it has and its behaviour such as adding and removing accounts.

The responsibility of the module or program, `bankApplication` is to model the business logic of a specific bank application. The program has a `Bank` object and it implements methods that can define specific bank business polices and practices such as how accounts are opened and closed.

Activity 2

Given subclass of the class `Claim` in Activity 1:

```
class MedicalClaim(Claim):
    def __init__(self, amount,  date, receiptNo, clinic):
        super().__init__(amount, date)
        self._receiptNo = receiptNo
        self._clinic = clinic

    def __str__(self):
        return super().__str__() \
            + ("Receipt No: {:10s} Clinic:  {:20s}"). \
                format(self._receiptNo,  self._clinic )
```

Make changes to the answer to Activity 1, the class `Employee`, to manage a list of claims, **including medical claims** made under an employee. If no change is necessary, indicate so in your answer.

a. The constructor for the `Employee` class.
b. The methods in the class `Employee`:
   - `locate`
   - `add`
   - `remove`
   - `listAll`
   - `__str__`
c. Write code to test the `Employee` class. Create various types of `Claim` objects.

1.3 Managing a Collection of Objects from Subclasses of an Abstract Superclass

A collection can also store objects from classes in a class hierarchy where the superclass is an abstract class such as the abstract superclass `Vehicle` and its subclasses `Car`, `MiniVan` and `Truck`.

Again for simplicity and so as to allow comparisons with examples in Sections 1.1 and 1.2, we implement a class `RentalCompany` with two attributes:

- `_name`, the name of the `RentalCompany` object, and
- `_vehicles`, a collection of objects from subclasses of the abstract class `Vehicle` which are `Car`, `MiniVan` and `Truck` objects. The collection will be implemented as a `list`.

An `RentalCompany` object will respond to similar service calls as seen in Sections 1.1 and 1.2, to maintain the list, `_vehicles`, of `Car`, `MiniVan` and `Truck` objects:

1. Locate a `Vehicle` object which is either a `Car`, a `MiniVan` or a `Truck` object in `_vehicles`, by its identifier, the vehicle registration number and return it.

   The method returns `None` if there is no such vehicle in `_vehicles`.

2. Add a `Vehicle` object which is either a `Car`, a `MiniVan` or a `Truck` object, to `_vehicles`.

   The vehicle registration number must not be duplicated.

   The method returns `True` if the vehicle is added. The method returns `False` otherwise.

3. Remove a vehicle which is either a `Car`, a `MiniVan` or a `Truck` object with an identifying registration number, from `_vehicles`.

   The method returns `True` if the vehicle is removed. The method returns `False` otherwise.

4. Return a string representation of the list of vehicles in `_vehicles`.

In addition, we include

- the method `__str__` to return the string representation of the `RentalCompany` object which includes its name and the details of vehicle objects in `_vehicles`.
- a method to return vehicles in `_vehicles` with a higher driver rate than their vehicle rate.
- a method to reduce the rate of vehicles by a specified rate if the vehicles are above a certain age, rate and discount.

The classes `Vehicle`, `Car`, `MiniVan` and `Truck` defined in Study Unit 2 (where) are reproduced here, but with additional attributes and methods that are indicated with comments:

```
import datetime
from abc import ABC, abstractmethod

class Vehicle(ABC):
    def __init__(self, regNum, make, model, cap, yearMfg= None, \
                 rate = None, onDiscount = None):
```

```
        self._registrationNumber = regNum
        self._make = make
        self._model = model
        self._capacity = cap
        self._yearOfManufacture = datetime.datetime.now().year \
        if yearMfg is None else yearMfg
        self.rate = 100 if rate is None else rate
        self._onDiscount = False if onDiscount is None else onDiscount

    @property
    def registrationNumber(self):
        return self._registrationNumber

    @property
    def make(self):
        return self._make

    @property
    def model(self):
        return self._model

    @property
    def yearOfManufacture(self):
        return self._yearOfManufacture

    def age(self):
        return datetime.datetime.now().year - self._yearOfManufacture

    @property
    def capacity(self):
        return self._capacity

    @abstractmethod
    def computeDiscountRate(self):
        pass

    @abstractmethod
    def getDriverRate(self):                # additional abstract method
        pass

    @property
    def rate(self):
        return self._rate * (1- self.computeDiscountRate() ) \
            if self._onDiscount else self._rate

    @rate.setter
    def rate(self, value):
        self._rate = 70 if value < 70 else 300 if value > 300 else value

    @property
    def onDiscount(self):
        return self._onDiscount

    def changeOnDiscount(self):
        self._onDiscount = not self._onDiscount

    def diffRates(self):                    # additional method
        return self.rate - self.getDriverRate()
```

```
    def __str__(self):
        return ("Registration No: {:9s} Make: {:15s} Model:  {:10s}"\
                + "   On Discount: {!r}\nEngine Capacity: {:4d}"\
                + "   Year of manufacture: {:4d}"
                + "   Rental Rate: $ {:5.2f}   Driver Rate: $ {:5.2f}\n"). \
                format(self._registrationNumber, self._make, self._model, \
                       self._onDiscount, self._capacity, self._yearOfManufacture,\
                       self.rate, self.getDriverRate())

class Car(Vehicle):
    def computeDiscountRate(self):
        age =  self.age()
        return 0.05 if age <= 2 else 0.10 if age <= 5 else 0.15

    def getDriverRate(self):                # additional attribute
        age = self.age()
        return 100 if age <= 3 else 120

class MiniVan(Vehicle):
    def __init__(self, regNum, make, model, cap, handicapAccess = None, \
                 yearMfg= None, rate = None, onDiscount = None):
        super().__init__(regNum, make, model, cap, yearMfg, rate, onDiscount)
                                            # additional attribute
        self._handicapAccess = False if handicapAccess is None else handicapAccess

    @property
    def handicapAccess(self):               # additional method
        return self._handicapAccess

    @handicapAccess.setter
    def handicapAccess(self, value):        # additional method
        self._handicapAccess = value

    def computeDiscountRate(self):
        return 0.1

    def getDriverRate(self):                # additional method
        surcharge = 0
        if self.age() >= 5:
            surcharge +=  0.1 * self.rate
        if  self._handicapAccess:
            surcharge += 25
        return 0.5*self.rate + surcharge

    def __str__(self):                      # additional method
        return ("Registration No: {:9s} Make: {:15s} Model:  {:10s}"\
                 + "  On Discount: {!r}  Handicap Accessible: {!r}\n"\
                 + "Engine Capacity: {:4d}  Year of manufacture: {:4d}"\
                + "  Rental Rate: $ {:5.2f}  Driver Rate: $ {:5.2f}\n"). \
                format(self._registrationNumber, self._make, self._model, \
                       self._handicapAccess, self._onDiscount, self._capacity,\
                       self._yearOfManufacture, self.rate, self.getDriverRate())


class Truck(Vehicle):
    def computeDiscountRate(self):
        return 0.1 if self.capacity <= 3000 else 0.05

    def getDriverRate(self):                # additional method
```

```
        return max(0.25*self.rate, 200)
```

As with the example classes `Company` and `Bank`, the constructor for the class `RentalCompany` requires one formal parameter to initialise the name of the newly created `RentalCompany` object. The newly created `RentalCompany` object has an empty vehicle list, `_vehicles`.

```
    def __init__(self, name):
        self._name = name
        self._vehicles = []
```

The class `RentalCompany` allows for the inspection and update to the rental company name via the decorator *`@property`*.

```
    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, n):
        self._name = n
```

To manage the collection, `_vehicles`, the class `RentalCompany` implements methods to respond to the four services listed earlier:

1. Locate a vehicle which is either a `Car`, a `MiniVan` or a `Truck` object by the identifier, the registration number and return the object.

   ```
       def locate(self, registrationNumber):
           for vehicle in self._vehicles:
               if vehicle.registrationNumber == registrationNumber:
                   return vehicle
           return None
   ```

   Notice that the structure of the `for` loop in the method is the same as that for the example method `for` in Sections 1.1 and 1.2.

   The method structure is also the same, whether the collection contains objects from a single class or the collection contains objects from several classes and whether or not the superclass of the classes is abstract.

   The class of each object in `_vehicles` has the accessor method `registrationNumber`, which is inherited from the abstract superclass `Vehicle`.

   The `Car`, `MiniVan` or `Truck` object that has the matching registration number is returned.

2. Add a vehicle which is either a `Car`, a `MiniVan` or a `Truck` object, to `_vehicles`.

```
def add(self, vehicle):
    if self.locate(vehicle.registrationNumber) is None:
        self._vehicles.append(vehicle)
        return True
    else:
        return False
```

Similarly, notice that the method structure is also the same as the method structure of the method `add` in Sections 1.1 and 1.2, whether the collection contains objects from a single class or the collection contains objects from several classes and whether or not the superclass of the classes is abstract.

Also, note that the formal parameter, `vehicle` gets a reference that is either a `Car`, a `MiniVan` or a `Truck` object Therefore, either a `Car`, a `MiniVan` or a `Truck` object is added to the collection `_vehicles`.

3. Remove a vehicle which is either a `Car`, a `MiniVan` or a `Truck` object with an identifying registration number, from `_vehicles`.

```
def remove(self, num):
    acct = self.locate(num)
    if acct is None:
        return False
    self._vehicles.remove(acct)
    return True
```

Again, notice that the method structure is also the same as the method structure of the method `remove` in Section 1.1, whether the collection contains objects from a single class or the collection contains objects from several classes and whether or not the superclass of the classes is abstract.

The program statement `self.locate(num)` returns a reference that is either a `Car`, a `MiniVan` or a `Truck` object. Thus, the removed object from `_vehicles` is either a `Car`, a `MiniVan` or a `Truck` object.

4. Return a string representation of vehicles in `_vehicles`.

```
def listAll(self):
    outputString = ''
    for i, vehicle in enumerate(self._vehicles, start = 1):
        outputString += str(i) + ') ' + str(vehicle) +  '\n'
    return outputString or 'No Account\n'
```

Again, notice that the method structure is also the same as the method structure of the method `remove` in Sections 1.1 and 1.2, whether the collection

contains objects from a single class or the collection contains objects from several classes and whether or not the superclass of the classes is abstract.

Note also that the tuple returned at each iteration has either a reference to a `Car`, a `MiniVan` or a `Truck` object. Thus, the `__str__` method bounded to `str(vehicle)` is either the `__str__` method in the `Car`, the `MiniVan` or the `Truck` class, depending on what class of object `vehicle` references. The expression `str(vehicle)` is polymorphic. This implementation adheres to the Polymorphic Open/Closed Principle.

Program implementations that apply the Polymorphic Open/Closed Principle are easier to maintain. The collection class needs not be modified when changes are made to the implementation of the subclasses or when new subclasses are defined.

We next define following methods:

- the method `__str__` to return the string representation of the `RentalCompany` object which includes its name and the details of vehicle objects in `_vehicles`.

```
def __str__(self):
    return "\nRental Company Name: " + self.name + "\n" + self.listAll()
```

- a method to return the vehicles in `_vehicles` with higher driver rate than their vehicle rate.

  The method iterates through the list, `_vehicles`, picks up a vehicle which is either a reference to a `Car`, a `MiniVan` or a `Truck` object and in each iteration, calls the service to get the difference between the vehicle rate and the driver rate. If the method returns a negative number, the vehicle to be added to the list of vehicles to be returned.

```
def higherDriverRate(self):
    vehicles = []
    for vehicle in self._vehicles:
        if vehicle.diffRates() < 0:
            vehicles.append(vehicle)
    return vehicles
```

  The subclasses inherit the `diffRate` method from their superclass, `Vehicle`.

  The statement

```
return self.rate - self.getDriverRate()
```

in the `diffRate` method in the class `Vehicle` is polymorphic and applies the template method pattern:

- The accessor method `rate` defined in the superclass `Vehicle` applies the template method pattern where `computeDiscountRate` is implemented in subclasses of `Vehicle`.

```
@property
def rate(self):
    return self._rate * (1- self.computeDiscountRate () ) \
        if self._onDiscount else self._rate
```

  Which subclass' method `computeDiscountRate` is bounded in the accessor method `rate` depends on which class that `self` references in the expression `self.rate`. For example, if `self` references a `Car` object, then the method `computeDiscountRate` is bounded to the method `computeDiscountRate` in the class `Car`.

- The `getDriverRate` method is defined as abstract in the superclass `Vehicle` and is implemented in the class `Car`, `MiniVan` or `Truck`. Which subclass' method is bounded in the method `getDriverRate` depends on which class that `self` references in the expression `self.getDriverRate()`. The expression `vehicle.getDriverRate()` is polymorphic.

- The algorithm structure in the method `diffRate` in the class `Vehicle` is fixed but the subclasses implement variation of the methods being called.

- a method to reduce the rate of vehicles by a specified rate if the vehicles are above a certain age, rate and discount.

  The method iterates through the list, `_vehicles`, picks up a vehicle which is either a `Car` , a `MiniVan` or a `Truck` object in each iteration, and requests an update on the vehicle rate by a specific rate drop, should the vehicle qualifies (above a specified age and vehicle rate and below a certain discount rate).

```
def updateRate(self, age, vehicleRate, discountRate, rateDrop):
    for vehicle in self._vehicles:
        if vehicle.age() > age and vehicle.rate > vehicleRate \
        and vehicle.computeDiscountRate() <= discountRate:
            vehicle.rate = vehicle.rate * (1- rateDrop)
```

  Again, `vehicle.rate`, `vehicle.computeDiscountRate()` and `vehicle.rate()` are polymorphic.

Note that collection classes, when managing a collection of objects from a class hierarchy with an abstract superclass should view the objects as having the functionality defined by the (common) interface of the superclass, similar to the case when a class manages a collection of objects from a class hierarchy with a concrete

superclass. In doing so, methods can then capitalise on the template method design to allow these methods to be open to extension.

We perform unit testing on the class `Bank` to test the provided services.

Create a `RentalCompany` object to test.

```
rc = RentalCompany('SUSS Vehicle Rental')
```

Testing the methods `listAll()` and `__str__`:

```
print(rc)
```

Output

```
Rental Company Name: SUSS Vehicle Rental
No Account
```

Testing the method `add`:

```
print('Add SX1234Y:', rc.add(Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 100.0,
False)))
print('Add PC2222F:', rc.add(MiniVan("PC2222F", 'Ford', 'Transit', 3700, False,
2017, 150, True)))
print('Add SX1235Y:', rc.add(Car("SX1235Y", 'Toyota', 'Prius', 1798, 2013, 70.0,
True)))
print('Add QA7777G:', rc.add(Truck("QA7777G", 'Isuzu', 'NPR-HD', 5200, 2017, 180,
True)))
print(rc)

print('Add QA7777G again:', rc.add(Truck("QA7777G", 'Isuzu', 'NPR-HD', 5200, 2017,
300, True)))
print(rc)
```

Output

```
Rental Company Name: SUSS Vehicle Rental
1) Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 100.00   Driver Rate: $ 100.00

2) Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 135.00  Driver Rate: $ 67.50

3) Registration No: SX1235Y   Make: Toyota          Model:  Prius        On Discount: True
Engine Capacity: 1798   Year of manufacture: 2013   Rental Rate: $ 63.00   Driver Rate: $ 120.00

4) Registration No: QA7777G   Make: Isuzu           Model:  NPR-HD       On Discount: True
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 171.00   Driver Rate: $ 200.00


Add QA7777G again: False
```

```
Rental Company Name: SUSS Vehicle Rental
1) Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 100.00   Driver Rate: $ 100.00

2) Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 135.00  Driver Rate: $ 67.50

3) Registration No: SX1235Y   Make: Toyota          Model:  Prius        On Discount: True
Engine Capacity: 1798   Year of manufacture: 2013   Rental Rate: $ 63.00   Driver Rate: $ 120.00

4) Registration No: QA7777G   Make: Isuzu           Model:  NPR-HD       On Discount: True
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 171.00   Driver Rate: $ 200.00
```

Testing the method `locate`:

```
print('locate QA7777G:', rc.locate('QA7777G'))
print('locate AA1234X:', rc.locate('AA1234X'))
```

Output

```
locate QA7777G: Registration No: QA7777G   Make: Isuzu           Model:  NPR-HD       On Discount: True
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 171.00   Driver Rate: $ 200.00

locate AA1234X: None
```

Testing the method `remove`:

```
print('remove QA7777G:', rc.remove('QA7777G'))
print('remove AA1234X:', rc.remove('AA1234X'))
print(rc)
```

Output

```
remove QA7777G: True
remove AA1234X: False

Rental Company Name: SUSS Vehicle Rental
1) Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 100.00   Driver Rate: $ 100.00

2) Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 135.00  Driver Rate: $ 67.50

3) Registration No: SX1235Y   Make: Toyota          Model:  Prius        On Discount: True
Engine Capacity: 1798   Year of manufacture: 2013   Rental Rate: $ 63.00   Driver Rate: $ 120.00
```

Testing the method `higherDriverRate`:

```
vehicles = rc.higherDriverRate()
for i, v in enumerate(vehicles, start = 1):
    print(i, ')', v, 'Difference {:4.1f}'.format(abs(v.diffRates())))
```

Output

```
1) Registration No: SX1235Y   Make: Toyota          Model:  Prius        On Discount: True
Engine Capacity: 1798   Year of manufacture: 2013   Rental Rate: $ 63.00   Driver Rate: $ 120.00
 Difference 57.0
Number of vehicles = 1
```

Testing the method `updateRate`

```
rc.updateRate(1, 70, 0.10, 0.4) # age, vehicleRate , discountRate, rateDrop
print(rc)
```

Output

```
Rental Company Name: SUSS Vehicle Rental
1) Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 70.00   Driver Rate: $ 100.00

2) Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 135.00  Driver Rate: $ 67.50

3) Registration No: SX1235Y   Make: Toyota          Model:  Prius        On Discount: True
Engine Capacity: 1798   Year of manufacture: 2013   Rental Rate: $ 63.00   Driver Rate: $ 120.00
```

Once we have tested the class `RentalCompany`, we can write a similar application as those in Sections 1.1 and 1.2, which uses the classes `Car`, `MiniVan`, `Truck` and `RentalCompany` to manage the vehicle list of a rental company to

- add and remove vehicles from its fleet,
- print out details of all vehicles
- print out details of vehicles with driver rate higher than vehicle rate
- update rental rate

Studying the code of the classes in Sections 1.1 to 1.3, we note the following points:

1. Superclasses should not be aware or be concerned about the implementations of its subclasses.

2. Abstract superclasses can impose what methods their subclasses must implement.

3. Subclasses know which classes are their superclasses. If the subclass is in a different module, the superclass must be imported.

4. Classes with the collection need not be aware of which classes the objects in the collection come from. The classes with the collection, however, will assume the interface of the superclass, whether concrete or abstract.

Activity 3

Given the abstract superclass `Employee`: and subclass `PartTimeEmployee`:

```
import datetime
from abc import ABC, abstractmethod

class Employee(ABC):

    _currentId = 0

    def __init__(self, name, dateJoined = None):
        Employee._currentId += 1
        self._id = Employee._currentId
        self._name  = name
        self._dateJoined  = \
        datetime.date.today() if dateJoined is None\
        else dateJoined

    @property
    def id(self):
        return self._id

    @property
    def name(self):
        return self._name

    @property
    def dateJoined(self):
        return self._dateJoined

    @abstractmethod
    def salary(self):
        pass

    @abstractmethod
    def incrementSalary (self, percent):
        pass

    def yearsOfService(self):
        today = datetime.date.today()
        currentDateJoined = datetime.date(today.year, \
                                          self._dateJoined.month, \
                                          self._dateJoined.day )
        return (today.year - self._dateJoined.year) \
            - (1 if today < currentDateJoined else 0)

    def __str__(self):
        return ("ID: {:2d} Name: {:} "\
                + "Joined: {:%#d %b %Y} "\
                + ("Years " if self.yearsOfService() > 1 \
                   else "Year  ")\
                + "Service: {:2d} Salary: ${:7.2f} ").\
                format(self._id, self._name, self._dateJoined, \
                      self.yearsOfService(), self.salary())
```

```
class PartTimeEmployee(Employee):

    _standardRate = 20
    _baseSalary = 500

    def __init__(self, name, dateJoined = None, rate = None):
        super().__init__(name, dateJoined)
        self._hours = 0
        self._rate  = \
        PartTimeEmployee._standardRate if rate is None else rate

    def salary(self):
        factor = 1 if self._hours < 20 else 1.2
        return type(self)._baseSalary * factor + self._hours *
self._rate

    def addHours(self, hours):
        self._hours += hours

    def resetHours(self):
        self._hours = 0

    def incrementSalary (self, percent):
        self._rate *= (1 + percent/100)

    def __str__(self):
        return super().__str__() + ("Rate: {:4.1f} Hours: {:4.1f} ").\
                format(self._rate, self._hours)
```

a) Implement the class `FullTimeEmployee` that has an instance variable `_salary`.
   - Implement the method `salary()`simply as just returning the salary
   - Implement the method `incrementSalary()` which accepts a percent and adds an additional 1% per every 10 years of service completed .
   - Refine the constructor and any other method, if necessary.

b) Implement a class `Company` that has an instance variable `_employees`.
   - `locate` that searches for employee with a given employee number.
     `add` that adds an employee to the list. .There should not be duplicate employee number. The method returns True if add is successful and False otherwise.
   - `remove` that removes an employee given an employee number
     The method returns True if remove is successful and False otherwise.
   - `listAll` that return a string representation of the `employees` in the list.
   - a `getEmployeeWithMaxSalary` that return the employee with the highest salary. Return None if the list is empty.
   - an `incrementSalary` that increment the salary of every employee in

the list by a specific percent

c) Test your classes.

Chapter 2 Design Patterns for Inheritance and Object Composition

Design patterns are best practices for solving commonly occurring problems. We consider two design patterns involving inheritance and object composition.

2.1 Template Method Pattern

The template method pattern applies to inheritance, when the algorithm for a problem varies according to different subclasses.

The algorithm has

- invariant steps that are common to and independent of the subclasses, and
- variant steps implemented in the different subclasses.

The template method pattern reduces code duplication. Had each subclass implemented the algorithm for the problem, the steps that are invariant would be duplicated in the methods for each subclass. A change in the algorithm involving the invariant steps will require changes in the methods in the various subclasses that implement the algorithm.

To avoid code duplication, the invariant steps are coded in the method for the superclass and the steps that are different are coded in the different subclasses. The method for the variant steps in each subclass has a common interface.

The method `rate` in the class `Vehicle` contains the invariant steps of the algorithm.

```
class Vehicle(ABC):
    def rate(self):
        return self._rate * (1- self.computeDiscountRate() ) \
            if self._onDiscount else self._rate
```

The method `computeDiscountRate` in the various subclasses of `Vehicle` contains the variant steps of the algorithm and have the same interface: method name `computeDiscountRate` with no formal parameter and a returned number value.

```
class Car(Vehicle):
    def computeDiscountRate(self):
        age =  self.age()
        return 0.05 if age <= 2 else 0.10 if age <= 5 else 0.15
```

```
class MiniVan(Vehicle):

    def computeDiscountRate(self):
        return 0.1

class Truck(Vehicle):
    def computeDiscountRate(self):
        return 0.1 if self.capacity <= 3000 else 0.05
```

Activity4

Identify the method(s) that applies template method pattern. Identify the invariant and variant steps of the algorithm.

```
import datetime
from abc import ABC, abstractmethod

class Employee(ABC):

    _currentId = 0

    def __init__(self, name, dateJoined = None):
        Employee._currentId += 1
        self._id = Employee._currentId
        self._name  = name
        self._dateJoined  = \
        datetime.date.today() if dateJoined is None\
        else dateJoined

    @property
    def id(self):
        return self._id

    @property
    def name(self):
        return self._name

    @property
    def dateJoined(self):
        return self._dateJoined

    @abstractmethod
    def salary(self):
        pass

    @abstractmethod
    def incrementSalary (self, percent):
        pass

    def yearsOfService(self):
        today = datetime.date.today()
        currentDateJoined = datetime.date(today.year, \
                                          self._dateJoined.month, \
                                          self._dateJoined.day )
        return (today.year - self._dateJoined.year) \
            - (1 if today < currentDateJoined else 0)
```

```
            def __str__(self):
                return ("ID: {:2d} Name: {:} "\
                        + "Joined: {:%#d %b %Y} "\
                        + ("Years " if self.yearsOfService() > 1 \
                           else "Year  ")\
                        + "Service: {:2d} Salary: ${:7.2f} ").\
                        format(self._id, self._name, self._dateJoined, \
                              self.yearsOfService(), self.salary())
```

2.2 Strategy Pattern

The Strategy pattern applies to inheritance and object composition, when the algorithm (or strategy) for a problem varies according to different subclasses, similar to the template method pattern. However, the strategy pattern aims to reduce the interface of the superclass, as illustrated in the example that follows.

For example, suppose there are various strategies of calculating the amount payable for a vehicle rental and the strategies depend on the type of rental.

There are 2 ways to solve this problem:

- using inheritance

  A superclass Rental and the various subclasses of Rental that implement the various strategies of calculating the amount payable for a vehicle rental are used in the solution.

- using object composition and applying the strategy pattern

  A superclass Strategy and the various subclasses of Strategy to implement the various strategies of calculating the amount payable for a vehicle rental are used in the solution.

  The Rental class has an instance variable to reference the strategy it needs to compute the amount payable.

Using inheritance:

```
from abc import ABC, abstractmethod
class Rental(ABC):
    _rentalNum = 1

    def __init__(self, veh, startDate, driver = None, endDate = None):
        self._vehicle  = veh
        self._start = startDate
        self._end = endDate
        self._rentNo = Rental._rentalNum
```

```
        self._driver = False if driver is None else driver
        Rental._rentalNum += 1

    def endDate(self, end):
        self._end = end

    endDate = property(None, endDate)

    def getDays(self):
        diff = self._end - self._start
        return diff.days

    def computePayment(self):
        if self._end is None:
            return -1
        dailyrate = self._vehicle.rate  \
        + (self._vehicle.getDriverRate() if self._driver else 0)
        return self.getAmount(self.getDays(), dailyrate)

    @abstractmethod
    def getAmount(self, numDays, dailyRate):
        pass

class HolidayRental(Rental):
    def getAmount(self, numDays, dailyRate):
        holidaySurcharge = 0.25
        factor = 1 if numDays <= 5 else 0.9
        return factor * numDays * dailyRate * (1 + holidaySurcharge)

class PreferentialRental(Rental):
    def getAmount(self, numDays, dailyRate):
        amt = numDays * dailyRate
        if amt >= 1000:
            amt *= 0.9
        elif amt > 500:
            amt *= 0.95
        return amt

class OffPeakRental(Rental):
    def getAmount(self, numDays, dailyRate):
        offPeakDiscount = 0.25
        return numDays * dailyRate * (1 - offPeakDiscount)

class PeakRental(Rental):
    def getAmount(self, numDays, dailyRate):
        peakSurcharge = 0.5
        return numDays * dailyRate * (1 + peakSurcharge)
```

In this solution, the template method pattern is used. The method `computePayment` implements the invariant steps while the method `getAmount` in each subclass implements the variant steps.

The various strategies of calculating the amount payable for a vehicle rental implemented in method `getAmount` require just two formal parameters – the number

of days of rental and the daily rate (the sum of vehicle rate and driver rate if rental requires driver).

The various strategies can work out the amount payable without referencing he rental object itself or requiring the services it provide. Note also that each subclass of `Rental` is defined for the sole purpose of implementing the method `getAmount`.

The subclasses thus have more functionality that what they require to implement the method `getAmount`, and this violates a design principle, specifically, the Interface Segregation Principle covered in Study Unit 6. The Interface Segregation Principle states that clients should not be made to depend on methods they do not need. The Interface Segregation Principle promotes a thin interface.

The method `getAmount` is placed wrongly as a method of a subclass of Rental. However, the method `computeDiscountRate` as discussed in section 2.1, is correctly placed as a method of the various subclasses of `Vehicle`.

Implementing the method `computeDiscountRate` may require knowledge about the vehicle such as its age or capacity or any other aspect of the vehicle. Furthermore, some subclasses may define additional instance variables to compute the discount rate. Placing the method `computeDiscountRate` in the various subclasses of `Vehicle` does not make the class dependent on methods it does not need, and thus, it does not violate the Interface Segregation Principle.

The classes can be tested with this code fragment:

```
v1 = Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 100.0, False)
print(v1)
v2 = MiniVan("PC2222F", 'Ford', 'Transit', 3700, False, 2017, 100, True)
print(v2)
v3 = Truck("QA7777G", 'Isuzu', 'NPR-HD', 5200, 2017, 100, False)
print(v3)

today = datetime.date.today()
someday = datetime.date(2018, 5, 15) #year, month, day
days = (someday-today).days
print(days, 'days' if days > 1 else 'day')
r1 = HolidayRental(v1, today, True, someday)
r2 = PreferentialRental(v2, today, False, someday)
r3 = OffPeakRental(v3, today, False, someday)
r4 = PeakRental(v3, today, True, someday)
print('r1', r1.computePayment())
print('r2', r2.computePayment())
print('r3', r3.computePayment())
print('r4', r4.computePayment())
```

Output

```
Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 100.00   Driver Rate: $ 100.00

Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap
Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 90.00  Driver Rate: $ 45.00

Registration No: QA7777G   Make: Isuzu           Model:  NPR-HD       On Discount: False
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 100.00   Driver Rate: $ 200.00

1 day
r1 250.0
r2 90.0
r3 75.0
r4 450.0
```

Using object composition and applying the strategy pattern:

```
from abc import ABC, abstractmethod
class CalcRentalStrategy(ABC):
    @abstractmethod
    def getAmount(self, num, rate):
        pass

class CalcHolidayRentalStrategy(CalcRentalStrategy):
    def getAmount(self, numDays, dailyRate):
        holidaySurcharge = 0.25
        factor = 1 if numDays <= 5 else 0.9
        return factor * numDays * dailyRate * (1 + holidaySurcharge)

class CalcPreferentialRentalStrategy(CalcRentalStrategy):
    def getAmount(self, numDays, dailyRate):
        amt = numDays * dailyRate
        if amt >= 1000:
            amt *= 0.9
        elif amt > 500:
            amt *= 0.95
        return amt

class CalcOffPeakRentalStrategy(CalcRentalStrategy):
    def getAmount(self, numDays, dailyRate):
        offPeakDiscount = 0.25
        return numDays * dailyRate * (1 - offPeakDiscount)

class CalcPeakRentalStrategy(CalcRentalStrategy):
    def getAmount(self, numDays, dailyRate):
        peakSurcharge = 0.5

        return numDays * dailyRate * (1 + peakSurcharge)


class Rental(object):
    _rentalNum = 1

    def __init__(self, veh, startDate, driver = None, calcRentalStrategy = None,
endDate = None):
        self._vehicle  = veh
```

```
        self._start = startDate
        self._end = endDate
        self._driver = False if driver is None else driver
        self._calcRentalStrategy= calcRentalStrategy
        self._rentNo = Rental._rentalNum
        Rental._rentalNum += 1

    def endDate(self, end):
        self._end = end

    endDate = property(None, endDate)

    def getDays(self):
        diff = self._end - self._start
        return diff.days

    def computePayment(self):
        if self._end is None:
            return -1
        dailyrate = self._vehicle.rate  \
        + (self._vehicle.getDriverRate() if self._driver else 0)
        if self._calcRentalStrategy is None:
            return self.getDays() * dailyrate
        return self._calcRentalStrategy.getAmount(self.getDays(), dailyrate)
```

Notice that the interface of the superclass in the inheritance hierarchy is reduced to only those required by the subclasses, namely, the method `getAmount`. In addition, notice that we applied object composition to the class `Rental`, giving it an additional attribute `_calcRentalStrategy`.

The classes can be tested with this code fragment:

```
v1 = Car("SX1234Y", 'Toyota', 'Prius', 1798, 2016, 100.0, False)
print(v1)
v2 = MiniVan("PC2222F", 'Ford', 'Transit', 3700, False, 2017, 100, True)
print(v2)
v3 = Truck("QA7777G", 'Isuzu', 'NPR-HD', 5200, 2017, 100, False)
print(v3)

today = datetime.date.today()
someday = datetime.date(2018, 5, 15) #year, month, day
days = (someday-today).days
print(days, 'days' if days > 1 else 'day')
r1 = Rental(v1, today,  True, CalcHolidayRentalStrategy(), someday)
r2 = Rental(v2, today, False,  CalcPreferentialRentalStrategy(), someday)
r3 = Rental(v3, today, False,  CalcOffPeakRentalStrategy(), someday)
r4 = Rental(v3, today, True, CalcPeakRentalStrategy(), someday)
print('r1', r1.computePayment())
print('r2', r2.computePayment())
print('r3', r3.computePayment())
print('r4', r4.computePayment())
```

Output

```
Registration No: SX1234Y   Make: Toyota          Model:  Prius        On Discount: False
Engine Capacity: 1798   Year of manufacture: 2016   Rental Rate: $ 100.00   Driver Rate: $ 100.00

Registration No: PC2222F   Make: Ford            Model:  Transit     On Discount: False  Handicap
Accessible: True
Engine Capacity: 3700  Year of manufacture: 2017  Rental Rate: $ 90.00  Driver Rate: $ 45.00

Registration No: QA7777G   Make: Isuzu           Model:  NPR-HD       On Discount: False
Engine Capacity: 5200   Year of manufacture: 2017   Rental Rate: $ 100.00   Driver Rate: $ 200.00

1 day
r1 250.0
r2 90.0
r3 75.0
r4 450.0
```

Activity 5

Differentiate the purpose of the template method pattern and strategy pattern..

Summary

An attribute of an object can be a collection. A class with a collection must provide services to allow clients to manage the collection. Standard collection services include locating an object in a collection by an identifying attribute, adding an object to the collection, removing an object from the collection and listing the objects in the collection.

Collection methods such as locate and list can be compared to the accessor method that allow instance variables to be inspected. Collection methods such as add and remove can be compared to the mutator method that allow instance variables to be updated

The structure of the corresponding methods for collection services is the same, regardless whether the objects objects in the collection are objects from a single class, objects from a concrete superclass and its subclasses or objects from subclasses of an abstract superclass. However, the classes of the objects must have the same interface, therefore, it is usually the interface of the superclass.

Design patterns are best practices for solving commonly occurring problems. Two design patterns involving inheritance and object composition are template method and strategy.

The template method pattern is used to implement an algorithm that has variant and invariant steps. The variant steps are implemented in a method of a subclass as the implementation is dependent on the interface of superclass. The template method help reduces code duplication.

The Strategy pattern introduces a new class hierarchy to reduce the interface to those needed by the variant steps. The new class hierarchy implements the variant steps or strategies, and each subclass implements one strategy. Object composition is used to associate a strategy to an object.

References

The Python Tutorial, https://docs.python.org/3/tutorial/

Formative Assessment

1. Which service is not a standard service for collection management?

A) removing an object from the collection
@ Incorrect. This is a standard service for collection management. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

B) adding an object to the collection
@ Incorrect. This is a standard service for collection management. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

*C) making a copy of the collection
@ Correct. This is not a standard service for collection management. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

D) locating an object in the collection
@ Incorrect. This is a standard service for collection management. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

2. In which method should the collection be created?

A) accessor method
@ Incorrect. The accessor method inspects the collection. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

B) mutator method
@ Incorrect. The mutator method updates the collection. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

C) the method `__str__`
@ Incorrect. The method `__str__` returns a string representation of the object with the collection. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

*D) the method `__init__`
@ Correct. The constructor will create an empty collection. Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

3. What is the purpose of an identifier in collection management?

A) It identifies the object that has the collection
@ Incorrect. This is not the purpose of an identifier in collection management Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

B) It identifies the collection

@ Incorrect. This is not the purpose of an identifier in collection management Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

*C) It identifies the object in the collection.
@ Correct. This is the purpose of an identifier in collection management Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3.

D) It identifies the collection method
@ Incorrect. This is not the purpose of an identifier in collection management Refer to Section 1.1 Managing a Collection of Objects from a Single Class, Study Unit3

4. Which statement correctly describes the methods for collections where objects are from single class and for collections where objects are from a concreate superclass and its subclasses?

A) The structure of the `for` loop in the method and the structure of the method are different.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

*B) The structure of the `for` loop in the method and the structure of the method are the same.
@ Correct. The statement correctly describes the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

C) The structure of the `for` loop in the method is the same but the structure of the method is different.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

D) The structure of the `for` loop in the method is different but the structure of the method is the same.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

5. Which statement correctly describes the methods for collections where objects are from a concreate superclass and its subclasses and for collections where objects are from the subclasses of an abstract superclass?

A) The structure of the `for` loop in the method and the structure of the method are different.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

*B) The structure of the `for` loop in the method and the structure of the method are the same.
@ Correct. The statement correctly describes the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

C) The structure of the `for` loop in the method is the same but the structure of the method is different.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

D) The structure of the `for` loop in the method is different but the structure of the method is the same.
@ Incorrect. The statement does not correctly describe the methods. Refer to 1.2 Managing a Collection of Objects from Subclasses and a Concrete Superclass, Study Unit 3.

6. Which interface should a collection method use when requesting services from the objects in the collection, and the objects are from subclasses of an abstract superclass?

A) The interface of the class that the object belongs to
@ Incorrect. This is an incorrect interface. Refer to Section 1.3 Managing a Collection of Objects from Subclasses of an Abstract Superclass, Study Unit 3.

*B) The interface of the abstract superclass
@ Correct. This is the correct interface. Refer to Section 1.3 Managing a Collection of Objects from Subclasses of an Abstract Superclass, Study Unit 3. .

C) The interface of the class that has the collection.
@ Incorrect. This is an incorrect interface. Refer to Section 1.3 Managing a Collection of Objects from Subclasses of an Abstract Superclass, Study Unit 3.

D) The interface of the class for the collection.
@ Incorrect. This is an incorrect interface. Refer to Section 1.3 Managing a Collection of Objects from Subclasses of an Abstract Superclass, Study Unit 3.

7. What is a purpose of the strategy design pattern?

A) It allows subclasses to have more functionality than they need to implement the strategy.
@ Incorrect. This is not a purpose of the strategy design pattern. Refer to Section 2.2 Strategy Pattern, Study Unit 3.

*B) It allows subclasses to have just enough functionality that they need to implement the strategy.
@ Correct. This is a purpose of the strategy design pattern. Refer to Section 2.2 Strategy Pattern, Study Unit 3.

C) It allows superclasses to have more functionality than they need to implement the strategy.
@ Incorrect. This is not a purpose of the strategy design pattern. Refer to Section 2.2 Strategy Pattern, Study Unit 3.

D) It allows superclasses to have just enough functionality that they need to implement the strategy.
@ Incorrect. This is not a purpose of the strategy design pattern. Refer to Section 2.2 Strategy Pattern, Study Unit 3.

View the answers at the end of this study unit.

Solutions or Suggested Answers

Activity 1

```
class Employee:
    _defSal = 2000
    _maxSalary = 15000
    _minSalary = 1200
    _defPerf = 1
    _perfFactor = {5:2, 4:1.2, 3:0.5}
    _nextNumber = 1

    @classmethod
    def addPerfFactor(cls, performanceLevel, factor):
        cls._perfFactor[performanceLevel] = factor

    @classmethod
    def removePerfFactor(cls, performanceLevel):
        return cls._perfFactor.pop(performanceLevel, 0)

    def __init__(self, n, salary = None, performance = None, supervisor= None):
        self._name = n
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self._performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
        self._claims = []

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def empNum(self):
        return self._empNum

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, value):
        self._salary = \
        Employee._minSalary if  value < Employee._minSalary \
        else Employee._maxSalary if value > Employee._maxSalary \
        else value

    def performance(self, value):
        self._performance = value
```

```
    performance = property(None, performance)

    def calculateBonus(self):
        return  Employee._perfFactor.get(self._performance , 0)  * self._salary

    def increaseSalaryBy(self, percent = 0, base = 20):
        self.salary = self._salary * (1 + percent) + base

    @property
    def supervisor(self):
        return self._supervisor

    @supervisor.setter
    def supervisor(self, supervisor):
        self._supervisor = supervisor

    def locate(self, dateClaimed):
        found = []
        for c in self._claims:
            if c._dateClaimed == dateClaimed:
                found.append(c)
        return found

    def add(self, c):
        self._claims.append(c)


    def remove(self, amount, dateClaimed):
        for c in self._claims:
            if c.amount == amount and c._dateClaimed == dateClaimed:
                self._claims.remove(c)
                return True
        return False

    def listAll(self):
        s = ''
        for i, c in enumerate(self._claims, start = 1):
            s += ("{:>3s}").format(str(i)) + ') ' + str(c) +  '\n'
        return s or 'No claim\n'


    def __str__(self):
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
                + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} " \
                + "Supervisor: {:} \n{:}") \
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor._name, self.listAll())

import datetime
from SU3.Claim import Claim

e1 = Employee("Sam", 2500)
print(e1)

e1.add(Claim(50, datetime.date(2017, 12, 15)))
e1.add(Claim(50, datetime.date(2017, 12, 16)))
e1.add(Claim(50, datetime.date(2017, 12, 15)))
```

```
print(e1)

claims = e1.locate(datetime.date(2017, 12, 15))
for i, c in enumerate(claims, start = 1):
    print(("{:>3s}").format(str(i)) + ') ' + str(c))

e1.remove(50, datetime.date(2017, 12, 15))
print(e1)
```

Activity 2

d. The constructor for the `Employee` class – no change

e. The methods in the class `Employee`:

- `locate` – no change
- `add` – no change
- `remove` – no change
- `listAll` – no change
- `__str__` – no change

f. Write code to test the `Employee` class. Create various types of `Claim` objects.

```
import datetime
from SU3.Claim import Claim, MedicalClaim

e1 = Employee("Sam", 2500)
print(e1)

print('Add claims')
e1.add(Claim(50, datetime.date(2017, 12, 15)))
e1.add(MedicalClaim(50, datetime.date(2017, 12, 15), '123/12/8',
'HealthCare Pte Ltd'))
e1.add(Claim(50, datetime.date(2017, 12, 16)))
e1.add(Claim(50, datetime.date(2017, 12, 15)))
print(e1)

claims = e1.locate(datetime.date(2017, 12, 15))
print('Locate claim')
for i, c in enumerate(claims, start = 1):
    print(("{:>3s}").format(str(i)) + ') ' + str(c))
print()

print('Remove claim', \
      e1.remove(50, datetime.date(2017, 12, 15)), '\n', e1)
print('Remove claim', \
      e1.remove(50, datetime.date(2017, 12, 15)), '\n', e1)
print('Remove claim', \
      e1.remove(50, datetime.date(2017, 12, 15)), '\n', e1)
print('Remove claim', \
      e1.remove(50, datetime.date(2017, 12, 15)), '\n', e1)
```

Activity 3

a)

```
class FullTimeEmployee(Employee):
    def __init__(self, name, salary, dateJoined = None):
        super().__init__(name, dateJoined)
        self._salary = salary

    def salary(self):
        return self._salary

    def incrementSalary (self, percent):
        self._salary *= (1 + (percent + self.yearsOfService() // 10) /100)
```

b)

```
class Company (object):
    def __init__(self):
        self._employees = []

    def locate(self, ID):
        for e in self._employees:
            if e.id == ID:
                return e
        return None

    def add(self, e):
        if self.locate(e.id) is None:
            self._employees.append(e)
            return True
        else:
            return False

    def remove(self, ID):
        s = self.locate(ID)
        if s is None:
            return False
        self._employees.remove(s)
        return True

    def listAll(self):
        s = ''
        for i, emp in enumerate(self._employees, start = 1):
            s += str(i) + ') ' + str(emp) +  '\n'
        return s or 'No employee\n'

    def getEmployeeWithMaxSalary (self):
        if len(self._employees) == 0:
            return None
        maxPos = 0
        maxSalary = self._employees[0].salary()
        for i, e in enumerate(self._employees):
            print('enumerate', i, e)
            if e.salary() > maxSalary:
                maxSalary = e.salary()
                maxPos = i
```

```
        return self._employees[maxPos]

    def incrementSalary (self, percent):
        for e in self._employees:
            e.incrementSalary(percent)

    def __str__(self):
        return self.listAll()
```

c)

```
com = Company()
print('max = ', com.getEmployeeWithMaxSalary())
p1 = PartTimeEmployee('Jane', datetime.date(2017, 12, 15))
p1.addHours(20)
f1 = FullTimeEmployee('Peter', 2000, datetime.date(2007, 12, 15) )
p2 = PartTimeEmployee('Judy', datetime.date(2017, 12, 15))
p2.addHours(40)
com.add(p1)
com.add(f1)
com.add(p2)
print(com)
print('max = ', com.getEmployeeWithMaxSalary())
print('remove 1', com.remove(1))
print('remove 4', com.remove(4))
print(com)
com.incrementSalary(10)
print(com)
```

Activity 4

```
    def __str__(self):
        return ("ID: {:2d} Name: {:} "\
                + "Joined: {:%#d %b %Y} "\
                + ("Years " if self.yearsOfService() > 1 \
                   else "Year  ")\
                + "Service: {:2d} Salary: ${:7.2f} ").\
                format(self._id, self._name, self._dateJoined, \
                      self.yearsOfService(), \  # invariant part till here
                      self.salary() )           # variant part
```

Activity 5

Template method pattern

- Implement an algorithm with invariant and variant steps
- The variant steps are implemented in a method of a subclass as the implementation is dependent on the interface of superclass.
- Uses inheritance.

Strategy pattern

- Uses a new class hierarchy for strategies
- The class hierarchy reduces the interface to those needed by the variant steps
- Implement object composition for strategy object

STUDY UNIT 4
EXCEPTION HANDLING

Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the source of exception
2. Describe the purpose of exception handling
3. Outline the implementation of exception handling and explain the try-except-else-finally execution flow
4. Implement exception handling
5. Explain the purpose of custom Exception classes and implement a custom Exception class
6. Explain the keyword raise and demonstrate the usage

Overview

This study unit focuses on handling runtime errors which occur whenever the runtime system or the interpreter is unable to successfully execute program statements. A runtime error is also called an exception. There are two ways of dealing with exception: preventive or defensive programming and recovery or handling exception after it has been raised. The latter is the focus of this study unit.

To recover from an exception after it has been raised requires special language constructs to route the program execution to statements that handle the exception. In particular, the study unit elaborates on the `try-except-else-finally` construct and the `with` statement provided in the Python language.

The study unit also demonstrates how to implement a user-defined exception class. Having user-defined exception classes allows an application to distinguish between exceptions raised from business rule violation and from semantically invalid program statements.

When a program detects that a business rule is violated, it can raise exceptions. Thus study unit covers the four ways that a program can raise exceptions from user-defined exception classes. The study guide also discusses the various scenarios of handling the exception in an application.

This study unit covers chapter 5, section 5.5 of the textbook. As there are also several online resources to read, it is estimated that the student will spend about 7 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 Programming with Exception Handling

1.1 Introduction to Exception Handling

Recall in ICT133 Study Unit 4, that when a program is being tested, three types of errors may result:

- Syntax error

  A syntax error occurs when the compiler or the interpreter is not able to break the various parts of the program statements into elements of the language, according to its grammar rules.

  For example, consider this code fragment:

```
a = 1
b = a--
```

  The statement

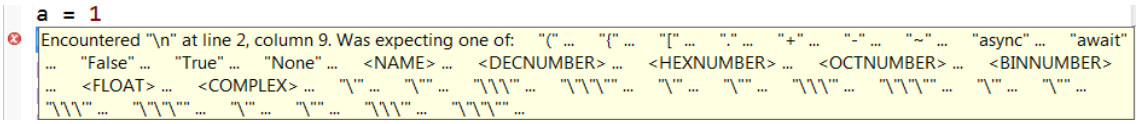
```
b = a--
```

  is invalid as `--` is not a valid operator in Python, although it is a valid operator in other languages such as C and Java to mean the decrement operator.

  As `--` is not a valid operator in Python, `a--` does not follow the grammar rule of Python.

  A program with syntax errors cannot be executed; the syntax errors must be resolved first.

  In Eclipse, Python highlights the syntax error and explains the reason:



  Syntax errors can be resolved by rewriting the erroneous program statements so that they obey the grammar rules of the programming language.

  The statement

```
b = a--
```

  must be rewritten as

```
b = a - 1
```

  to follow Python grammar rules.

- Runtime error

  A runtime error occurs when the runtime system or the interpreter is not able to successfully execute a program statement because the operation specified in the statement is invalid, given the current state or the values of variables in the program.

  Performing an invalid operation is the source of runtime error. When a runtime error occurs, we say **an exception has been raised**.

  Exceptions cause a program to halt or to terminate abnormally.

  For example, this code fragment does not contain syntax errors:

```
a = 1
b = a - 1
print('Answer', end = ' ')
print( a/b)
print('program end')
```

  but executing it causes an exception to be raised as performing `a/b` is invalid when the value of `b` is `0`. Division by `0` cannot be performed correctly and so the program halts.

```
<terminated> syntaxErr.py [C:\Program Files\Python36\python.exe]
Traceback (most recent call last):
Answer   File "C:\Users\Jenny\Desktop\python\
    print( a/b)
ZeroDivisionError: division by zero
```

  Exceptions can be handled in 2 ways:

  - Defensive programming

    Some exceptions can be prevented by applying defensive programming such as by performing value checks whenever possible, before the program invokes an operation that can potentially raise exception.

    For example, the code fragment shows how value check can preempt an exception from being raised:

```
a = 1
b = a - 1
print('Answer', end = ' ')
if b != 0: # value check to pre-empt exception divide by zero
    print( a/b)
else:
    print('error as b is 0')
print('program end')
```

```
<terminated> syntaxErr.py [C:\Program Fil
Answer error as b is 0
program end
```

- Exception handling

  Exception handling is handling a runtime error when it occurs or when an exception is raised and is the focus of this study unit.

  Exception handling is applied when performing value checks is not straightforward or when it is not possible to perform value checks exhaustively. In such cases, it is simpler for a program to apply exception handling to handle the runtime error when it occurs.

  When an exception is raised, instead of halting, a program uses the constructs of the language to enter into an exception-handling routine to respond to or to recover from the raised exception. Thus, a program can use except handling as a means of controlling the flow of program execution.

  Exception handling uses the construct `try-except-else-finally`, covered in the next section.

- Logic error

  Logic error occurs when a program does not meet the specification of the problem such as giving out incorrect output. Logic errors are caused either by an incorrect algorithm (incorrect design) or an incorrect translation (incorrect mapping) of the algorithm into program statements.

  Logic error can be detected by running the program using a number of test cases which represent the set of possible values that a program gets as input when the program is deployed. When the program output differs from the expected results, we say the program has a logic error.

  Logic errors are resolved through debugging sessions, in which the flow of execution is traced, erroneous transformations of input to intermediate or to output values are detected, and the program steps and statements are corrected.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 179-180.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Read

https://docs.python.org/3/tutorial/errors.html

Read sections 8.1 and 8.2

Activity 1

Categorise the error that the code fragments produce.
Resolve the errors:

a)
```
print(a=1)
```

b)
```
married = input('Married? y or n: ')
if married != 'y' or != 'n':
    pass
```

c)
```
print(a)
```

d)
```
gender = input('Gender? m or f: ')
if gender != 'f' or gender != 'F':
    print('Mr.')
else:
    print('Ms.')
```

e)
```
b = input('Enter a number: ')
print('Answer is ', b**2)
```

f)
```
a = float(input('Enter a number: '))
b = float(input('Enter a number: '))
if a < b:
    bigger = a
else:
    bigger = b
print('The bigger number is ', bigger)
```

1.2 try-except-else-finally construct

Exception handling applies the `try-except-else-finally` construct with syntax as shown:

```
try:
    <block of statements that potentially can raise exception>
except (SomeExceptionClass1, …,  SomeExceptionClassi) as e:
    <block of statements to respond to
    SomeExceptionClass1, …,  SomeExceptionClassi exception object>
    …
except (SomeExceptionClassj, …,  SomeExceptionClassk) as e:
```

```
    <block of statements to respond to
    SomeExceptionClass_j, …,  SomeExceptionClass_k exception object>
else:
    <block of statements to execute if no exception is raised or
    When the try block is not exited>
finally:
    <block of statements to execute whether or not an exception is raised>
```

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 180-182.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

- `try` block

  Statements that can potentially raise exceptions are placed into the `try` block.

  If any exception is raised, instead of halting, the program continues execution in one of the `except` blocks.

  For example, `print(a/b)` raises an exception when

  - `b` is `0` or
  - `a` or `b` are not numbers.

  Therefore, we place `print(a/b)` into the `try` block:

  ```
  try:
      print(a/b)
  ```

- `except` block

  Statements that respond to or recover from a raised exception, are placed into the `except` block.

  There can be one or more `except` blocks, each block responding to various types or classes of exception objects.

  There are three forms of `except` block:

  - ```
    except SomeExceptionClass as e:
        <block of statements which responds to SomeExceptionClass
    exception object>
    ```

    where `as e` is optional.
    ```

Statements that respond to an exception object from the specified exception class, `SomeExceptionClass`, are placed into the `except` block.

The statements in this `except` block are executed only when an exception from the specified class `SomeExceptionClass` or from a subclass of `SomeExceptionClass`, is raised.

For example, this `except` block responds to a raised exception from the class `Exception` or from any of the subclasses of `Exception`:

```
except Exception as e:
    print(e)
```

- `except (SomeExceptionClass`$_1$`, …, SomeExceptionClass`$_i$`) as e:`

```
    <block of statements which responds to
   SomeExceptionClass1, …,  SomeExceptionClassi exception object>
```

where `as e` is optional as well.

Statements that respond to an exception object from any of the specified exception classes, `SomeExceptionClass`$_1$`, …, SomeExceptionClass`$_i$, are placed into the `except` block.

The statements in this `except` block are executed only when an exception from the specified classes `SomeExceptionClass`$_1$`, …, SomeExceptionClass`$_i$ or from their subclasses, is raised.

For example, this `except` block responds to exceptions from the classes `NameError`, `TypeError` and `ZeroDivisionError` and from their subclasses:

```
except (NameError, TypeError , ZeroDivisionError) as e:
    print(e)
```

- `except:`

```
    <block of statements to respond to any exception>
```

Statements that respond to any exception, are placed into the `except` block.

The statements in this `except` block are executed only when an exception occur.

For example, this `except` block responds to any exceptions from any exception classes:

```
except:
    print('An error has occurred')
```

There are three scenarios when an exception is raised:

- When one `except` block can respond

  When an exception is raised, the `except` block that can respond to the class of the exception object will execute.

- When more than one `except` blocks can respond

  When an exception is raised, *only the first* `except` block that can respond to the class of the exception object will execute. *No other* `except` block *will execute*, even if the other `except` block can also respond to the class of the exception object.

  Thus, it is important to place `except` blocks for more specific classes before `except` blocks for more general classes. It follows that `except` blocks for subclasses should be placed before the `except` blocks for superclasses.

  In this example code fragment:

```
except Exception as e:
    except Exception print('All here', type(e), e)
except ZeroDivisionError as e:
    print('divide error', e)
```

  the second `except` block for `ZeroDivisionError` never gets executed because Exception is a superclass of `ArithmeticError` which is a superclass of `ZeroDivisionError`.

  When a `ZeroDivisionError` exception is raised, the first `try` block that can respond is the block `except Exception`, as `ZeroDivisionError` is an indirect subclass of `Exception`.

- When no `except` block can respond

  In the case *when none* of the `except` blocks can respond to the exception object, the program will terminate abnormally

  - after executing statements in the `finally` block if there is a `finally` block.
  - immediately if there is no `finally` block.

  Refer to the discussion on the `finally` block.

Read

https://docs.python.org/3/library/exceptions.html#bltin-exceptions

Refer to section 5.4 for the class hierarchy for exceptions (or runtime errors), and read section 5.2 for a description of the exceptions.

- `else` block.

  The `else` block is optional.

  Statements that should execute only if executing statements in the `try` block does not raise any exception, are placed into the `else` block.

  For example, this `else` block executes `print('Success!')` if there is no exception raised when statements in the `try` block are executed:

  ```
  else:
      print('Success!')
  ```

  The `else` block can be thought of as a continuation of the `try` block. If the `try` block succeeds, the execution continues at the `else` block. If the `try` block is exited, the `else` block will not execute.

  Statement in the `else` block can be placed into the `try` block, and the `else` block is removed, as shown here:

  ```
  try:
      print(a/b)
      print('Success!')
  ```

  However, having an `else` block can help improve the readability of a program.

- `finally` block.

  The `finally` block is optional.

  Statements that should execute regardless whether executing statements in the `try` block raises exception, are placed into the `finally` block.

  For example, this `finally` block executes `print('Application is closing')` regardless whether or not executing statements in the `try` block raises an exception:

  ```
  finally:
      print('Application is closing')
  ```

A program may enter into the `finally` block, if there is a `finally` block, under one of these three circumstances:

- the `try` block does not raise an exception

  In this case, after the `try` block completes execution.

  The `else` block continues, if there is an `else` block.

  The `finally` block next executes.

  After the `finally` block has executed, the program continues to execute statements outside the `try-except-else-finally` construct.

  Figure 4.1 shows the flow of execution when there is no exception raised.

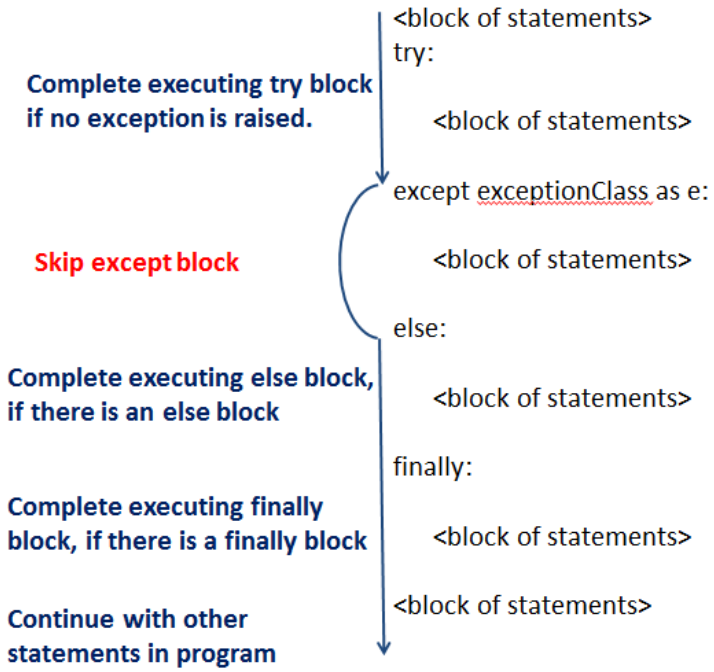


Figure 4.1 Flow of execution when no exception is raised

- the `try` block raises an exception and an `except` block can respond to the exception

  In this case, the `try` block is abandoned at the statement where the exception is raised.

  The first `except` block that can respond to the exception is entered.

After the `except` block completes execution, the `finally` block is entered and its statements executed.

Figure 4.2 shows the flow of execution when there is a raised exception and one `except` block can respond to the exception.

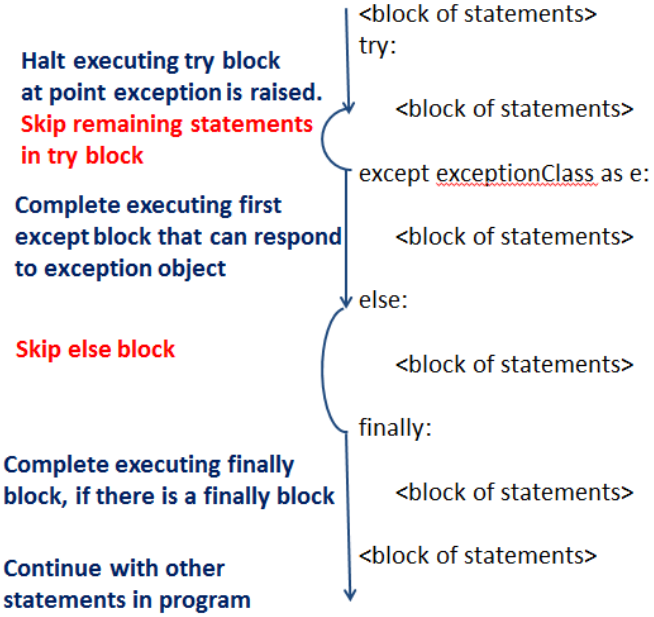


Figure 4.2 Flow of execution when there is a raised exception, and one except block can respond to the exception.

- the `try` block raises an exception and no `except` block can respond to the exception

  As in the previous case, the `try` block is abandoned at the statement where the exception is raised.

  The `finally` block is entered and its statements are executed.

  The program then terminates abnormally with a runtime error message.

  Figure 4.3 shows the flow of execution when there is a raised exception and no `except` block can respond to the exception.

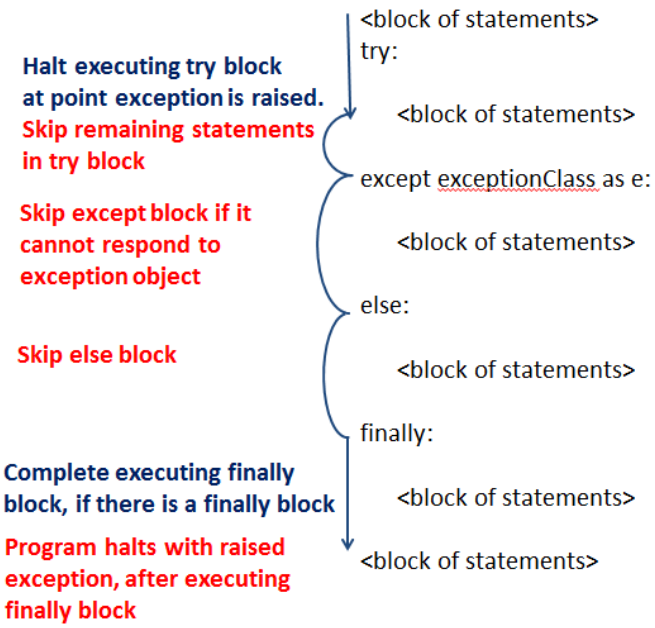


Figure 4.3 Flow of execution when there is a raised exception, and no except block can respond to the exception.

Similar to other constructs such as decision and loop structures, the `try-except-else-finally` construct can be *nested* and so it can appear within a `try`, an `except`, an `else` or a `finally` block.

Read

https://docs.python.org/3/tutorial/errors.html

Read sections 8.3 and 8.6 on handling exception.

Besides the `try-except-else-finally` construct, Python also provides a `with` statement that uses methods defined in a context manager to perform clean-up action

The context manager provides two methods:

- `__enter__`

  This method is invoked when the `with` statement is executed with some resource object attached to the `with` statement.

  For example;

```
with open('in.txt', 'r') as inp:
```

In this case, a context manager for files will open the file *in.txt* using its method `__enter__`.

- `__exit__`

  This method of the context manager is invoked when execution reaches the end of the `with` statement.

  For example;

```
with open('in.txt', 'r') as inp:
    for line in inp.readlines():
        print(line)
```

  When the last line of the file is read and printed, the context manager will perform clean-up actions such as to close the file *in.txt* using its method `__exit__`.

The `with` statement is like a `try-finally` construct without the `except` and `else` blocks. Thus, we often place `with` statements within the `try-except-else-finally` construct to handle exceptions.

For example;

```
try:
    with open('in.txt', 'r') as inp:
        for line in inp.readlines():
            print(line)
except Exception as e:
    print(e)
```

Read

https://docs.python.org/3/tutorial/errors.html

Read section 8.6 on predefined clean-up actions.

https://docs.python.org/3.3/reference/compound_stmts.html#with

Read section 8.5 on with statement.

Activity 2

For those code fragments in Activity 1 that raise exception, apply the try-except construct with exception classes specified, to print the runtime error message.

Activity 3

Correct this program such that when user enters <Enter> at the start of the run, the program produces this correct output:

```
Enter how many numbers to average or <Enter> to end:
Program ended
```

instead of this incorrect output:

```
Enter how many numbers to average or <Enter> to end:
Iteration 1
Program ended
```

```
iterNum = 0
while True:
    try:
        numstr = input('Enter how many numbers to average or <Enter>
to end: ')
        if numstr == '':
            break
        total = 0
        num = int(numstr)
        for i in range(num):
            total += float(input('Enter number: '))
        average = total/num
    except Exception as e:
        print("An error has occurred:" , e)
    else:
        print('Average is ', average)
    finally:
        iterNum += 1
        print('Iteration', iterNum)
print('Program ended')
```

Activity 4

Modify the corrected program in Activity 3 by applying nested exception handling to perform the following actions:

- differentiate those errors due to input and those due to division by zero and give the appropriate error messagesfor them using the print statements:
  - `print("An input error has occurred:" , e)`
  - `print("An computation error has occurred:" , e)`
- if the user has n numbers to enter in a particular iteration, allow the user to keep entering input until he has completed entering n valid numbers.

Activity 5

Currently, running the code fragment:

```
s =  SavingsAccount()
print(s)
```

produces a runtime error.

Therefore, modify the method `assignBalance` in the class SavingsAccount to handle the runtime error by assigning the default value defBalance to balance if none is supplied.

```
class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1

    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        self.assignBalance(balance)


    def assignBalance(self, balance):
        self._balance = float(balance)

    def __str__(self):
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumber, self._balance)
```

Test your modification so that executing the code fragment again:

```
s =  SavingsAccount()
print(s)
```

produce this output:

```
Account No:   1 Balance: $  500.00
```

Activity 6

Modify the method `assignBalance` in the class SavingsAccount in Activity 5 such that balance is assigned the default value _defBalanmce when

- the value supplied to the formal parameter balance is None or
- the value supplied to the formal parameter balance is not a number .

Chapter 2 Defining our own Exception

2.1 Creating Exception Classes

A raised exception is an object from a built-in exception class found within the class hierarchy for exceptions, given in section 5.4 of https://docs.python.org/3/library/exceptions.html#bltin-exceptions.

This class hierarchy for exceptions can be extended when programmers define new exception classes or user-defined exception classes.

A user-defined exception class allows an application to distinguish between exceptions raised because

- business rules are violated

  An example of a business rule violation is attempting to withdraw from an account with insufficient fund.

- program statements are semantically invalid

  Semantically invalid program statements are statements that attempt to operate on invalid data or state.

  An example of a semantically invalid statement is to read from a file that has been closed.

A user-defined exception class is defined in same way as any other user-defined class is defined, using the keyword `class`.

The following recommendations for user-defined exception are taken from the Python documentation:

- End the name of the user-defined exception class with the suffix `Error`, which is consistent with the names of standard built-in exception classes.

- A user-defined exception class must either directly or indirectly inherit from the class `Exception`.

- Include a docstring to the class definition for a user-defined exception class.

- Keep the class definition simple even though a user-defined exception class can have attributes and methods.

A user defined exception class can simply be defined with a class header and a body that includes only a docstring.

For example,

```
class AccountError(Exception):
    '''Class for exception for the account application'''
```

The class **`AccountError`** inherits methods such as `__init__` and `__str__` from its superclass `Exception`. There is no need to override these or any other methods.

Sometime, rather than simply defining one user-defined exception class, a whole hierarchy of user-defined exception classes may be defined. This allows an application to define several `except` blocks to cater to the exceptions raised from different user-defined exception classes.

For example,

```
class AccountError(Exception):
    '''Base class for exception for the account application'''

class InsufficentFundError(AccountError):
    '''Raised when account balance is insufficient'''

class AccountDetailsError(AccountError):
    '''Raised when account details is invalid'''
```

Having this hierarchy of user-defined exception classes allow us to have various `except` blocks to handle a raised exception differently depending on its class, such as:

```
except InsufficentFundError as e:
    <some statements>
except AccountDetailsError as e:
    <some statements>
except AccountError as e:
    <some statements>
```

However, if exceptions raised from different user-defined exception classes are handled similarly, then it is not necessary or useful to define different user-defined exception classes. The rule is to keep the implementation of the application simple and do not implement something that is not needed.

Read

https://docs.python.org/3/tutorial/errors.html

Read section 8.5 on user-defined exception.

Activity 7

Define the exception classes and provide corresponding skeletal `except` blocks to handle raised exceptions for each scenario:

a) A library member must not have fines before borrowing books. A reservation for a book can be made of the library member does not have a fine. Both errors are handled in a same manner.

b) A student enrol in a program must be at least 35 years old and a citizen. He can register for not more than 5 modules if he has paid the program fee. Errors due to disqualification, to module registration for non-payment and to module registration for exceeding the number of modules registered are handled differently.

c) An order must record a delivery address and delivery date. A delivery delay must record a reason for the delay. All errors due to missing input are handled in a same manner.

2.2 Raising Exceptions

When business rules are violated, a program can raise exceptions from user-defined exception classes, and use except handling as a means of controlling the flow of program execution.

To raise exceptions, the `raise` statement is used. The syntax of the `raise` statement is as follows:

```
raise <expression  <from  expression>>
```

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 182-183.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

We will use the class `SavingsAccount` in Activity 6 to illustrate four ways to raise an exception.

```
class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1

    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        self.assignBalance(balance)


    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except (TypeError, ValueError):
            self._balance = SavingsAccount._defBalance

    def __str__(self):
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumber, self._balance)

s =  SavingsAccount('a)
print(s)
```

- `raise`

  `raise` re-raise the last exception within the scope.

  Suppose we wish to re-raise the last exception if it is raised by creating a `SavingsAccount` object with a non-numeric balance.

```
    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except (TypeError, ValueError) as e:
            self._balance = SavingsAccount._defBalance
            if type(e) == ValueError:
                raise
```

  Some points of interest when adding a `raise` statement in the method `assignBalance`:

- Ensure that a `raise` statement is placed where there is a previous exception within the scope.

  If there is no exception within the scope, executing the `raise` statement raises a `RuntimeError` exception.

  For example, if we simply include a `raise` statement at the beginning of the `try` block and test the code with this code fragment:

```
s =  SavingsAccount('1000')
print(s)
```

```
<terminated> Secrion2RaiseExample.py [C:\Program Files\Python36\python.exe]
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeE
    s =  SavingsAccount('a')
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeE
    self.assignBalance(balance)
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeE
    raise
RuntimeError: No active exception to reraise
```

- Calling the method `assignBalance` may cause an exception to be raised. A caller of the method `assignBalance` should apply exception-handling or risk having the program terminating abnormally.

```
def __init__(self, balance = None):
    self._accountNumber = SavingsAccount._nextAccountNumber
    SavingsAccount._nextAccountNumber += 1
    try:
        self.assignBalance(balance)
    except ValueError as e:
        print('default value has been assigned as', e)
```

- `raise SomeExceptionClass`

  This `raise` statement raises an exception object from a specific exception class with an empty message.

  For example,

```
def assignBalance(self, balance):
    try:
        self._balance = float(balance)
    except (TypeError, ValueError) as e:
        self._balance = SavingsAccount._defBalance
        if type(e) == ValueError:
            raise ValueError
```

```
<terminated> Secrion2RaiseExample.py [C:\Program Files\Pyth
default value has been assigned as
Account No:   1 Balance: $  500.00
```

- `raise SomeExceptionClass(expression)`

  This `raise` statement raises an exception object from a specific exception class with the error message generated by evaluating `expression`.

  For example,

```
    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except (TypeError, ValueError) as e:
            self._balance = SavingsAccount._defBalance
            if type(e) == ValueError:
                raise ValueError('balance must be numeric')
```

```
<terminated> Secrion2RaiseExample.py [C:\Program Files\Python36\python.exe]
default value has been assigned as balance must be numeric
Account No:   1 Balance: $  500.00
```

- `raise SomeExceptionClass(expression) from OriginalExceptionClass`

  This `raise` statement raises a different exception object and makes the last exception to be its cause. If there is no previous exception, the cause is assigned the value `None`.

  For example,

```
    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        try:
            self.assignBalance(balance)
        except AccountError as e:
            print('default value has been assigned as', e)
            print('The operation failed as it', e.__cause__)


    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except (TypeError, ValueError) as e:
            self._balance = SavingsAccount._defBalance
            if type(e) == ValueError:
                raise AccountError('balance must be numeric') from e
```

```
<terminated> Secrion2RaiseExample.py [C:\Program Files\Python36\python.exe]
default value has been assigned as balance must be numeric
The operation failed as it could not convert string to float: 'a'
Account No:    1 Balance: $   500.00
```

Read

https://docs.python.org/3.3/reference/simple_stmts.html#raise

Read section 7.4 on the raise statement.

https://docs.python.org/3/tutorial/errors.html

Read section 8.4 on raising exceptions.

https://stackoverflow.com/questions/2052390/manually-raising-throwing-an-exception-in-python

Read recommendations on the raise statement.

The best practice on raising exceptions is to raise an exception from the most specific exception class, according to the business rule violation.

Research

Read up on the exc_info() method in the sys module which allows you to extract further details about the exception object.

An exception raised from a user-defined exception class is handled in the same manner as all other exceptions from built-in exception classes, via `except` blocks. In addition, `except` blocks for more specific exceptions should precede those for more specific exceptions.

Activity 8

Study the class `SavingsAccount` and modify it so that `_nextAccountNumber` is incremented only if a `SavingsAccount` object is created:

```
class AccountError(Exception):
    '''Class for exception for account application'''

class InsufficientFundError(AccountError):
    '''Raised when account has insufficient balance to carry out
operation'''

class AccountDetailsError(AccountError):
    '''Raised when account details is invalid'''

class AccountNegativeBalanceError(AccountDetailsError):
    '''Raised when balance is negative'''


class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1

    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        try:
            self.assignBalance(balance)
        except AccountNegativeBalanceError as e:
            print(e)
            raise
        except AccountDetailsError as e:
            print('default value has been assigned as', e)
            print('The operation failed as it', e.__cause__)


    def assignBalance(self, balance):
        try:
            balance = float(balance)
            if balance < 0:
                raise AccountNegativeBalanceError\
                    ('balance must not be negative')
        except (TypeError, ValueError) as e:
            self._balance = SavingsAccount._defBalance
            if type(e) == ValueError:
                raise AccountDetailsError('balance must be numeric')\
                    from e
        else:
            self._balance = balance
```

Use this code fragment to test your modification:

```
count = 0
while True:
    balance = input('enter balance or 0 to end: ')
    if balance == '0':
```

```
            break
        try:
            s =  SavingsAccount(balance)
            print(s)
        except AccountError as e:
            print('Caught this AccountError:', e)
        except Exception as e:
            print('Caught this Exception error:', e)
        finally:
            count += 1
            print('Iteration', count)
print('program end')
```

A sample run is shown here:

```
enter balance or 0 to end: -2
balance must not be negative
Caught this AccountError: balance must not be negative
Iteration 1
enter balance or 0 to end: a
default value has been assigned as balance must be numeric
The operation failed as it could not convert string to float: 'a'
Account No:   1 Balance: $  500.00
Iteration 2
enter balance or 0 to end: -3
balance must not be negative
Caught this AccountError: balance must not be negative
Iteration 3
enter balance or 0 to end: 1000
Account No:   2 Balance: $ 1000.00
Iteration 4
enter balance or 0 to end: 0
program end
```

Notice that the account numbers of the SavingsAccount objects should be in running order even when exceptions are raised in between the creation of SavingsAccount objects.

2.3 Raising and Handling Exceptions in Applications

When building a module or a class, the designer of the supplier code first decide which functions or methods will raise exceptions. Programmers then document the exceptions raised by functions and methods accordingly, using docstrings. The documentation serves to inform callers of the functions and methods that exceptions should be handled as they may be raised. Not handling the exceptions put callers at risk of being terminated abnormally.

The client or user must read the documentation, also called the application programmer interface (API), to effectively use imported functions and methods. The client should either handle the exceptions raised by the methods they call, or

document that they themselves will throw the exception, raised by functions or methods they call.

The exceptions raised or thrown in a function or method are documented using the keyword `raise` or `raises` in the docstring for the function or the method.

For example,

```
    def assignBalance(self, balance):
        """
    Assign the balance of the account with the argument.

    Arguments:
      balance (float): The amount to assign to the balance of account.

    Returns:
      None

    Raises:
      AccountNegativeBalanceError: If the argument is negative.
      AccountDetailsError: If the argument is non-numeric.

    """
        try:
            balance = float(balance)
            if balance < 0:
                raise AccountNegativeBalanceError\
                    ('balance must not be negative')
        except (TypeError, ValueError) as e:
            self._balance = SavingsAccount._defBalance
            if type(e) == ValueError:
                raise AccountDetailsError('balance must be numeric')\
                    from e
        else:
            self._balance = balance
```

Consider the functions supplied in a module called `bmiCalculator`. The designer of functions can decide

- which functions check whether a business rule is violated and raise an exception and
- whether functions will handle the exceptions raised by called functions.

Design 1

The provided functions do not handle exceptions.

```
class InvalidDataError(Exception):
    '''Raised when data is out of range
    '''

def bmi(height, weight):
    '''
```

```
    Return bmi given arguments: height and weight.

    Arguments:
      height (float): The height in metres
      weight (float): The weight in kilograms

    Returns:
      BMI using formula weight/(height * height)

    Raises:
      TypeError: If the arguments are non-numeric.
      InvalidDataError:  If height or weight is not in range
    '''
    if heightInRange(height) and weightInRange(weight):
        return weight/(height * height)

def heightInRange(height):
    '''
    Return True if given argument height is within range

    Argument:
      height (float): The height in metres

    Raises:
      TypeError: If the argument is non-numeric.
      InvalidDataError:  If height not in range of 0.5 and 2.6 metres
    '''
    if height < 0.5:
        raise InvalidDataError('Height must be at least 0.5 meters')
    if height > 2.6:
        raise InvalidDataError('Height must not be more than 2.6 meters')
    return True

def weightInRange(weight):
    '''
    Return True if given argument weight is within range

    Argument:
      weight (float): The weight in kilograms


    Raises:
      TypeError: If the argument is non-numeric.
      InvalidDataError: If weight not in range of 10 and 150 kilograms

    '''
    if weight < 10:
        raise InvalidDataError('Weight must be at least 10 kilograms')
    if weight > 150:
        raise InvalidDataError('Weight must not be more than 150 kilograms')
    return True
```

In this design, the client code must handle exceptions described in the documentation for the functions in the module, failing which the client program will terminate abnormally.

If the argument to the functions `heightInRange(height)` and `weightInRange(weight)` are not floating point numbers, the comparison operators (< and >) will raise a `TypeError`

exception. Furthermore, if the floating point number is outside the valid range, the functions will raise a `InvalidDataError` exception.

The function `bmi(height, weight)` does not handle `TypeError` or `InvalidDataError` exceptions raised by the functions `heightInRange(height)` and `weightInRange(weight)`. Therefore, client code that calls the function `bmi(height, weight)` must handle both `TypeError` and `InvalidDataError` exceptions. Otherwise, when these receptions are raised, the client program will terminate abnormally.

```
from SU4.bmiCalculator import bmi, InvalidDataError

def main():
    try:
        h = 'b'
        w = 1
        print('bmi = {:.2f}'.format(bmi(h, w)))
    except TypeError:
        print('Please provide decimal numbers for both weight and height')
    except InvalidDataError as e:
        print(e)
```

When a called routine raises an exception, there are two possible scenarios regarding the caller:

- caller does not handle the exception.

  For example, the function `main` calls the function `bmi`, and the function `bmi` calls the function `heightInRange`. Both function `main` and function `bmi` do not handle exceptions.

  When the called function `heightInRange(height)` raises a `TypeError` exception,
  - the function `bmi` terminates abnormally,
  - the function `bmi` returns to function `main` with the raised exception.
  - the function `main` also terminates abnormally, and the execution ends with error messages:

```
Traceback (most recent call last):
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU4
    main()
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU4
    print('bmi = {:.2f}'.format(bmi(h, w)))
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU4
    if heightInRange(height)\
  File "C:\Users\Jenny\Desktop\python\MyOwnCodeExample\ICT162\src\SU4
    if height < 0.5:
TypeError: '<' not supported between instances of 'str' and 'float'
```

  Refer to Figure 4.4 for the flow of execution. The lines in blue show function being called whereas the lines in red show function returns with raised exception.

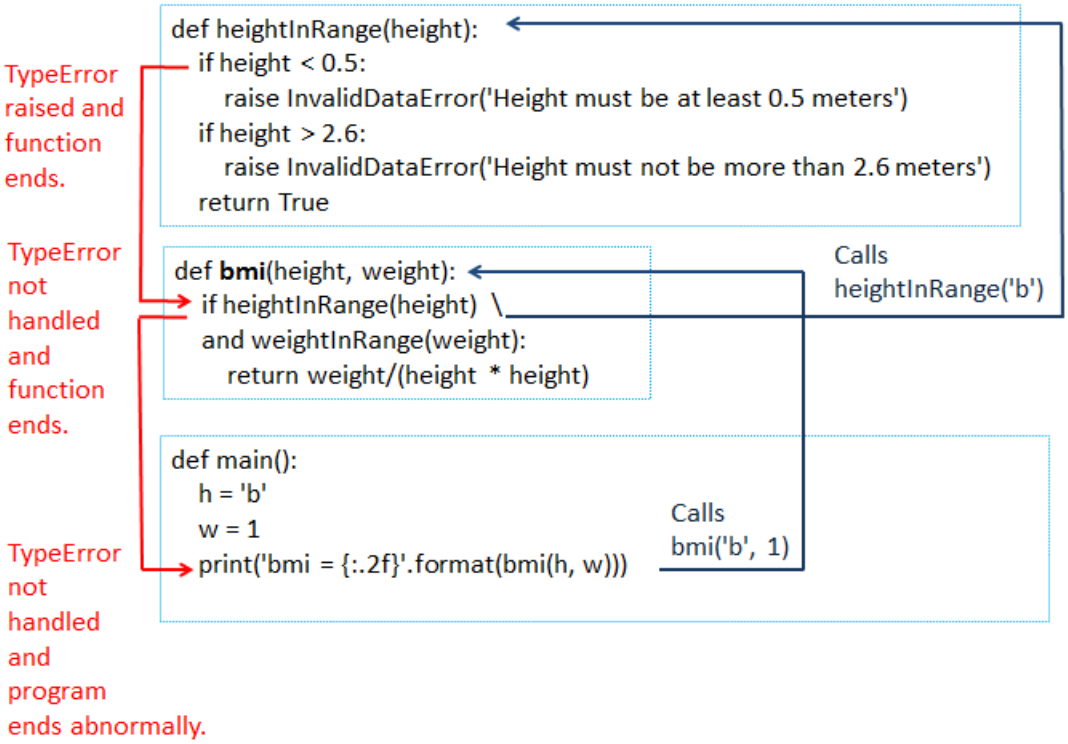


Figure 4.4 Flow of execution when there is a raised exception is not handled by any of the callers.

- caller handles the exception.

  For example, the function `main` handles exception but the function `bmi`, does not.

  When the called function `heightInRange(height)` raises a `TypeError` exception,
  - the function `bmi` terminates abnormally,
  - the function `bmi` returns to function `main` with the raised exception.
  - the function `main` does not terminates abnormally as it handles the raised exception:

```
<terminated> main1.py [C:\Program Files\Python36\python.exe]
Please provide decimal numbers for both weight and height
```

  Refer to Figure 4.5 for the flow of execution. The lines in blue show function being called whereas the lines in red show function returns with raised exception.

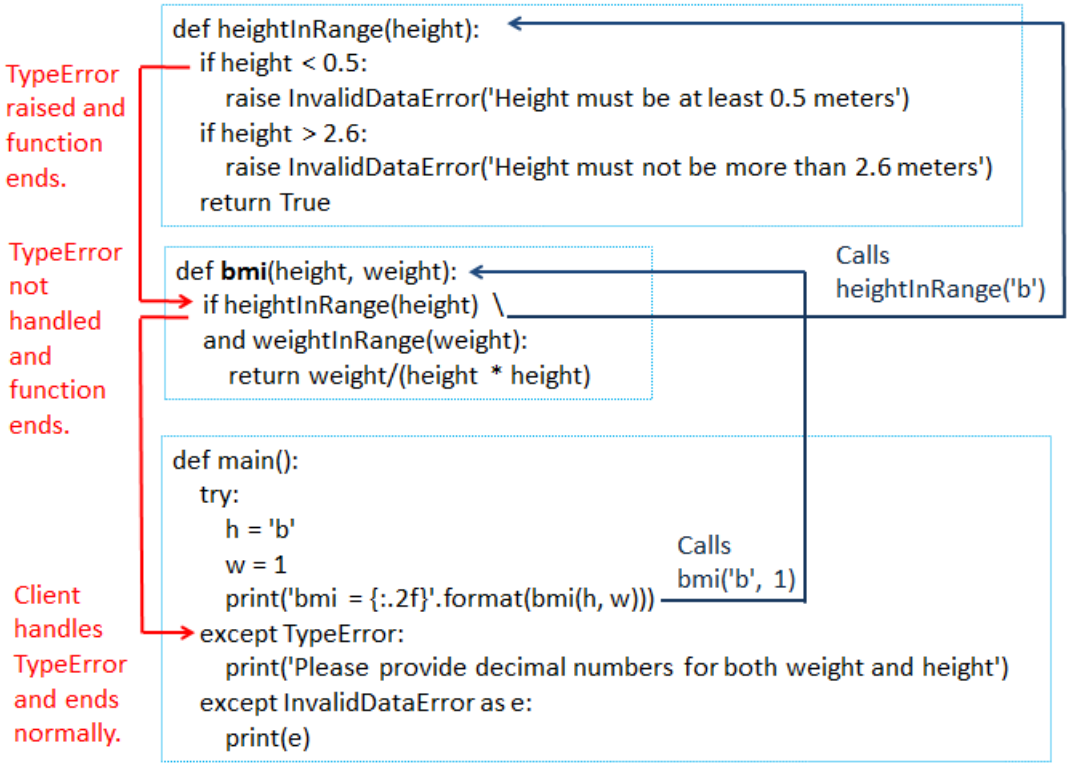


Figure 4.5 Flow of execution when there is a raised exception is handled by one caller

If a function handles the raised exception, the raised exception does not get propagated to its caller.

For example, suppose the function `bmi` handles the exception:

```
def bmi(height, weight):
    '''
    Return bmi given arguments: height and weighn.

    Arguments:
      height (float): The height in metres
      weight (float): The weight in kilograms

    Returns:
      BMI using formula weight/(height * height) pr
     -1 if data is invalid
    '''
    try:
        if heightInRange(height)\
        and weightInRange(weight):
            return weight/(height * height)
    except (TypeError, InvalidDataError) as e:
        print(e)
        return -1
```

When the called function `heightInRange(height)` raises a `TypeError` exception,

- the function `bmi` handles it and ends normally,

```
<terminated> main1.py [C:\Program Files\Python36\python.exe]
'<' not supported between instances of 'str' and 'float'
```

- the function `bmi` returns to function `main` without the raised exception.
- the function `main` need not handle the `TypeError` exception.

Refer to Figure 4.6 for the flow of execution. The lines in blue show function being called whereas the lines in red show function returns with raised exception.

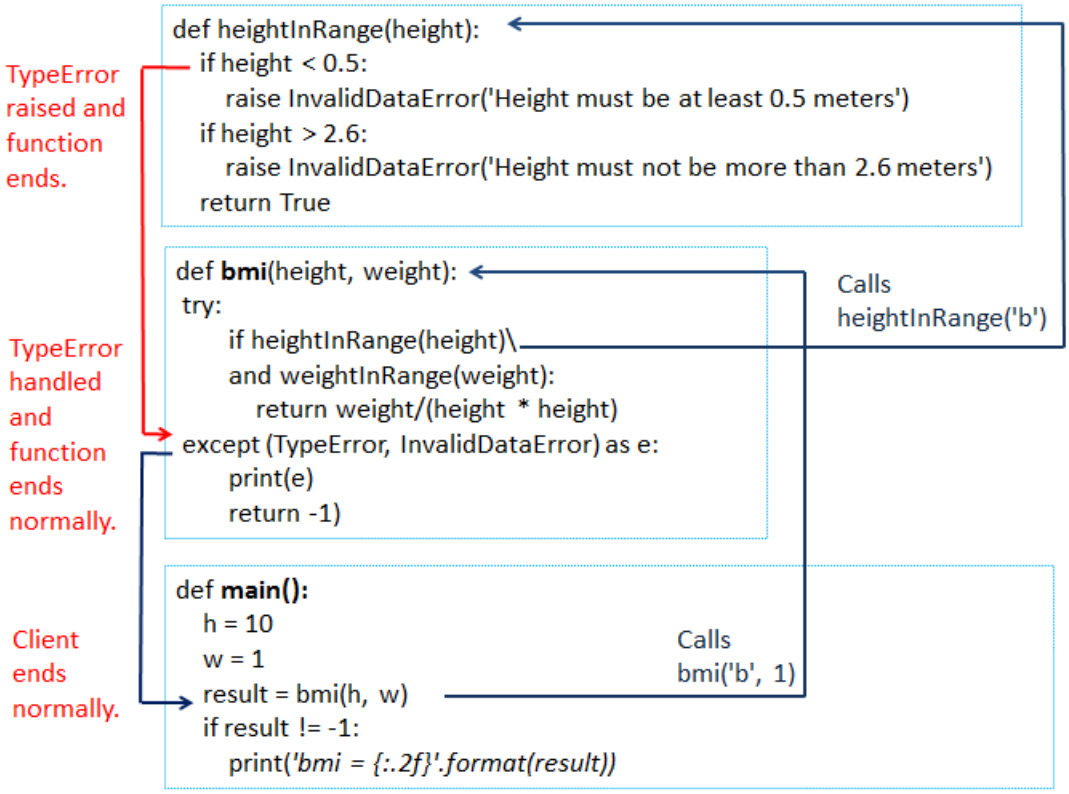


Figure 4.6 Flow of execution when there is a raised exception is handled by one caller

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 183-185.
Retrieved from http://cs.slu.edu/~goldwamh/oopp/

Activity 9

Given the classes **`EmployeeError`** and **`Employee`**.

```
class EmployeeError(Exception):
     '''Raised when Employee operations encounter error'''

class Employee:
    _defSal = 2000
    _maxSalary = 15000
    _minSalary = 1200
    _defPerf = 1
    _perfFactor = {5:2, 4:1.2, 3:0.5, 2:0, 1:0}
    _nextNumber = 1

    @classmethod
    def addPerfFactor(cls, performanceLevel, factor):
        cls._perfFactor[performanceLevel] = factor

    @classmethod
    def removePerfFactor(cls, performanceLevel):
        return cls._perfFactor.pop(performanceLevel, 0)

    def __init__(self, n, salary = None, performance = None,
supervisor= None):
        self._name = n
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1
        self.salary = Employee._defSal if salary is None else salary
        self.performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def empNum(self):
        return self._empNum

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, value):
        self._salary = \
        Employee._minSalary if  value < Employee._minSalary \
        else Employee._maxSalary if value > Employee._maxSalary \
        else value

    def performance(self, value):
        self._performance = value
```

```
    performance = property(None, performance)


    def calculateBonus(self):
        return  Employee._perfFactor.get(self._performance , 0)  *
self._salary

    def increaseSalaryBy(self, percent = 0, base = 20):
        self.salary = self._salary * (1 + percent) + base

    @property
    def supervisor(self):
        return self._supervisor

    @supervisor.setter
    def supervisor(self, supervisor):
        self._supervisor = supervisor

    def __str__(self):
        return ("Emp No: {:3d}  Name: {:<10}  Performance Level: " \
                + "{:1d}  Salary: ${:8.2f} Bonus: ${:8.2f} Supervisor:
{:}") \
                .format(self._empNum, self._name, self._performance, \
                        self._salary, self.calculateBonus(),\
                        "None" if self._supervisor is None \
                        else self._supervisor._name)
```

Modify the class `Employee` as described:

a) The salary given to an employee cannot be less than the minimum salary, `_minSalary` or more than the maximum salary, `_maxSalary`. If the salary violates this rule, do not adjust the salary to the lower or upper limit but instead, raise an error with a message for the cause of the violation.

b) The performance level given to an employee must be one of the key values in the dictionary `_perfFactor`. If the performance level violates this rule, raise an error with a message for the cause of the violation.

c) If the salary increment is more than the current salary, raise an error with a message for the cause of this violation.

d) The constructor should increment `Employee._nextNumber` only if there is not error when the `Employee` object is created.

Activity 10

Given the classes **`CompanyError`** and **`Company`**.

```
class CompanyError(Exception):
        '''Raised when Company operations encounter error'''

class Company (object):
    def __init__(self, n):
        self._name = n
        self._employees = []

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, n):
        self._name = n

    def locate(self, num):
        for e in self._employees:
            if e.empNum == num:
                return e
        return None

    def add(self, e):
        if self.locate(e.empNum) is None:
            self._employees.append(e)
            return True
        else:
            return False

    def isSupervisor(self, s):
        for e in self._employees:
            if e.supervisor == s:
                return True
        return False

    def remove(self, num):
        s = self.locate(num)
        if s is None:
            return False
        if self.isSupervisor(s):
            return False
        self._employees.remove(s)
        return True

    def listAll(self):
        s = ''
        for i, emp in enumerate(self._employees, start = 1):
            s += str(i) + ') ' + str(emp) +  '\n'
        return s or 'No employee\n'


    def __str__(self):
        return "\nCompany Name: " + self.name + "\n" + self.listAll()
```

Modify the class `Company` as described:

a) Raise an error with a message for the cause if an attempt is made to add an employee with the same employee number as that for an existing employee.

b) Add a method that accepts a percentage and a base amount that are used as parameters to increase the salaries of every employee in the company. Do not handle exceptions if they are raised.

c) Raise an error with a message for the cause when removing an employee if
   - the supplied employee number does not belong to an existing employee,
   - the employee is a supervisor

d) Add a method that accepts an employee number and a performance level, The method raises an error with a message for the cause if the employee number does not belong to an existing employee. Otherwise, update the performance level of the employee with the supplied employee number. The method returns True if the update is successful. Do not handle exceptions if they are raised.

Activity 11

Make modification to the program to handle raised exceptions from the user defined exception classes `EmployeeError` and `CompanyError`.

```
def displayMenu():
    while True:
        print('Menu\n' + \
              '1. Hire Employee\n'+\
              '2. Terminate Employee\n' + \
              '3. List Employees\n' +\
              '4. Increase Salary of All Employees\n' +\
              '5. Change Performance level of Employee\n' +\
              '0. Exit')

        option = input('Enter option:')
        if '0' <= option<= '5':
            return int(option)
        print('Enter 0 to 5 only')

def pickSupervisor(company):
    output = company.listAll() or None
    if output is None or output == 'No employee\n':
        print("No employee yet. Can't choose a supervisor")
        return None
    print('Choose supervisor or <enter> if employee has no
supervisor\n' + output)
    s = input('Enter supervisor employee number:') or None
    if s is None:
        return None
    return company.locate(int(s))

def addEmployee(company):
    name = input('Enter employee name:') or None
    if name is None:
        print('Invalid data')
        return False
    salary = input('Enter employee salary or <Enter> for default:') or
None
    if not salary is None:
        salary = float(salary)
    perf = input('Enter employee performance or <Enter> for default:')
or None
    if not perf is None:
        perf = int(perf)
    supervisor = pickSupervisor(company)
    company.add(Employee(name, salary, perf, supervisor))
    return True

def removeEmployee(company):
    num = input('Enter employee number to terminate:') or None
    if num is None:
        print('Invalid data')
        return False
    return company.remove(int(num))

def increaseSalary(company):
    percent = input('Enter percentage to increase or <Enter> for
```

```
        default:') or None
            if not percent is None:
                percent = float(percent)
            base = input('Enter base amount to add to salary or <Enter> for
        default:') or None
            if not base is None:
                base = float(base)
            company.increaseSalaryBy(percent, base)

        def updatePerformance(company):
            num = input('Enter employee number to update performance:') or
        None
            if num is None:
                print('Invalid data')
                return False
            performance = input('Enter new performance level:') or None
            if performance is None:
                print('Invalid data')
                return False
            company.updatePerformance(int(num), int(performance))

        def listEmployees(company):
            print(company.listAll())

        def main():
            company = Company('SUSS')

            while True:
                option = displayMenu();
                if option == 1:
                    if addEmployee(company):
                        print("Employee successfully added")
                    else:
                        print("Add Employee Error")
                elif option == 2:
                    if removeEmployee(company):
                        print("Employee successfully removed")
                    else:
                        print("Remove Employee Error")
                elif option == 3:
                    listEmployees(company)
                elif option == 4:
                    increaseSalary(company)
                elif option == 5:
                    updatePerformance(company)
                elif option == 0:
                    break;
            print('Application exiting')
```

Summary

An exception is raised when the runtime system or the interpreter is unable to successfully execute program statements. Two ways of dealing with exception are: prevention or applying defensive programming though checks before invoking the program statements, and recovery or handling raised exception using the `try-except-else-finally` construct or the `with` statement, both provided in the Python language as a means of controlling the flow of program execution.

The `try-except-else-finally` construct allows program statements to be placed into various blocks: the `try` block contains statements that can potentially raise exceptions, one or more `except` blocks contain statements that respond to or recover from various raised exceptions, an optional `else` blocks contains statements that should execute only if no exception is raised, and an optional `finally` block contains statements that execute regardless whether an exception is raised. When there are more than one `except` blocks, the `except` blocks for more specific exception classes should be placed before `except` blocks for more general exception classes.

The `try-except-else-finally` construct can be nested, appearing within a `try`, an `except`, an `else` or a `finally` block.

The `with` statement is like a `try-finally` construct without the `except` and `else` blocks. It uses the methods `__enter__` and `__exit__` defined in a context manager to perform clean-up action for a resource object such as a file.

One or more user-defined exception class can be implemented for exceptions raised due to various business rule violations. A user-defined exception class must inherit from the class `Exception`, either directly or indirectly. The class definition is kept simple as usually, there is no necessity to override any methods such as `__init__` and `__str__` from its superclass.

When business rules are violated, a function that checks for the violation will raise exceptions using the `raise` statement. A function that calls such a function must either handle the exception or document that it raises exceptions.

References

Goldwasser, M. H., & Letscher, D. (2014). *Object-Oriented Programming in Python [pdf version]. Retrieved from http://cs.slu.edu/~goldwamh/oopp/*

https://docs.python.org/3/tutorial/errors.html

https://docs.python.org/3/library/exceptions.html#bltin-exceptions

https://docs.python.org/3.3/reference/compound_stmts.html#with

https://docs.python.org/3.3/reference/simple_stmts.html#raise

https://stackoverflow.com/questions/2052390/manually-raising-throwing-an-exception-in-python

Formative Assessment

1. An exception is raised as a result of _____.

A) a syntax error
@ Incorrect. A program cannot execute when there is a syntax error. Refer to Section 1.1 Introduction to Exception Handling, Study Unit 4.

*B) a runtime error
@ Correct. When a runtime error occurs, an exception has been raised. Refer to Section 1.1 Introduction to Exception Handling, Study Unit 4.

C) a logic error
@ Incorrect. Logic error occurs when a program does not meet the specification of the problem. Refer to Section 1.1 Introduction to Exception Handling, Study Unit 4.

D) All of the above
@ Incorrect. An exception is raised only when a runtime error occurs. Refer to Section 1.1 Introduction to Exception Handling, Study Unit 4.

2. Which statements should be placed into an `else` block?

A) Statements which can potentially raise exceptions
@ Incorrect. An `else` block should not contain statements that can potentially raise exceptions. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

B) Statements which can respond to or recover from a raised exception
@ Incorrect. An `else` block does not contain statements that respond to or recover from a raised exception. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

*C) Statements which should execute only if statements executed in the `try` block do not raise any exception,
@ Correct. An `else` block contains statements that should execute only if executing statements in the `try` block does not raise any exception. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

D) Statements that should execute regardless whether executing statements in the `try` block raises exception
@ Incorrect. An `else` block does not contain statements that should execute regardless whether executing statements in the `try` block raises exception. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

3. When a statement in the `try` block raises an error, what will be the flow of execution?

A) Complete executing the `try` block, execute the `except` block that can handle the raised exception, skip the `else` block and then execute the `finally` block.
@ Incorrect. The flow of execution is incorrect. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

B) Complete executing the `try` block, execute the `except` block that can handle the raised exception, execute the `else` block and then execute the `finally` block.
@ Incorrect. The flow of execution is incorrect. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4

*C) Abandon the `try` block, execute the `except` block that can handle the raised exception, skip the `else` block and then execute the `finally` block.
@ Correct. The flow of execution is correct. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

D) Abandon executing the `try` block, execute the `except` block that can handle the raised exception, execute the `else` block and then execute the `finally` block.
@ Incorrect. The flow of execution is incorrect. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

4. When should user-defined exception classes be created?

A) When a business rule is violated
@ Incorrect. A business rule violation causes an exception to be raised. Refer to Section 2.1 Creating Exception Classes, Study Unit 4.

B) When a semantically invalid program statement is executed
@ Incorrect. A semantically invalid program statement causes an exception to be raised. Refer to Section 2.1 Creating Exception Classes, Study Unit 4.

C) When it is necessary for an application to handle exceptions
@ Incorrect. An application uses the `try-except-else-finally` construct to handle exceptions. Refer to Section 1.2 try-except-else-finally construct, Study Unit 4.

*D) When we wish to distinguish between exceptions caused by business rule violation and exceptions caused by semantically invalid program statements
@ Correct. User-defined exception classes allow us to distinguish between exceptions caused by business rule violation and exceptions caused by semantically invalid program statements. Refer to Section 2.1 Creating Exception Classes, Study Unit 4.

5. Which action cannot be done by using a `raise` statement?

*A) raise one or more new exceptions
@ Correct. A `raise` statement can raise only one exception. Refer to Section 2.2 Raising Exceptions, Study Unit 4.

B) re-raise a previous exception
@ Incorrect. A previous exception can be re-raised using a `raise` statement. Refer to Section 2.2 Raising Exceptions, Study Unit 4.

C) raise a different exception from a previous exception
@ Incorrect. A different exception from a previous exception can be raised using a `raise` statement. Refer to Section 2.2 Raising Exceptions, Study Unit 4.

D) All of the above can be done by using a `raise` statement
@ Incorrect. Only one exception can be raised using a `raise` statement. Refer to Section 2.2 Raising Exceptions, Study Unit 4.

6. Which two policies for raising exceptions are best practices?
   i. Choose to raise an exception from built-in exception classes over raising an exception from user-defined exception classes
   ii. Choose to raise an exception from user-defined exception classes over raising an exception from built-in exception classes
   iii. Raise an exception from the most general exception class, according to the business rule violation
   iv. Raise an exception from the most specific exception class, according to the business rule violation.

A) i. and iii.
@ Incorrect. These are not the two policies. Refer to Section 2.1 Creating Exception Classes and to Section 2.2 Raising Exceptions, both in Study Unit 4.

B) i. and iv.
@ Incorrect. These are not the two policies. Refer to Section 2.1 Creating Exception Classes and to Section 2.2 Raising Exceptions, both in Study Unit 4.

C) ii. and iii.
@ Incorrect. These are not the two policies. Refer to Section 2.1 Creating Exception Classes and to Section 2.2 Raising Exceptions, both in Study Unit 4.

*D) ii. and iv.
@ Correct. These are the two policies. Refer to Section 2.1 Creating Exception Classes and to Section 2.2 Raising Exceptions, both in Study Unit 4.

7. Which method or function must document raised exceptions?
   i. Any method or function that raises exceptions.
   ii. Any method or function that raises exceptions but handles them.
   iii. Any method or function that raises exceptions but does not handle them.
   iv. Any method or function that calls a method or function which raises an exception.
   v. Any method or function that calls a method or function which raises an exception, and the calling method or function handles the exception.
   vi. Any method or function that calls a method or function which raises an exception, and the calling method or function does not handle the exception.

A) i.

@ Incorrect. An additional criterion is required. Refer to Section 2.3 Raising and Handling Exceptions in Applications Study Unit 4.

B) i. and iv.
@ Incorrect. An additional criterion is required. Refer to Section 2.3 Raising and Handling Exceptions in Applications Study Unit 4.

C) ii. and v.
@. Incorrect. The criteria are incorrect. Refer to Section 2.3 Raising and Handling Exceptions in Applications Study Unit 4.

*D) iii. and vi.
@ Correct. The criteria are incorrect. Refer to Section 2.3 Raising and Handling Exceptions in Applications Study Unit 4.

Solutions or Suggested Answers

Activity 1

a) TypeError: 'a' is an invalid keyword argument for this function
Runtime error

```
a=1
print(a)
```

b) if married != 'y' or != 'n': ^
SyntaxError: invalid syntax

```
married = input('Married? y or n: ')
if married != 'y' or married != 'n':
     pass
```

c) NameError: name 'a' is not defined
Runtime error

```
a=1
print(a)
```

d) Logic error
Output is Mr

```
gender = input('Gender? m or f: ')
if gender != 'f' and gender != 'F':
    print('Mr.')
else:
    print('Ms.')
```

e) print('Answer is ', b**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'

Runtime error

```
b = int(input('Enter a number: '))
print('Answer is ', b**2)
```

f) Logic error
Output is the smaller number

```
a. 
b. a = float(input('Enter a number: '))
c. b = float(input('Enter a number: '))
d. if a > b:
e.     bigger = a
f. else:
g.     bigger = b
```

h. `print('The bigger number is ', bigger)`

Activity 2

a)
```
try:
    print(a=1)
except TypeError as e:
     print(e)
```

b) No runtime error

c)
```
try:
    print(a)
except NameError as e:
```

i.
```
    print(e)
```

d) No runtime error

e)
```
try:
    b = input('Enter a number: ')
    print('Answer is ', b**2)
except TypeError as e:
    print(e)
```

f) No runtime error

Activity 3

```
iterNum = 0
while True:
    numstr = input('Enter how many numbers to average or <Enter> to end: ')
    if numstr == '':
        break
    total = 0
    try:
        num = int(numstr)
        for i in range(num):
            total += float(input('Enter number: '))
        average = total/num
    except Exception as e:
        print("An error has occurred:" , e)
    else:
        print('Average is ', average)
    finally:
        iterNum += 1
        print('Iteration', iterNum)
print('Program ended')
```

Activity 4

```
iterNum  = 0
```

```
while True:
    numstr = input('Enter how many numbers to average or <Enter> to end: ')
    if numstr == '':
        break
    total = 0
    try:
        num = int(numstr)
        count = 0
        while count != num:
            try:
                total += float(input('Enter number: '))
            except ValueError as e:
                print('Input is not a number')
            else:
                count += 1
        average = total/num
    except ValueError as e:
        print("An input error has occurred:" , e)
    except ZeroDivisionError as e:
        print("A computation error has occurred:" , e)
    else:
        print('Average is ', average)
    finally:
        iterNum  += 1
        print('Iteration', iterNum)
  print('Program ended')
```

Activity 5

```
class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1

    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        self.assignBalance(balance)


    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except TypeError:
            self._balance = SavingsAccount._defBalance

    def __str__(self):
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumber, self._balance)

s =  SavingsAccount()
print(s)
```

Activity 6

```
class SavingsAccount(object):
    _defBalance = 500
    _nextAccountNumber = 1
```

```
    def __init__(self, balance = None):
        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
        self.assignBalance(balance)


    def assignBalance(self, balance):
        try:
            self._balance = float(balance)
        except (TypeError, ValueError):
            self._balance = SavingsAccount._defBalance

    def __str__(self):
        return ("Account No: {:3d} Balance: ${:8.2f}") \
                .format(self._accountNumber, self._balance)

s =  SavingsAccount(10000)
print(s)
```

Activity 7

a)

```
class LibraryError(Exception):
    '''Class for exception for the library application'''

try:
...
except LibraryError:
    <do something>
```

b)

```
class StudentError(Exception):
    '''Base class for exception for the student application'''

class DisqualificationError(StudentError):
    '''Raised when a person does not qualify to be a student'''

class NoPaymentError(StudentError):
    '''Raised when student has not paid program fees'''

class NumberOfModulesExceededError(StudentError):
    '''Raised when student register for more modules than entitled'''

try:
...
except DisqualificationError:
    <do something>
except NoPaymentError:
    <do something>
except NumberOfModulesExceededError:
    <do something>
except StudentError:
    <do something>
```

c)

```
class InufficentInfoError(Exception):
    '''Vlass for exception for the order delivery application'''

try:
...
except InufficentInfoError:
    <do something>
```

Activity 8

```
    def __init__(self, balance = None):
        try:
            self.assignBalance(balance)
        except AccountNegativeBalanceError as e:
            print(e)
            raise
        except AccountDetailsError as e:
            print('default value has been assigned as', e)
            print('The operation failed as it', e.__cause__)

        self._accountNumber = SavingsAccount._nextAccountNumber
        SavingsAccount._nextAccountNumber += 1
```

Activity 9

```
class Employee:
...
    def __init__(self, n, salary = None, performance = None, supervisor= None):
        self._name = n
        self.salary = Employee._defSal if salary is None else salary
        self.performance = Employee._defPerf if performance is None \
        else performance
        self._supervisor = supervisor
        self._empNum = Employee._nextNumber
        Employee._nextNumber += 1

...
    @salary.setter
    def salary(self, value):
        if value < Employee._minSalary:
            raise EmployeeError('Salary cannot be less than $' +
str(Employee._minSalary))
        if value > Employee._maxSalary:
            raise EmployeeError('Salary cannot be more than $' +
Employee._maxSalary)
        self._salary = value

    def performance(self, value):
        if value not in Employee._perfFactor.keys():
            raise EmployeeError('Performance level is invalid')
        self._performance = value

...
    def increaseSalaryBy(self, percent = 0, base = 20):
```

```
        increment = self._salary * percent + base
        if increment > self._salary:
            raise EmployeeError('Increment $' + str(increment) +  \
                                ' is greater than salary $' + str(self._salary))
        self.salary = self._salary * (1 + percent) + base
```

Activity 10

```
class Company (object):
…
   def add(self, e):
        if self.locate(e.empNum) is None:
            self._employees.append(e)
            return True
        else:
            raise CompanyError('Employee number ' + str(e.empNum) + ' already
exists')


    def increaseSalaryBy(self, percent = 0, base = 20):
        for e in self._employees:
            e.increaseSalaryBy(percent, base)

    def remove(self, num):
        s = self.locate(num)
        if s is None:
            raise CompanyError('Employee number ' + str(num) + ' does not exist')
        if self.isSupervisor(s):
            raise CompanyError('Employee is a supervisor')
        self._employees.remove(s)
        return True

    def updatePerformance(self, num, performance):
        e = self.locate(num)
        if e is None:
            raise CompanyError('Employee number ' + str(num) + ' does not exist')
        e.performance = performance
        return True
…
```

Activity 11

```
def main():
    company = Company('SUSS')

    while True:
        option = displayMenu();
        if option == 0:
            break;
        try:
            if option == 1:
                if addEmployee(company):
                    print("Employee successfully added")
                else:
                    print("Add Employee Error")
            elif option == 2:
                if removeEmployee(company):
```

```
                print("Employee successfully removed")
            else:
                print("Remove Employee Error")
        elif option == 3:
            listEmployees(company)
        elif option == 4:
            increaseSalary(company)
        elif option == 5:
            updatePerformance(company)
    except Exception as e:
        print(e)
print('Application exiting')
```

STUDY UNIT 5
GRAPHICAL USER INTERFACE

Learning Outcomes

By the end of this unit, you should be able to:

1. Identify and describe GUI components
2. List the different widgets and layout managers
3. Demonstrate how to create and lay out widgets using layout manager
4. Explain and implement event handling
5. Apply object oriented programming to GUI implementation
6. Illustrate MVC framework

Overview

The focus of this study unit is graphical user interface (GUI) application using tkinter which is bundled with Python. tkinter is a Python interface to Tk, an open source cross platform widget toolkit.

The different types of widgets that make up a GUI and the various layout managers that position widgets are first discussed. The study unit also demonstrates how the different types of widgets are created and how layout managers are used to position them.

The study unit covers event handling. User actions on the GUI generate events. To be able to respond to these events, the GUI application must include event-handling code, that is, code that will execute under certain user actions. The study unit discusses two ways of binding event handling code to widgets that generate events

The study unit finally discusses how object-oriented programming techniques are applied to GUI application, to modularize the implementation and to separate the various programming concerns. In particular, the study unit illustrates the model-view-controller (MVC) framework, which separates the implementation of the data (model) from the implementation of various views of the data, and allows the data and the view to communicate indirectly through the controller l.

This study unit covers chapter 15 of the textbook. As there are also several online resources to read, it is estimated that the student will spend about 7 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 Programming GUI with tkinter

1.1 Introduction to GUI and Event Driven Programming

Python provides a number of frameworks for programmers to implement graphical user interface (GUI). In this course, we use tkinter which is bundled with Python. tkinter is a Python interface to Tk, an open source cross platform widget toolkit which is an extension of the scripting language Tcl.

A typical GUI includes widgets for

- data entry

  Widgets allow a user to enter data, e.g., text fields and radio buttons.

- output

  Widgets display output e.g., scrollable texts and labels.

- information display

  Widgets display instruction or provide information to users e.g., labels and progress bar.

- event generation

  Widgets generate events in response to user actions. The events elicit a response from the GUI application.

  For example, a user can click on a button to generate a button-clicked event and cause the GUI to execute a function in response to the button click. A user can also move the cursor out of a text field to generate a focus-out event and cause the GUI to execute a different function in response to the focus-out event.

- containing other widgets

  Container widgets help group related widgets and simplify their management.

Figure 5.1 shows an example GUI with some commonly used widgets. The example GUI contains the following widgets:

- 2 labels to provide information to user on what input is expected from the user, specifically weight and height.
- 2 text fields to allow user to enter input for weight and height.
- 2 radio buttons to allow user to choose between entering the height in m (metres) and in cm (centimetres).
- 2 buttons to allow user to invoke either a function to calculate the bmi or a function to reset the GUI.
- A scroll text to display the output. The computed bmi is appended to the scroll text.
- 5 containers to group related widgets.

  The containment is depicted in Figure 5.2.

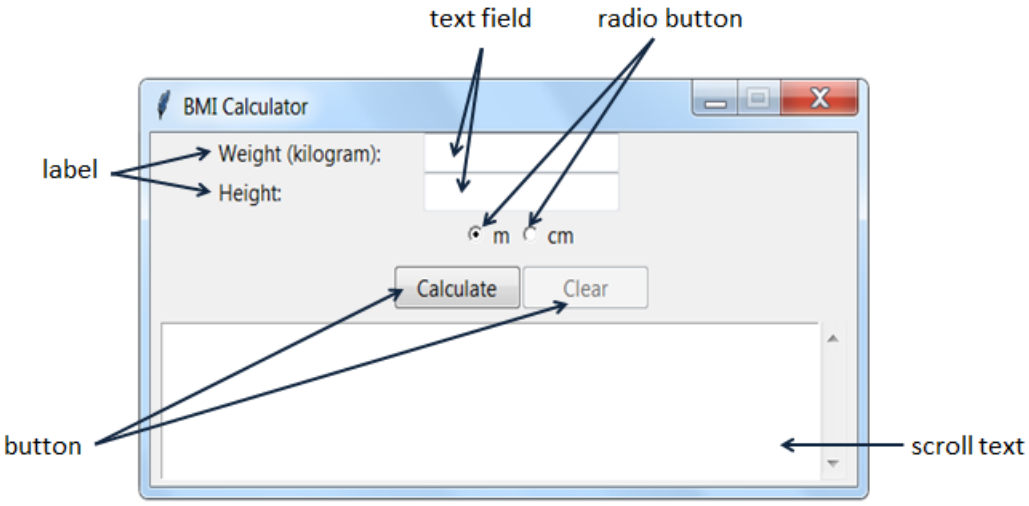


Figure 5.1 An example GUI with commonly used widgets

Activity 1

Identify the purposes for the widgets in the following scenario:

A user entered his user id and password in two text fields, appropriately labelled user id and password, respectively. After clicking on the button labelled login, the GUI application shows an error dialog with the error message that the user id or password is incorrect. The user clicks the ok button in the error dialog to close the dialog.

The implementation of a GUI entails the following steps:

- Identify the widgets necessary for the application

  Identify widgets for the input and output requirements of the application, as well as, widgets for providing instruction or information to a user, and widgets for user to perform actions on, to generate events to call functions.

  Widgets are covered in section 1.2.

- Identify the container widgets required to contain widgets

  Every GUI application has a top level frame that acts as a container to hold widgets such as buttons, text fields or other non-top level containers. These containers group widgets and help simplify their management.

  The container widget and contained widgets have a parent-child relationship which is a containment relationship, not an inheritance relationship.

  Figure 5.2 shows the containment or parent-child relationships for the widgets in the GUI in Figure 5.1.

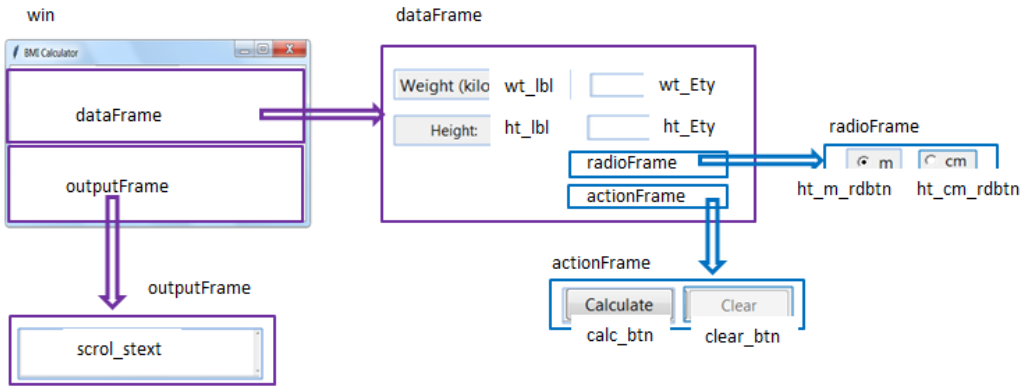


Figure 5.2 Parent-child relationships in pictorial form

The containment relationships or parent-child relationships can also depicted in a tree hierarchy as shown in Figure 5.3:

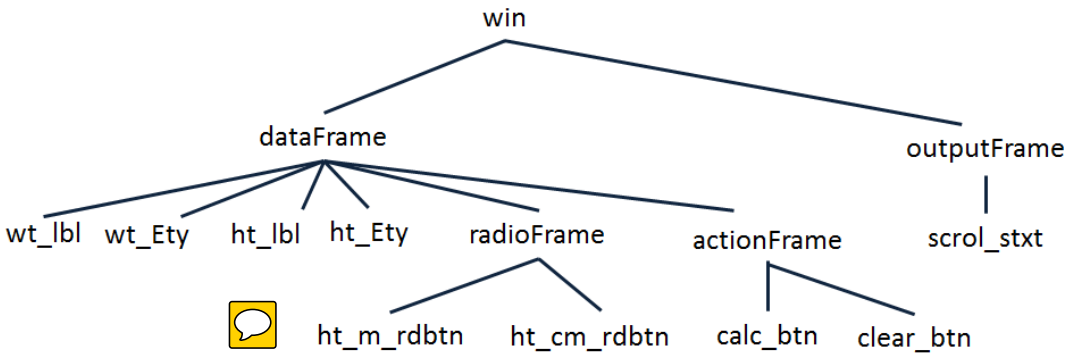


Figure 5.2 Parent-child relationships in tree hierarchy form

Naming conventions used on widgets can help identify the type (class) of widget a name refers to.

For example, a particular naming convention may apply the following suffix to names of widgets so that names with the suffix

- `Frame` refer non-top level containers,
- `lbl` refer labels,
- `Ety` refer data entry widgets or text fields,
- `rdbtn` refer radio buttons,
- `btn` refer buttons and
- `stxt` refer scroll text.

The parent-child relationships in Figure 5.2 and Figure 5.3 are listed here:

- the top level container is `win`. `win` is the root widget and it does not have a parent.
- `dataFrame`, `radioFrame`, `actionFrame` and `outputFrame` are non-top level containers.

  `dataFrame` and `outputFrame` are children of `win`. Alternatively, `win` is the parent of `dataFrame` and `outputFrame`.
- `wt_lbl`, `wt_Ety`, `ht_lbl`, `ht_Ety`, `radioFrame` and `actionFrame` are children of `dataFrame`.

- `ht_m_rdbtn` and `ht_cm_rdbtn` are children of `radioFrame`.
- `calc_btn` and `clear_btn` are children of `actionFrame`.
- `scrol_stxt` is a child of `actionFrame`.

Container widgets are also covered in section 1.2.

Activity 2

Name the widgets and show them their parent-child relationships in tree hierarchy form for the widgets in Activity 1.

- Identify the layout manager to position the widgets in their containers

  There are three types of layout managers in tkinter: pack, grid and place.

  Each container can use only one layout management to position the widgets it is a parent for. However, different containers may use different layout management.

  Layout management is covered in section 1.3.

- Identify how a user interacts with the GUI application to trigger a response from the GUI application

  - Identify the widgets that generate events when user performs certain actions.
  - Write the functions to handle the events.

    Functions must be written to handle each event that the GUI application has a response for.
  - Associate the functions to the widgets that generate the events.

  This type of programming is called **event-driven programming** as the functions that handle events are executed only when the events occur. Event-driven programming is covered in section 1.4.

For the example GUI shown in Figure 5.1, the button `calc_btn` generates an event that is handled by the function `calcBMI`. The function `clear` handles the event generated by the button `clear_btn`.

Read

https://docs.python.org/3/library/tk.html

1.2 Widgets

Once the widgets are identified, the next step is to

- create them using the constructor of classes of the widgets found in the various packages and modules in tkinter,
- configure the widgets and then
- place them into containers.

The focus of this section is widgets and container widgets. Thus, the simplest layout manager, the pack geometry manager which provides simple layout management, will be used to place widgets into container widgets. The pack geometry manager is covered in greater detail in section 1.3.

To create the widget for:

- top level container

  Every GUI application must have a top level container, also called the root widget. The root widget is associated with the Tcl interpreter.

  The root widget is created by calling a constructor of the class `Tk`.

```
import tkinter as tk # rename tkinter as tk
win = tk.Tk() # call the constructor to create the top level container
```

  If the GUI application does not explicitly create this root container, Python will create one for the application.

  The root widget may be configured such as:

```
win.resizable(False, False) # prevent it from being
                            # resized at both width and height

win.title("BMI Calculator") # give it a title

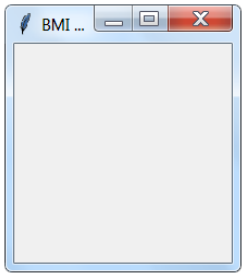
win.geometry("400x450")     # give it an initial width and height
```

We will not set the initial width and height of the root widget so that we can see how tkinter adjusts the size of the root widget to accommodate widgets added to it.

It is recommended that tkinter is allowed to determine the best size for the root widget, based on the size of the widgets added to it.

Once the required widgets have been created and placed into the root widget, the root widget is made to execute in a loop, listening for events generated by user actions and executing functions associated with these events.

```
win.mainloop()
```



Activity 3

Create a root widget and named it `inputFrame`, for the GUI described in Activity 1. Set the title of the frame to Login

- non-top level container

A non-top level container is useful for grouping related widgets so that the widgets can be configured together, possibly in a loop. Placing a non-top level in a specific position of its parents will also place the widgets it contains into that position.

A non-top level container can be created from a number of classes such as `Frame` and `Labelframe`, both from the themed tkinter package, `ttk` which gives better-looking widgets.

```
from tkinter import ttk # import themed tkinter package
dataFrame = ttk.Frame(win) #created with win as parent
```

Notice that the actual argument to the constructor `Frame` is `win`. This makes `win` the parent of `dataFrame`.

To show up a widget in the GUI, the widget must be placed into its container, and the container placed into the top level container. As we are using the pack layout manager, we invoke the method `pack` without specifying any arguments, to place `dataframe` into `win`.

```
dataFrame.pack()
```

As `dataframe` does not contain any widget, it is given a minimum space by the pack layout manager.

- label

A label is useful for providing information or instruction to users.

It is created using the constructor from the class `Label` in the themed tkinter package.

```
wt_lbl = ttk.Label(dataFrame, text="Weight (kilogram):", \
                   width = 20, font=("Helvetica", 12),\
                   justify=tk.LEFT, anchor=tk.W) # anchor to west
# specify parent is dataframe
```

Notice that the first actual argument to the constructor `Label` is `dataFrame`. This makes `dataFrame` the parent of `wt_lbl`.

A label object can be configured at creation time. In this example, the label is given a text with a certain font type, font size and justification, a width and an anchor to position the label in the location it is given in the container.

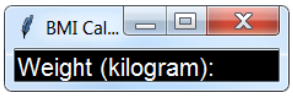
A widget can also be configured after being created, using either the method `configure` or the method `config`.

For example,

```
wt_lbl.config(foreground = 'white',  background = 'black')
```

The label must be put in its container using the method `pack` to make it show up in the GUI.

```
wt_lbl.pack()
```



- text field for data entry

A text field contains user-input text which is stored as a string variable, an object from the class `StringVar`. The `StringVar` object is created before creating the text field.

```
weight = tk.StringVar() #string variable for textfield or Entry
                        # default value is empty string
```

A text field is created using the constructor of the class `Entry` in the themed tkinter package.

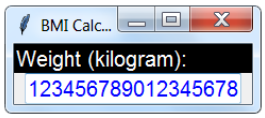
```
wt_Ety = ttk.Entry(dataFrame, width=18, textvariable=weight)
```

In this example, the parent of the text field is `dataFrame`, the width of the text field is `18` characters and the string variable for its content is `weight`.

An `Entry` object can be configured at creation time or after being created, also using the method `configure` or the method `config`. It is placed into its container using the method `pack`.

For example,

```
wt_Ety.configure(foreground = 'blue', font=("Helvetica", 12))
wt_Ety.pack()
```

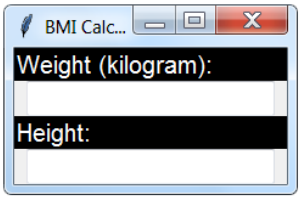


The following code fragment adds the label for height and an entry for user to input height. Note that the entry for height also requires a `StringVar` object .

```
ht_lbl = ttk.Label(dataFrame, text="Height:", width = 20, \
                   font=("Helvetica", 12),
                   justify=tk.LEFT, anchor=tk.W)
ht_lbl.configure(foreground = 'white',  background = 'black')
ht_lbl.pack()
```

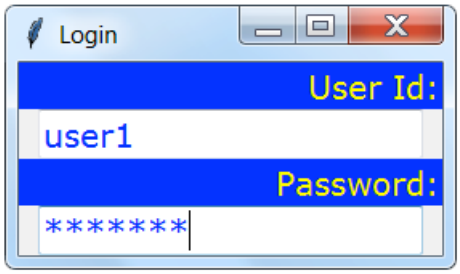
```
height = tk.StringVar()
ht_Ety = ttk.Entry(dataFrame, width=18, textvariable=height)
ht_Ety.configure(foreground = 'blue', font=("Helvetica", 12))

ht_Ety.pack()
```



Activity 4

Create and place the labels and text fields into `inputFrame`, for a GUI shown below. The labels and text fields use font type Verdana and font size 12.



Research how you will show up * in place of the characters user enters in the text field for password.

- radio button

  Radio buttons come in a group, and at most, only one radio button in the group can be selected at any time.

  It is usually the case that we place radio buttons belonging to a group into a dedicated container.

  In this example, we create `radioFrame`, a non-top level container to be the parent of the two radio buttons `ht_m_rdbtn` and `ht_cm_rdbtn` that we will create.

  `radioFrame` is created as a child of `dataFrame`, and is placed into `dataFrame`.

```
radioFrame = ttk.Frame(dataFrame)
radioFrame.pack()
```

A radio button has a value. When one radio button in a group is selected, its value should be stored in an integer variable. Radio buttons belonging to the same radio button group should use the same integer variable.

The two radio buttons `ht_m_rdbtn` and `ht_cm_rdbtn` use the same variable, `radValue`. `ht_m_rdbtn` is given the value `0` and `ht_cm_rdbtn` is given the value `1`.

```
radValue = tk.IntVar() # integer variable for value of
                       # selected radio button
                       # default value is 1
```

A radio button is created using the constructor of the class `Radiobutton` in the themed tkinter package.

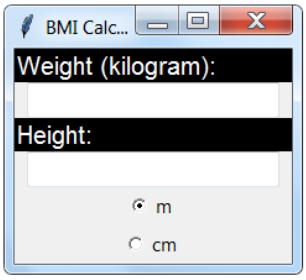
The two radio buttons `ht_m_rdbtn` and `ht_cm_rdbtn` are first created and then placed into their containers.

```
ht_m_rdbtn = ttk.Radiobutton(radioFrame, text='m', variable=radValue,
value=0)
ht_cm_rdbtn = ttk.Radiobutton(radioFrame, text='cm', variable=radValue,
value=1)

ht_m_rdbtn.pack()
ht_cm_rdbtn.pack()
```

To set the initial selected radio button to `ht_m_rdbtn`, simply set `radValue` to `0` as the value of `ht_m_rdbtn` is `0`.

```
radValue.set(0)
```

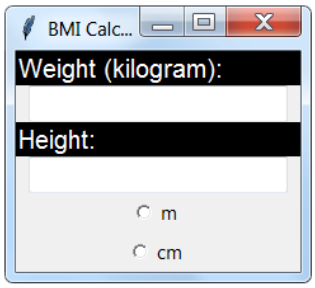


If `radValue` is set to a number that is not a value for radio buttons using `radValue` as variable, then none of the radio buttons is selected.

For example,

```
radValue.set(-1)
```

produces this GUI:



- button

  A button allows a user to invoke a call to a function by clicking on it.

  A button is created using the constructor of the class `Button` in the themed tkinter package.

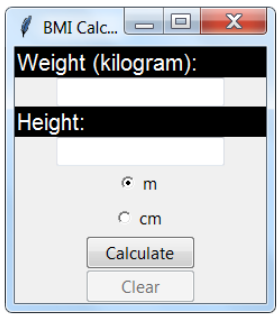
  A `Button` object can be configured at creation time or after being created using the method `configure` or the method `config`. It is placed into its container using the method `pack`.

  The code fragment creates two buttons, `calc_btn` showing the text `Calculate` and `clear_btn` showing the text `clear`. The `clear_btn` button is disabled, that is, it is not clickable. Both buttons are children of `actionFrame`, and are placed into a non-top level container, `actionFrame`.

```
actionFrame = ttk.Frame(dataFrame)
actionFrame.pack()

calc_btn = ttk.Button(actionFrame, text="Calculate")
calc_btn.pack()

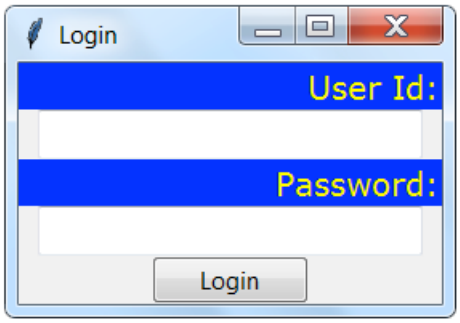
clear_btn = ttk.Button(actionFrame, text="Clear")
clear_btn.config(state = tk.DISABLED) # prevent clicking
clear_btn.pack()
```



We have not associated the buttons with the functions that should execute when the GUI application receives a notification that the event has occurred, that is, the button has been clicked. Thus, clicking on the buttons now will not cause the GUI to respond.

Activity 5

Create and place a button labelled Login into `inputFrame`, for a GUI shown below.



- scrolled text

  A scrolled text is useful for multi-line input or output.

  A scrolled text is created using the constructor of the class `ScrolledText` in the tkinter `scrolledtext` package.

  A `ScrolledText` object can be configured at creation time or after being created using the method `configure` or the method `config`. It is placed into its container using the method `pack`.

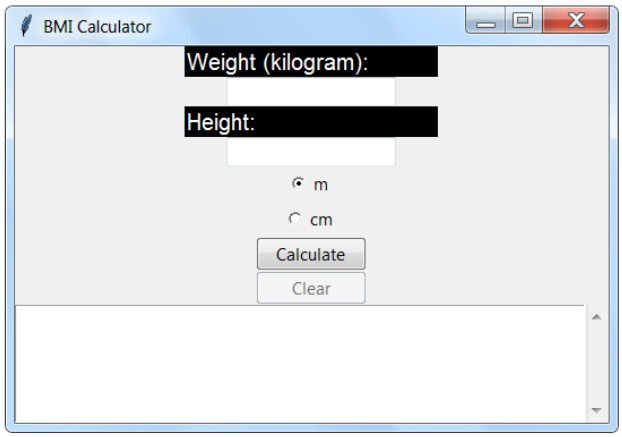
  The code fragment creates scrolled text, `scrol_stxt` with `outputFrame` as its parent. The scrolled text will be used during event handling when event handling is implemented in section 1.4. Whenever `calc_btn` is clicked, the function handling the event will add another line of output into `scrol_stxt`.

  The `scrol_stxt` scrolled text is disabled, that is, it is not editable. This disallows adding, changing or deleting text in `scrol_stxt`.

```
from tkinter import scrolledtext

outputFrame = ttk.Frame(win)
outputFrame.pack()
```

```
scrol_stxt = scrolledtext.ScrolledText(outputFrame, width=50, height=5,
wrap=tk.WORD)
scrol_stxt.config(state = tk.DISABLED) # prevent editing
scrol_stxt.pack()
```



Read

https://docs.python.org/3/library/tkinter.ttk.html#ttk-widgets

1.3 Layout Managers

A layout manager helps to arrange widgets in a container. The layout manager or geometry manager that we used thus far to place the widgets into their container is the pack geometry manager.

tkinter provides 3 types of geometry managers:

- pack

  The pack geometry manager is useful when there are few widgets and the layout of GUI is simple.

  The method `pack` can accept a number of arguments such as:

  - `side`

    The `side` argument allows us to position widgets from the left, the right, the top or the bottom. For example, to place the radio buttons beside each other, we apply either the value `tk.LEFT` or `tk.RIGHT`.

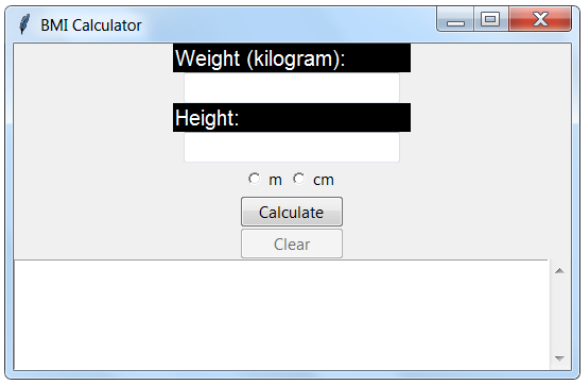
    Either

```
ht_m_rdbtn.pack(side = tk.LEFT)      # place next left position
ht_cm_rdbtn.pack(side = tk.LEFT)     # of container
```

or

```
ht_cm_rdbtn.pack(side = tk.RIGHT)    # place next right position
ht_m_rdbtn.pack(side = tk.RIGHT)     # of container
```

produces:



The default value for `side` is `tk.TOP`.

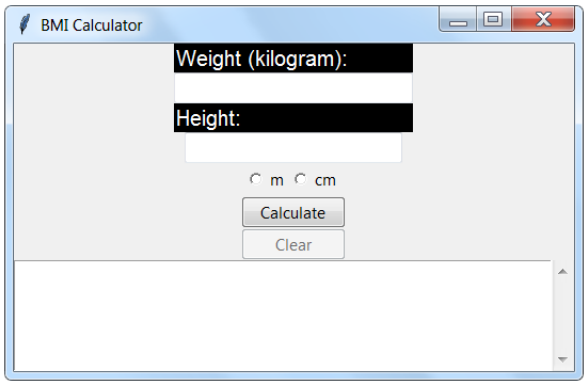
- `fill`

The `fill` argument allows us to extend widget to its location in the container. For example, to extend the text fields, we apply either of these values: `tk.X`, `tk.Y` or `tk.BOTH`.

In this code fragment:

```
wt_Ety.pack(fill = tk.X)
ht_Ety.pack(fill = tk.Y)
```

the `Entry` objects `wt_Ety` and `ht_Ety` are extended to their locations in the containers in in the directions `tk.X` (horizontal) and `tk.Y` (vertical) respectively.

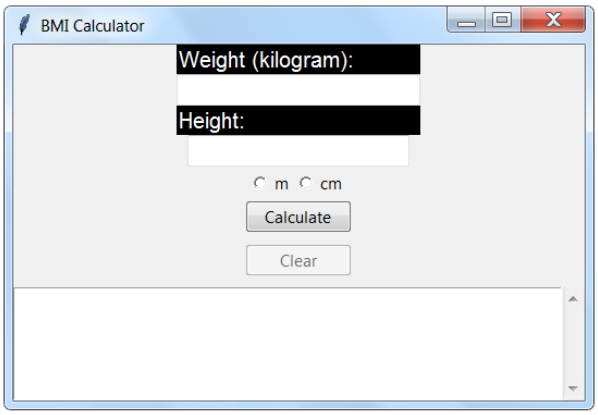


- `padx`, `pady`, `ipadx`, `ipady`

Padding using `padx` and `pady` provide extra external spaces around the widget whereas `ipadx` and `ipady` provide extra internal spaces.

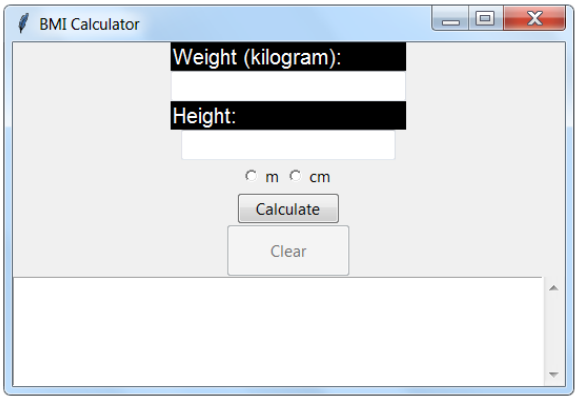
Effect of external padding:

```
clear_btn.pack(padx = 10, pady = 10)
```



Effect of internal padding:

```
clear_btn.pack(ipadx = 10, ipady = 10)
```

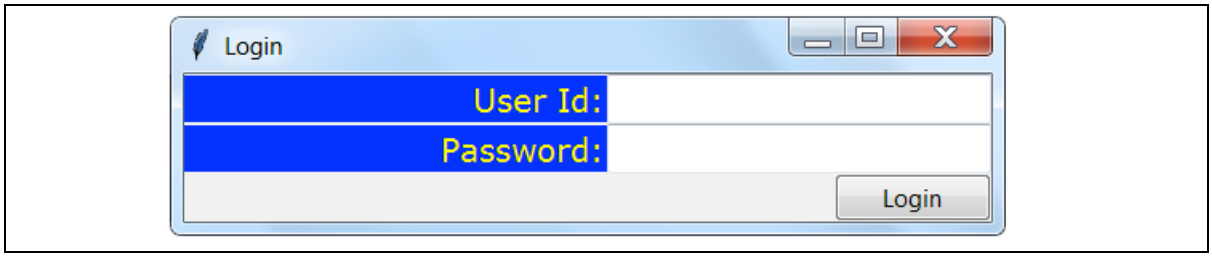


Read

https://docs.python.org/3/library/tkinter.html#the-packer

Activity 6

Use only the pack geometry manager to place the widgets as shown below. You will need non-top level containers.



- grid

  The grid geometry manager is the most commonly used geometry manager, and it is useful for a GUI with a more complicated layout.

  The grid geometry manager divides a container into rows and columns, and widgets can be positioned in their container by specifying the column number and row number.

  Both column and row numbers start with 0 for each container, as shown in Figure 5.4. By default, the widgets are placed in the center of the cell they are put into.

  Each container in Figure 5.4 uses the grid geometry manager:

  - The root widget `win` has 2 rows and 1 column as only 2 rows and 1 column are used to contain the non-top level containers `dataFrame` and `outputFrame`.

    ```
    dataFrame = ttk.Frame(win)
    outputFrame = ttk.Frame(win)

    dataFrame.grid(column=0, row=0)
    outputFrame.grid(column=0, row=1)
    ```

  - `dataFrame` has 4 rows and 2 columns.

    ```
    wt_lbl = ttk.Label(dataFrame, text="Weight (kilogram):" )
    wt_Ety = ttk.Entry(dataFrame, width=18, textvariable=weight)
    ht_lbl = ttk.Label(dataFrame, text="Height:" )
    ht_Ety = ttk.Entry(dataFrame, width=18, textvariable=height)
    radioFrame = ttk.Frame(dataFrame)
    actionFrame = ttk.Frame(dataFrame)
    ```

    Row 0 contains the label and text field for weight.

    ```
    wt_lbl.grid(column=0, row=0)
    wt_Ety.grid(column=1, row=0)
    ```

    Row 1 contains the label and text field for weight.

```
ht_lbl.grid(column=0, row=1)
ht_Ety.grid(column=1, row=1)
```

Row 2 and column 1 contains `radioFrame`, another non-top level container.

```
radioFrame.grid(column=1, row=2)
```

Row 3 and column 1 contains `radioFrame`, another non-top level container.

```
actionFrame.grid(column=1, row=3)
```

- `radioFrame` has 1 row and 2 columns.

```
ht_m_rdbtn = tk.Radiobutton(radioFrame, text = 'm', variable=radValue,
value=0)
ht_cm_rdbtn = tk.Radiobutton(radioFrame, text = 'cm',
variable=radValue,value=1)

ht_m_rdbtn.grid(column=0, row=0)
ht_cm_rdbtn.grid(column=1, row=0)
```

- `actionFrame` has 1 row and 2 columns.

```
calc_btn = ttk.Button(actionFrame, text="Calculate")
clear_btn = ttk.Button(actionFrame, text="Clear")

calc_btn.grid(column=0, row=0)
clear_btn.grid(column=1, row=0)
```

- `outputFrame` has 1 row and 1 column.

```
scrol_stxt = scrolledtext.ScrolledText(outputFrame, width=50,
height=5, wrap=tk.WORD)

scrol_stxt.grid(column=0, row=0)
```

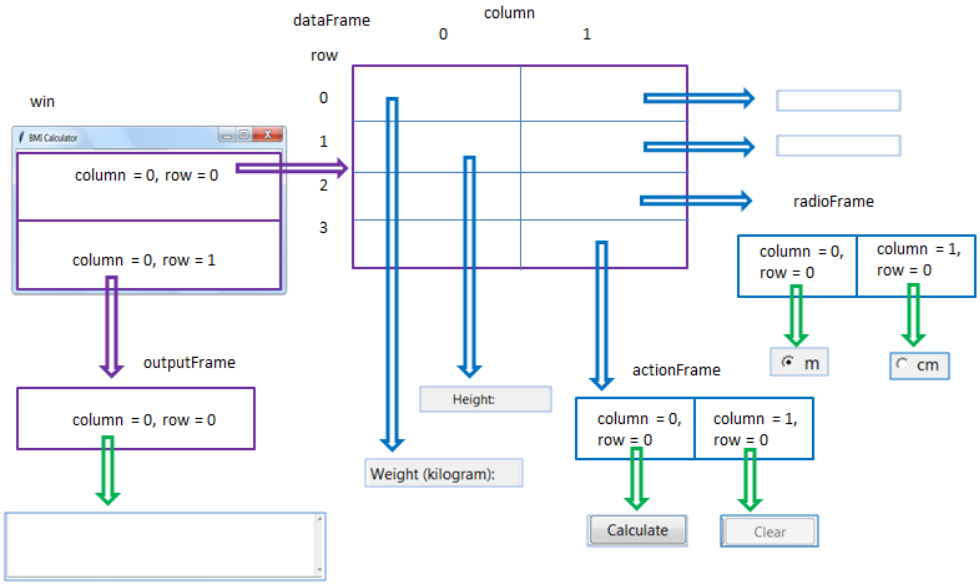
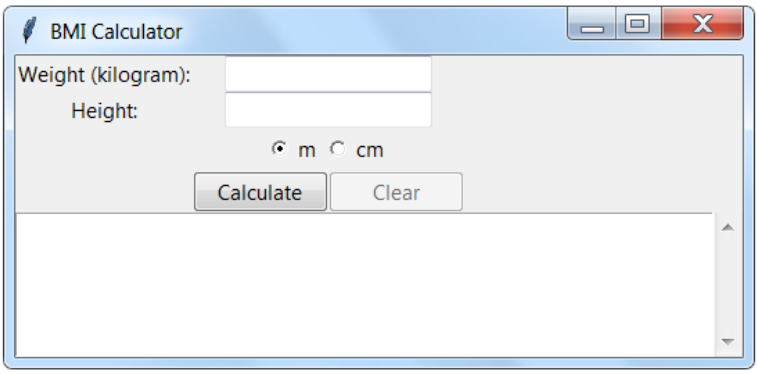


Figure 5.4 Row and column positions in GUI using grid geometry layout

Unused rows and columns do not show up in the GUI. For example, if row 10 is used instead of row 1 to place `outputFrame` in `win`, the GUI will have no change in appearance as rows 1 to 9 are unused, and so these rows do not show up.

```
outputFrame.grid(column=0, row=10)
```



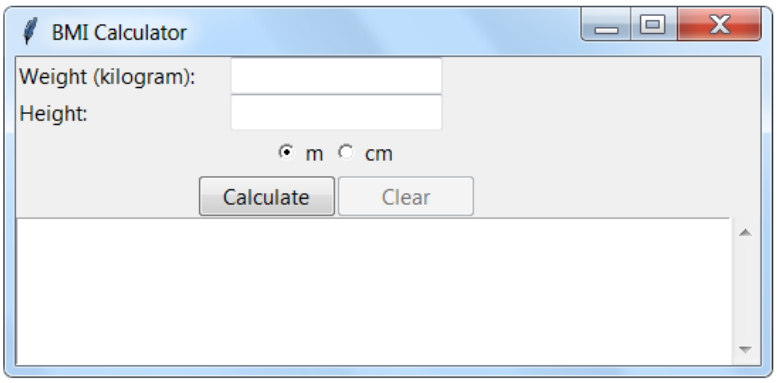
If no column and no row numbers are specified, the grid geometry manager will position the widget into column 0 of the next unused row.

The method `grid` can accept a number of arguments such as:

- `sticky`

The `sticky` argument allows a widget to align to the border of its cell. For example, to stick the label for height to the west or left of the cell, we apply `tk.W`. To stick the label to all 4 corners of its cell, use `tk.NSEW`.

```
ht_lbl.grid(column=0, row=1, sticky = tk.NSEW)
```

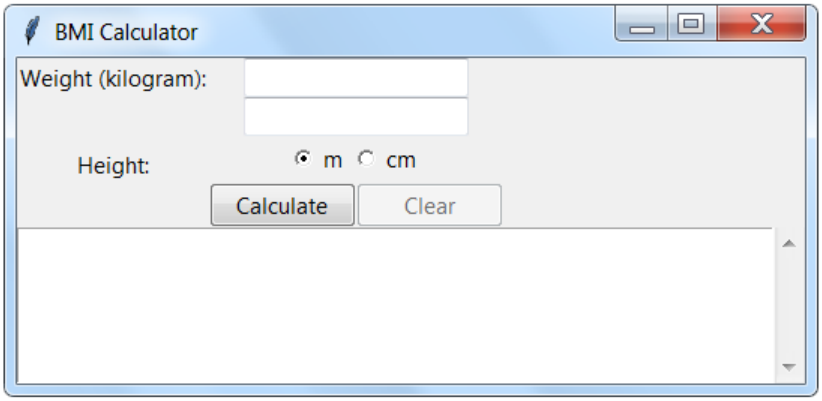


- `columnspan` and `rowspan`

The `columnspan` and `rowspan` arguments allow a widget to occupy more than one cell.

For example, the label for height occupies 2 rows from column 0 and row 1, that is, the label occupies column 0 of rows 1 and 2.

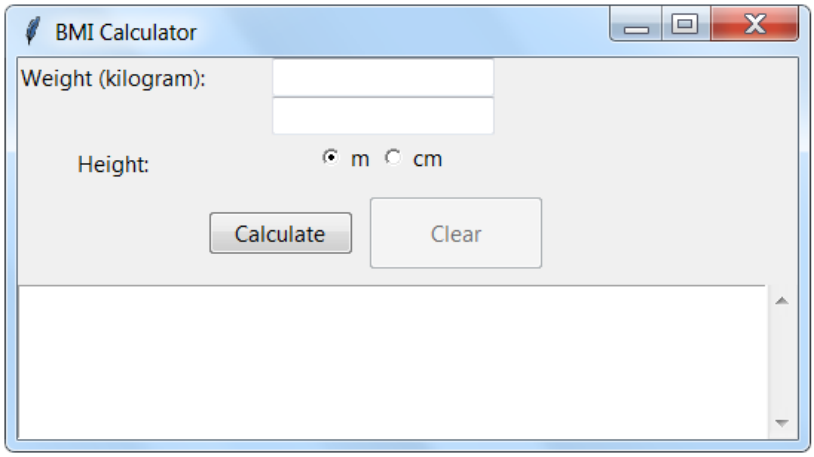
```
ht_lbl.grid(column=0, row=1, sticky = tk.S, rowspan = 2)
```



- `padx`, `pady`, `ipadx`, `ipady`

Padding using `padx` and `pady` provide extra external spaces around the widget whereas `ipadx` and `ipady` provide extra internal spaces work in the same manner as how these arguments are used in the method `pack`.

```
clear_btn.pack(padx = 10, pady = 10, ipadx = 10, ipady = 10)
```

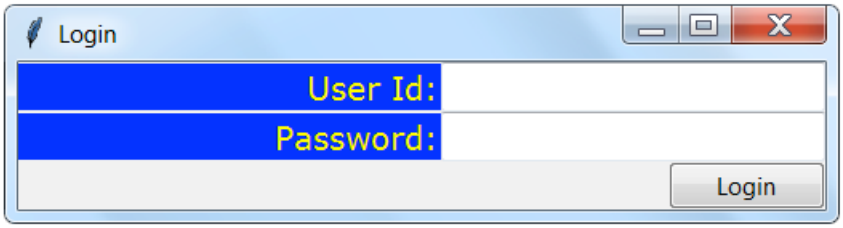


Read

https://www.tcl.tk/man/tcl/TkCmd/grid.htm

Activity 7

Use only the grid geometry manager to place the widgets as shown below.



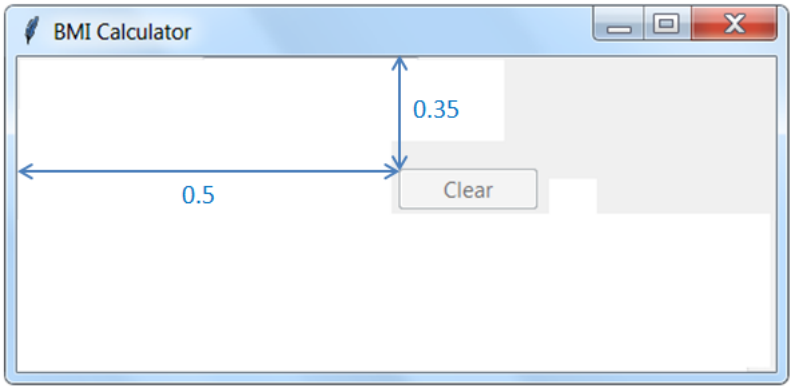
- place

  The place geometry manager gives the programmer the most control of the GUI layout as it allows us to specify the absolute or relative position of a widget in its container.

  However, the place geometry manager is not as popular as the programmer needs to compute the positions to place the widgets.

  In this example, the button `clear_btn` is placed into its container by specifying its relative position to `win`.

```
clear_btn = ttk.Button(win, text="Clear")
clear_btn.place(relx = 0.5, rely = 0.35)
```



1.4 Event Handling: Events and Callbacks

User actions on widgets generate events and a GUI application must respond to these events for the GUI to be interactive. Responding to events is also called event handling.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 493-499.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

During event handling, an event handling function executes to respond to the user action.

To perform event handling:

- Design the interaction

  Identify the user action and the widgets the user acts on.

  For example, the GUI BMI application responds with the following actions whenever a user clicks on button:

  - `calc_btn`

    The application performs the following actions:
    - compute the bmi with the values entered in the text fields for weight and height,
    - output the result by adding it to the scrolled text,
    - clear the text fields and
    - enable the button `clear_btn`.

- `clear_btn`

  The application performs the following actions:
  - clear the scrolled text and
  - disable the button `clear_btn`.

- code the functions to respond to the events.

  Name and implement the functions that are executed when the widgets generate events.

  For example, the name of the function that handles the button-clicked event generated by the button `calc_btn` is `calcBMI`. The function calculates the BMI.

  `calcBMI` needs to perform the following steps:

  - Read the string variables associated with the text fields and convert the values to floating point numbers:

    ```
    h = float(height.get())
    w = float(weight.get())
    ```

  - Check the value of the integer variables associated with the radio buttons to determine whether the radio button for height in centimetres is selected.

    If so, the height should be converted to metres.

    ```
    if radValue.get() == 1:
        h = h/100
    ```

  - Compute the BMI

    ```
    result = w/(h*h)
    ```

  - Output to the scrolled text.

    Since the scrolled text has been disabled to prevent user from changing its content, the function must
    - enable the scrolled text
    - write the output
    - disable the scrolled text.

    ```
    scrol_stxt.config(state = tk.NORMAL)
    scrol_stxt.insert('end', \
                      'Weight = {:.1f}kg, Height = {:.2f}m,' + \
                      'BMI = {:.2f}\n'.\
                      format(w, h, result))
    scrol_stxt.see('end')
    scrol_stxt.config(state = tk.DISABLED)
    ```

- clear the text fields by setting their string variables to the empty string.

```
height.set("")
weight.set("")
```

The complete function is shown here:

```
def calcBMI():
    h = float(height.get())
    w = float(weight.get())

    if radValue.get() == 1:
        h = h/100

    result = w/(h*h)

    scrol_stxt.config(state = tk.NORMAL)
    scrol_stxt.insert('end', \
                      'Weight = {:.1f}kg, Height = {:.2f}m,' + \
                      'BMI = {:.2f}\n'.\
                      format(w, h, result))
    scrol_stxt.see('end')
    scrol_stxt.config(state = tk.DISABLED)

    height.set("")
    weight.set("")

    clear_btn.config(state = tk.NORMAL)
```

We will name the function that handles the button-clicked event for the `clear_btn` button `clear`. It resets the GUI.

`clear` needs to perform the following steps:

- clear the scrolled text.

  Since the scrolled text has been disabled to prevent user from changing its content, the function must

  - enable the scrolled text
  - delete its content
  - disables the scrolled text

    ```
    scrol_stxt.config(state = tk.NORMAL)
    scrol_stxt.delete(1.0,tk.END)
    scrol_stxt.config(state = tk.DISABLED)
    ```

- disable the button `clear_btn`

  ```
  clear_btn.config(state = tk.DISABLED)
  ```

The complete function is shown here:

```
def clear():
    scrol_stxt.config(state = tk.NORMAL)
    scrol_stxt.delete(1.0,tk.END)
    scrol_stxt.config(state = tk.DISABLED)
    clear_btn.config(state = tk.DISABLED)
```

- associate the event-handling functions to the widgets

  Event handling functions in Python are also called **callback functions** as they are arguments to functions which associate them to widgets.

  When an action is performed on the widget, the widget generates an event and consequently, the event-handling function which the widget previously received as argument is called (back). Thus, the name callback is given to event-handling functions.

  An event-handling function can be associated to a widget in 2 ways.

  - As the configuration argument

    To associate a callback function to a button widget, we use the keyword `command` for the configuration argument.

    - At widget creation

      ```
      calc_btn = ttk.Button(actionFrame, text="Calculate",
      command=calcBMI)
      clear_btn = ttk.Button(win, text="Clear", command= clear)
      ```

    - At configuration using the method `config`.

      ```
      calc_btn.config(command=calcBMI)
      clear_btn.config(command= clear)
      ```

    Using the keyword `command` for buttons will associate the left button-clicked event on the button to the event-handling function. The callback does not happen if the right button is used to click on the button.

  - As an argument to the method `bind`

    Many widgets have the method `bind` to bind an event type and a callback function to themselves.

    For example, the code fragment binds the callback function `clear`, the events of left button-clicked (`<Button-1>`) and right button-clicked (`<Button-3>`) to the button `clear_btn`.

```
clear_btn.bind('<Button-1>', clear)
clear_btn.bind('<Button-3>', clear)
```

Using the method `bind` requires that the callback function `clear` accepts an argument for the event generated. Therefore, the header for the callback function `clear` must be changed as follows:

```
def clear(event):
```

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 499-503.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Having the event object allows the callback function `clear` to access details of the event that triggers its execution.

```
print('event =', event, event.widget.cget('text'))
if event.widget == clear_btn:
    print('Clear button is clicked!')
if event.num == 1:
    print('clearing with left button')
else:
    print('clearing with right button')
```

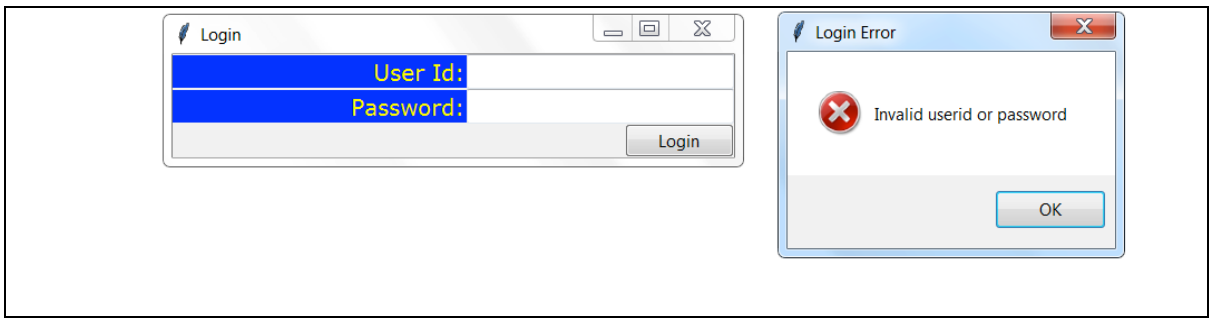
This code example produces the following output when the left button click on the button `clear_btn`, followed by a right button click on the same button:

```
event = <ButtonPress event state=Mod1 num=1 x=58 y=9> Clear
Clear button is clicked!
clearing with left button


event = <ButtonPress event state=Mod1 num=3 x=45 y=10> Clear
Clear button is clicked!
clearing with right button
```

Activity 8

Research on how to create an error message box and then perform event handling such that whenever the button labelled Login is left-clicked, an error message box shows up as shown below.



Read

https://docs.python.org/3/library/tkinter.html#bindings-and-events

1.5 Implementing GUI

Applying object-oriented programming is a good way to structure any application, including GUI application. Notice that tkinter itself is an object-oriented implementation. Each widget is created from a class which defines its attributes and behaviour.

Therefore, we restructure the GUI application in 3 classes and handle exceptions:

- User defined exception class

```
class InvalidDataError(Exception):
    '''Raised when data is out of range
    '''
```

- BMI calculation and range checks from section 2.3 Raising and Handling Exceptions in Applications, Study Unit 4, now implemented as a class.

```
from SU5.RangeError import InvalidDataError
class BMICalculator:

    @classmethod
    def heightInRange(cls, height):
        if height < 0.5:
            raise InvalidDataError('Height must be at least 0.5 meters')
        if height > 2.6:
            raise InvalidDataError('Height must not be more than 2.6
meters')
        return True

    @classmethod
```

```
    def weightInRange(cls, weight):
        if weight < 10:
            raise InvalidDataError('Weight must be at least 10 kilograms')
        if weight > 150:
            raise InvalidDataError('Weight must not be more than 150
kilograms')
        return True

    @classmethod
    def bmi(cls, height, weight):
        if cls.heightInRange(height) and cls.weightInRange(weight):
            return weight/(height * height)
```

- GUI creation and event handling, coded as a class

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from SU5.bmiCalculator import BMICalculator

class BMIGui:

    def __init__(self):
        self.win = tk.Tk()
        self.win.resizable(False, False)

        # Add a title
        self.win.title("BMI Calculator")
        self.create_widgets()
        self.win.mainloop()

    def create_widgets(self):
        dataFrame = ttk.Frame(self.win)
        dataFrame.grid(column=0, row=0)

        wt_lbl = ttk.Label(dataFrame, text="Weight (kilogram):")
        wt_lbl.grid(column=0, row=0, sticky='W')

        self.weight = tk.StringVar()
        self.wt_Ety = ttk.Entry(dataFrame, width=18,
textvariable=self.weight)
        self.wt_Ety.grid(column=1, row=0,  columnspan=2)

        ht_lbl = ttk.Label(dataFrame, text="Height:")
        ht_lbl.grid(column=0, row=1, sticky='W')

        self.height = tk.StringVar()
        self.ht_Ety = ttk.Entry(dataFrame, width=18,
textvariable=self.height)
        self.ht_Ety.grid(column=1, row=1,  columnspan=2)

        self.radValue = tk.IntVar()
        self.radValue.set(0)

        radioFrame = ttk.Frame(dataFrame)
        radioFrame.grid(column=1, row=2)
```

```
        self.ht_m_rdbtn = ttk.Radiobutton(radioFrame, text = 'm',
variable=self.radValue,
                                          value=0)
        self.ht_cm_rdbtn = ttk.Radiobutton(radioFrame, text = 'cm',
variable=self.radValue,
                                          value=1)
        self.ht_m_rdbtn.grid(column=0, row=0, sticky=tk.W)
        self.ht_cm_rdbtn.grid(column=1, row=0, sticky=tk.W)


        actionFrame = ttk.Frame(dataFrame)
        actionFrame.grid(column=1, row=3, padx=8, pady=4)

        self.calc_btn = ttk.Button(actionFrame, text="Calculate")
        self.calc_btn.bind('<Button-1>', self.calcBMI)

        self.calc_btn.pack(side = tk.LEFT)

        self.clear_btn = ttk.Button(actionFrame, text="Clear")
        self.clear_btn.bind('<Button-1>', self.clear)

        self.clear_btn.config(state = tk.DISABLED)
        self.clear_btn.pack(side = tk.LEFT)

        outputFrame = ttk.Frame(self.win)
        outputFrame.grid(column=0, row=2, padx=8, pady=4, columnspan=2)

        scrol_w  = 50
        scrol_h  =  5
        self.scrol_stxt = scrolledtext.ScrolledText(outputFrame,
width=scrol_w, height=scrol_h, wrap=tk.WORD)
        self.scrol_stxt.grid(column=0, row=0, sticky='WE', columnspan=2)
        self.scrol_stxt.config(state = tk.DISABLED)
        self.wt_Ety.focus()

    def calcBMI(self, event):
        self.scrol_stxt.config(state = tk.NORMAL)
        try:
            h = float(self.height.get())
            if self.radValue.get() == 1:
                h = h/100
            w = float(self.weight.get())
            result = BMICalculator.bmi(h, w)
        except Exception as e:
            self.scrol_stxt.insert('end', str(e) + '\n')
        else:
            self.scrol_stxt.insert('end', \
                                   'Weight = {:.1f}kg, Height = {:.2f}m,
BMI = {:.2f}\n'.\
                                   format(w, h, result))
        finally:
            self.scrol_stxt.see('end')
            self.scrol_stxt.config(state = tk.DISABLED)
            self.height.set("")
            self.weight.set("")
            self.wt_Ety.focus()
            self.clear_btn.config(state = tk.NORMAL)

    def clear(self, event):
```

```
        self.scrol_stxt.config(state = tk.NORMAL)
        print(self.scrol_stxt.get("1.0",tk.END))
        self.scrol_stxt.delete(1.0,tk.END)
        self.scrol_stxt.config(state = tk.DISABLED)
        self.wt_Ety.focus()
                    self.clear_btn.config(state = tk.DISABLED)
```

Activity 9

Apply oriented –oriented programming and recode the solution to Activity 8 and incorporate the following changes:

a) Implement a class userPasswordData
   This class maintains a dictionary of userid and password pairs. Include the following methods:
   - __init__ : creates an empty dictionary
   - check : accepts a userid and password pair and returns True if the pair is an entry in the dictionary. Return False otherwise
   - add: accepts a userid and password pair and adds to dictionary if the userid does not exist in the dictionary yet. Returns True if add is successful and False otherwise.
   - update: accepts a userid and password pair and update the password if the userid exists in the dictionary. Returns True if update is successful and False otherwise.

b) Implement a class GUI that show up the sane GUI as Activity 8. Handle events by calling the appropriate methods in the class userPasswordData.

Read

Goldwasser, M. H., & Letscher, D. (2014).
Object-Oriented Programming in Python [pdf version], pages 512-519.
Retrieved from *http://cs.slu.edu/~goldwamh/oopp/*

Applying object-oriented programming to the implementation of GUI applications allows us to separate the various programming concerns. As seen in Activity 9, the class userPasswordData is concerned with the data aspect whereas the class GUI is concerned with the look and user interaction afforded by the GUI.

A popular framework for GUI application is MVC, where the GUI application is roughly divided into 3 major concerns: the model (data), the view (look and interaction) and the controller that sits between the model and the view.

There are different opinions about what behaviour goes into the model, view and controller such as whether the view has direct access to the methods of the model. For this study unit, we have adopted a strict communication protocol whereby all communication between the model and the view is via the controller.

Some implications of the MVC framework are as follows:

- There can be more than one view for the same model.

  In our framework, when the model changes, the controller invokes an update on each of the views, so the change is observed by all views of the model.

- Any interaction between the model and the view is indirect and is via the controller.

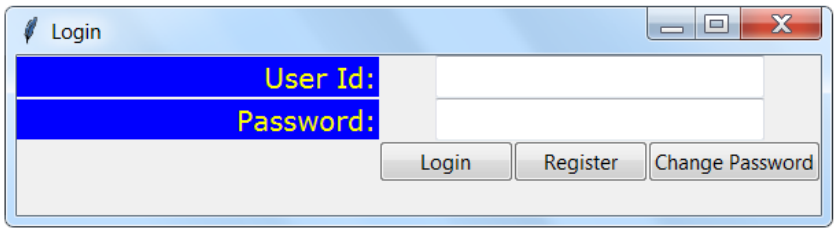
  In our framework, a view gets data to display from the model when the controller invokes an update on the view, and sends it the updated data.

  A view can sends input to the model via the controller. The callback of the view can invoke a method in the controller that then updates the model and the views. Therefore, once a view changes the model (indirectly), all views will observe the change.

- In our framework, the controller applies object composition, and has a reference to the model and all the views.

- In our framework, each view has a reference to the controller.

The extended solution coded in the MVC framework for the below GUI is shown here:



Model:

```
class userPasswordData(object):
    def __init__(self):
        self.dic = {}

    def check(self, user, pwd):
        if user not in self.dic:
            return False
        if self.dic[user] != pwd:
            return False
```

```
        return True

    def add (self, key, value):
        if key not in self.dic:
            self.dic[key] = value
            return True
        return False

    def update (self, key, value):
        if key not in self.dic:
            return False
        self.dic[key] = value
        return True

    def getData(self):              # this method returns data of the model
        return self.dic.keys()      # Assume that keys or userids are the only data
                                    # we allow to display on GUI
```

Controller:

```
class Controller:                   # new Controller class that sits in between
    def __init__(self, model):
        self.model = model          # reference a data model
        self.views = []             # reference a collection of views

    def addViews(self, view):       # the controller can add view to the collection
        self.views.append(view)     # of views or GUIs.

    def update(self):               # each view can get an updated data, a list
        for v in self.views:        # of userIds, if this list is to be displayed
            v.update(self.model.getData())

    def check(self, user, pwd):     # controller calls method of model for view
        return self.model.check(user, pwd)

    def register(self, user, pwd): # controller calls method of model for view
        if self.model.add(user, pwd):
            self.update()           # a call to update all the views
            return True
        return False

    def updatePwd(self, user, pwd): # controller calls method of model for view
        if self.model.update(user, pwd):
            self.update()           # a call to update all the views
            return True
        return False C
```

View:

```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

class GUI:   # one of the possible views

    def update(self, data):
```

```
        pass      #implement the changes to GUI with the update data of model

    def showResult(self, event):
        if not self.controller.check(self.userId.get(), self.password.get()):
            messagebox.showerror('Login Error', 'Invalid userid or password')
        else:
            messagebox.showinfo('Login Status', 'Valid userid and password')

    def register(self, event):
        self.controller.register(self.userId.get(), self.password.get())

    def updatePwd(self, event):
        self.controller.updatePwd(self.userId.get(), self.password.get())

    def __init__(self, contrl):
        self.controller = contrl      # view has a reference to controller
        self.inputFrame = tk.Tk()
        self.inputFrame.title("Login")
        self.create_widgets()
        self.inputFrame.mainloop()

    def create_widgets(self):
        uid_lbl = ttk.Label(self.inputFrame, text="User Id:", width = 20, \
                            font=("Verdana", 12),
                            justify=tk.RIGHT, anchor=tk.E)

        uid_lbl.configure(foreground = 'yellow',  background = 'blue')
        uid_lbl.grid(column = 0, row = 0)

        self.userId = tk.StringVar()

        self.uid_ety = ttk.Entry(self.inputFrame, width=18,
textvariable=self.userId)
        self.uid_ety.configure(foreground = 'blue', font=("Verdana", 12))
        self.uid_ety.grid(column = 1, row = 0)

        pwd_lbl = ttk.Label(self.inputFrame, text="Password:", width = 20, \
                            font=("Verdana", 12),
                            justify=tk.RIGHT, anchor=tk.E)
        pwd_lbl.configure(foreground = 'yellow',  background = 'blue')
        pwd_lbl.grid(column = 0, row = 1)

        self.password = tk.StringVar()
        self.pwd_ety = ttk.Entry(self.inputFrame, width=18,
textvariable=self.password, show = '*')
        self.pwd_ety.configure(foreground = 'blue', font=("Verdana", 12))
        self.pwd_ety.grid(column = 1, row = 1)

        actionFrame = ttk.Frame(self.inputFrame)
        actionFrame.grid(column = 1, row = 2)
        login_btn = ttk.Button(actionFrame, text="Login")
        login_btn.bind('<Button-1>', self.showResult)
        login_btn.pack(side = tk.LEFT)

        add_btn = ttk.Button(actionFrame, text="Register")
        add_btn.bind('<Button-1>', self.register)
        add_btn.pack(side = tk.LEFT)
```

```
        update_btn = ttk.Button(actionFrame, text="Change Password")
        update_btn.bind('<Button-1>', self.updatePwd)
        update_btn.pack(side = tk.LEFT)
```

This code fragment creates a model, a controller with the model and a view with the controller, and gets the view running in a loop. It also adds the view to the controller:

```
m = userPasswordData()
c = Controller(m)
v = GUI(c)
c.addViews(v)
```

Summary

A graphical user interface (GUI) consists of widgets, layout managers and event handlers.

A typical GUI includes widgets for data entry, output display, information display, user action or event generation and containment. Some common widgets are text fields, labels and buttons. Widgets are first created, configured and then placed into containers using layout managers.

Layout managers arrange widgets in containers. The layout managers or geometry managers available in standard package of tkinter are pack, grid and place. The pack geometry manager allows widgets to be placed in positions such as `LEFT`, `RIGHT`, `TOP` and `BOTTOM` of their container. The grid geometry manager divides a container to rows and columns, and places widgets into the various cells of the container. The place geometry uses either relative or absolute positions to specify the location in the container to place the widgets.

User actions on widgets generate events and a GUI application responds to these events via callbacks or functions that handle events. The callback functions must be associated to the widgets that generate the events using the method `bind` or the configuration argument `command`.

Object-oriented programming help modularize the implementation of GUI application, in particular, the framework, MVC is commonly applied.

References

Goldwasser, M. H., & Letscher, D. (2014). *Object-Oriented Programming in Python [pdf version]. Retrieved from http://cs.slu.edu/~goldwamh/oopp/*

https://docs.python.org/3/library/tk.html

https://docs.python.org/3/library/tkinter.ttk.html#ttk-widgets

https://docs.python.org/3/library/tkinter.html#the-packer

https://www.tcl.tk/man/tcl/TkCmd/grid.htm

https://docs.python.org/3/library/tkinter.html#bindings-and-events

View the answers at the end of this study unit.

Formative Assessment

1. Which is not a purpose of widgets?

A) Information display
@ Incorrect. This is a purpose of widgets. Refer to Section 1.1 Introduction to GUI and Event Driven Programming, Study Unit 5.

*B) Handle event
@ Correct. This is not a purpose of widgets. Refer to Section 1.1 Introduction to GUI and Event Driven Programming, Study Unit 5.

C) User Input
@ Incorrect. This is a purpose of widgets. Refer to Section 1.1 Introduction to GUI and Event Driven Programming, Study Unit 5.

D) Output display
@ Incorrect. This is a purpose of widgets. Refer to Section 1.1 Introduction to GUI and Event Driven Programming, Study Unit 5.

2. Why should non-top level containers be introduced in a GUI?

A) They are associated with a tcl interpreter.
@ Incorrect. Only the root (container) widget is associated with a tcl interpreter. Refer to Section 1.2 Widgets, Study Unit 5.

B) They layout the widgets.
@ Incorrect. Layout managers layout the widgets. Refer to Section 1.2 Widgets, Study Unit 5.

*C) They group widgets so that the widgets can be better managed,
@ Correct. Widgets in a group can be configured in a loop and laid out in their container. Their container can then be placed at a specified position in top level container. Refer to Section 1.2 Widgets, Study Unit 5.

D) They group widgets so that the widgets can show up in the GUI.
@ Incorrect. A widget can be placed in the top level container to show up in the GUI. Refer to Section 1.2 Widgets, Study Unit 5.

3. Which class creates a text field in tkinter?

A) `StringVar`
@ Incorrect. This class creates a string variable to store the string for the text field. Refer to Section 1.2 Widgets, Study Unit 5

B) `Data`

@ Incorrect. This is not a class for text field implemented in tkinter. Refer to Section 1.2 Widgets, Study Unit 5

*C) `Entry`
@ Correct. This is a class for text field implemented in tkinter. Refer to Section 1.2 Widgets, Study Unit 5

D) `Textfield`
@ Incorrect. This is not a class for text field implemented in tkinter. Refer to Section 1.2 Widgets, Study Unit 5

4. How are radio buttons put into a group?

A) By applying the method `group` on the radio buttons
@ Incorrect. This is not correct solution for tkinter. Refer to Section 1.2 Widgets, Study Unit 5

B) By ending the names of the radio buttons with the same suffix `rdbtn`.
@ Incorrect. This is not correct solution for tkinter. Refer to Section 1.2 Widgets, Study Unit 5

C) By declaring only one interger variable
@ Incorrect. This solution is incomplete for tkinter. Refer to Section 1.2 Widgets, Study Unit 5

*D) By ensuring that they all use the same integer variable
@Correct. This is the correct solution for tkinter. Refer to Section 1.2 Widgets, Study Unit 5

5. Give the correct difference(s) between the pack and grid geometry manager?

A) pack will place one widget at a time into a container whether grid allows more than one widget to be simultaneously placed into a container.
@ Incorrect. Only one widget can be placed into a container at any time. Refer to Section 1.3 Layout Managers, Study Unit 5.

*B) pack allows 4 different positions of the container to be specified at a time whereas grid allows column and row of the container to be specified.
@ Correct. This is a correct difference. Refer to Section 1.3 Layout Managers, Study Unit 5.

C) grid can place non-top level containers into their parent but pack cannot.
@ Incorrect. This is incorrect. Both grip and pack can place non-top level containers into their parent. Refer to Section 1.3 Layout Managers, Study Unit 5.

D) All of the above are correct differences.
@ Incorrect. Only one difference is correct. Refer to Section 1.3 Layout Managers, Study Unit 5.

6. Why are event handlers also called callback functions?

i. They are arguments to some functions
ii. They receive other functions as arguments
iii. They get executed only when the GUI application calls the function
iv. They will always get executed after some time

*A) i. and iii.
@ Correct. These are two properties of event handler. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

B) i. and iv.
@ Incorrect. These are not two properties of event handler. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

C) ii. and iii.
@ Incorrect. These are not two properties of event handler. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

D) ii. and iv.
@ Incorrect. These are not two properties of event handler. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

7. What are true about using the configuration argument `command` and the method `bind` to associate an event handler with a button?

i. Using the configuration argument `command` requires that the callback function has an `event` formal parameter
ii. Using the method `bind` requires that the callback function has an `event` formal parameter
iii. Using the configuration argument `command` allows different types of events to associate to the callback function
iv. Using the method `bind` allows different types of events to associate to the callback function

A) i. and iii only
@ Incorrect. The statements are not correct. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

*B) ii. and iv. only
@ Correct. The statements are correct. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5.

C) i., iii. and iv. only
@ Incorrect. The statements are not correct. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5

D) All statements are true
@ Incorrect. The statements are not correct. Refer to Section 1.4 Event Handling: Events and Callbacks, Study Unit 5

8. What is/are benefits of applying object-oriented programming to a GUI application?

A) Modularity or separation of concerns only
@ Incorrect. The statement is not correct. Refer to Section 1.5 Implementing GUI, Study Unit 5.

B) Ease of maintenance only
@ Incorrect. The statement is not correct. Refer to Section 1.5 Implementing GUI, Study Unit 5.

C) Abstraction and information hiding only
@ Incorrect. The statement is not correct. Refer to Section 1.5 Implementing GUI, Study Unit 5.

*D) All the above benefits
@ Correct. The statement is correct. Refer to Section 1.5 Implementing GUI, Study Unit 5.

.

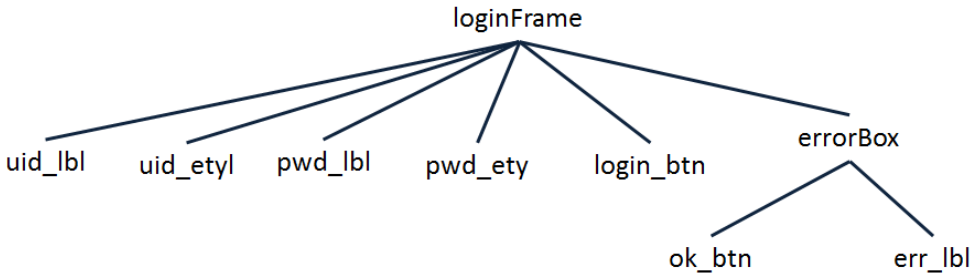
Solutions or Suggested Answers

Activity 1

2 text fields for user id and name for input
2 labels for user id and name labels for user information
1 login button for event generation
1 container for 5 above widgets
1 error dialog for output, acts as container for
    1 button labelled ok and
    1 label for error message

Activity 2

2 text fields for user id: uid_ety
    and name: pwd_ety
2 labels for user id: uid_lbl
    and name: pwd_lbl
1 login button: login_btn
1 container for 5 above widgets: loginFrame
1 error dialog for output, acts as container: errorBox
    1 button labelled ok: ok_btn
    1 label for error message: err_lbl



Activity 3

```
import tkinter as tk

inputFrame = tk.Tk()
inputFrame.title("Login")
inputFrame.mainloop()
```

Activity 4

```
uid_lbl = ttk.Label(inputFrame, text="User Id:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)

uid_lbl.configure(foreground = 'yellow',  background = 'blue')
uid_lbl.pack()
```

```
userId = tk.StringVar()

uid_ety = ttk.Entry(inputFrame, width=18, textvariable=userId)
uid_ety.configure(foreground = 'blue', font=("Verdana", 12))
uid_ety.pack()

pwd_lbl = ttk.Label(inputFrame, text="Password:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)
pwd_lbl.configure(foreground = 'yellow',  background = 'blue')
pwd_lbl.pack()

password = tk.StringVar()

pwd_ety = ttk.Entry(inputFrame, width=18, textvariable=password, show = '*')
pwd_ety.configure(foreground = 'blue', font=("Verdana", 12))

pwd_ety.pack()
```

Activity 5

```
login_btn = ttk.Button(inputFrame, text="Login")
login_btn.pack()
```

Activity 6

```
import tkinter as tk
from tkinter import ttk

inputFrame = tk.Tk()
inputFrame.title("Login")

userIdFrame = ttk.Frame(inputFrame)
userIdFrame.pack()
uid_lbl = ttk.Label(userIdFrame, text="User Id:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)
uid_lbl.configure(foreground = 'yellow',  background = 'blue')
uid_lbl.pack(side = tk.LEFT)

userId = tk.StringVar()
uid_ety = ttk.Entry(userIdFrame, width=18, textvariable=userId)
uid_ety.configure(foreground = 'blue', font=("Verdana", 12))
uid_ety.pack(side = tk.LEFT)

pwdFrame = ttk.Frame(inputFrame)
pwdFrame.pack()
pwd_lbl = ttk.Label(pwdFrame, text="Password:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)
pwd_lbl.configure(foreground = 'yellow',  background = 'blue')
pwd_lbl.pack(side = tk.LEFT)

password = tk.StringVar()
pwd_ety = ttk.Entry(pwdFrame, width=18, textvariable=password, show = '*')
pwd_ety.configure(foreground = 'blue', font=("Verdana", 12))
```

```
pwd_ety.pack(side = tk.LEFT)

login_btn = ttk.Button(inputFrame, text="Login")
login_btn.pack(side = tk.RIGHT)
inputFrame.mainloop()
```

Activity 7

```
import tkinter as tk
from tkinter import ttk

inputFrame = tk.Tk()
inputFrame.title("Login")

uid_lbl = ttk.Label(inputFrame, text="User Id:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)
uid_lbl.configure(foreground = 'yellow',  background = 'blue')
uid_lbl.grid(column = 0, row = 0)

userId = tk.StringVar()
uid_ety = ttk.Entry(inputFrame, width=18, textvariable=userId)
uid_ety.configure(foreground = 'blue', font=("Verdana", 12))
uid_ety.grid(column = 1, row = 0)

pwd_lbl = ttk.Label(inputFrame, text="Password:", width = 20, \
                    font=("Verdana", 12),
                    justify=tk.RIGHT, anchor=tk.E)
pwd_lbl.configure(foreground = 'yellow',  background = 'blue')
pwd_lbl.grid(column = 0, row = 1)

password = tk.StringVar()
pwd_ety = ttk.Entry(inputFrame, width=18, textvariable=password, show = '*')
pwd_ety.configure(foreground = 'blue', font=("Verdana", 12))
pwd_ety.grid(column = 1, row = 1)

login_btn = ttk.Button(inputFrame, text="Login")
login_btn.grid(column = 1, row = 2, sticky = tk.E)
inputFrame.mainloop()
```

Activity 8

```
def showResult(event):
    messagebox.showerror('Login Error', 'Invalid userid or password')


login_btn.bind('<Button-1>', showResult)
```

Activity 9

```
class userPasswordData(object):

    def __init__(self):
```

```
        self.dic = {}

    def getData(self):
        return self.dic.keys()

    def check(self, user, pwd):
        if user not in self.dic:
            return False
        if self.dic[user] != pwd:
            return False
        return True

    def add (self, key, value):
        if key not in self.dic:
            self.dic[key] = value
            return True
        return False

    def update (self, key, value):
        if key not in self.dic:
            return False
        self.dic[key] = value
        return True


import tkinter as tk
from tkinter import ttk
from tkinter import messagebox

class GUI:

    def showResult(self, event):
        if not self.dataModel.check(self.userId.get(), self.password.get()):
            messagebox.showerror('Login Error', 'Invalid userid or password')
        else:
            messagebox.showinfo('Login Status', 'Valid userid and password')

    def register(self, event):
        self.dataModel.register(self.userId.get(), self.password.get())

    def updatePwd(self, event):
        self.dataModel.updatePwd(self.userId.get(), self.password.get())


    def __init__(self, userIdPwdData):
        self.dataModel = userIdPwdData
        self.inputFrame = tk.Tk()
        self.inputFrame.title("Login")
        self.create_widgets()
        self.inputFrame.mainloop()

    def create_widgets(self):
        uid_lbl = ttk.Label(self.inputFrame, text="User Id:", width = 20, \
                            font=("Verdana", 12),
                            justify=tk.RIGHT, anchor=tk.E)

        uid_lbl.configure(foreground = 'yellow',  background = 'blue')
        uid_lbl.grid(column = 0, row = 0)
```

```
        self.userId = tk.StringVar()

        self.uid_ety = ttk.Entry(self.inputFrame, width=18,
textvariable=self.userId)
        self.uid_ety.configure(foreground = 'blue', font=("Verdana", 12))
        self.uid_ety.grid(column = 1, row = 0)

        pwd_lbl = ttk.Label(self.inputFrame, text="Password:", width = 20, \
                            font=("Verdana", 12),
                            justify=tk.RIGHT, anchor=tk.E)
        pwd_lbl.configure(foreground = 'yellow',  background = 'blue')
        pwd_lbl.grid(column = 0, row = 1)

        self.password = tk.StringVar()
        self.pwd_ety = ttk.Entry(self.inputFrame, width=18,
textvariable=self.password, show = '*')
        self.pwd_ety.configure(foreground = 'blue', font=("Verdana", 12))
        self.pwd_ety.grid(column = 1, row = 1)

        actionFrame = ttk.Frame(self.inputFrame)
        actionFrame.grid(column = 1, row = 2)
        login_btn = ttk.Button(actionFrame, text="Login")
        login_btn.bind('<Button-1>', self.showResult)
        login_btn.pack(side = tk.LEFT)

        add_btn = ttk.Button(actionFrame, text="Register")
        add_btn.bind('<Button-1>', self.register)
        add_btn.pack(side = tk.LEFT)


        update_btn = ttk.Button(actionFrame, text="Update")
        update_btn.bind('<Button-1>', self.updatePwd)
        update_btn.pack(side = tk.LEFT)
```

STUDY UNIT 6
OO DESIGN PRINCIPLES

Learning Outcomes

By the end of this unit, you should be able to:

1. List the 5 SOLID principles
2. Explain the benefits of applying SOLID principles
3. Relate design patterns to SOLID principles
4. Discuss the applications of SOLID principles in design patterns such as template method, strategy and MVC design patterns.
5. Evaluate a given OO design against SOLID principles.

Overview

Poor object-oriented program designs cause programs to be fragile, rigid and immobile. This study unit covers five software design principles, SOLID to overcome these problems.

The SOLID principles are first described and then illustrated using three design patterns: the template method pattern, the strategy pattern and the model-view-controller pattern.

This study unit does not cover any chapter of the textbook. However, as there are several online resources to read, it is estimated that the student will spend about 6 hours to read the resources, in conjunction with the study notes, to work out the activities and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered and to try out the review and project questions given at the end of each chapter in the textbook. It is advisable to use the study notes to guide the reading of each chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 SOLID Principles

1.1 Introduction to SOLID Principles

While object-oriented programming is easy to learn, it has also been demonstrated that poor object-oriented (OO) program designs causes programs to be

- Fragile

  Easily broken when a single change is made

- Rigid

  Hard to change as a change as cascading effects

- Immobile

  Hard to reuse as it is tightly coupled with the existing application

For example, poor OO design can cause the banana-gorilla-forest problem, roughly described as "you want a banana, but you also get the gorilla that is holding the banana as well as the entire forest with the gorilla".

Activity 1

Read up on the banana-gorilla-forest problem by Joe Armstrong and other discussions on it.

Is the banana-gorilla-forest problem a case of bad OO design or simply bad OO paradigm? Explain.

There are many sets of principles that have been proposed to help programmers write better OO solutions that are easy to extend and maintain. SOLID principles form one such set that was compiled by Robert Martin (commonly known as Uncle Bob) in the early 1990s. These OO design principles were then arranged by Michael Feathers, into the acronym SOLID, with each letter representing one of the five design principles.

Read

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

This study unit focuses on SOLID principles, each of which is listed and described below:

- Single Responsibility Principle (SRP)

  A class should have one and only one reason to change.

  – Robert Martin

  This principle suggests that a class should model one thing or is responsible for only one task so that the only reason for the class to change is a change in that thing or task.

  For example, the class Student should be only about a student, and the class Module should be about a module. A student may have a list of modules that he is currently enrolled in. While a student may have a collection of modules (references), the modules objects are described by the class Module rather than the class Student.

  The responsibility or the methods defined in a class depends on the data it carries. For example, it is the class Module's responsibility to compute the number of credit units a module is worth as it has the details of a module, and it is the class Student's responsibility to determine the number of modules a student needs to enroll in.

  When an aspect (data or behaviour) of the thing is changed, only the class needs to be changed. For example, if the computation of credit unit is changed, the change affects only the class Module.

  Program structured in this manner reduces coupling and increase cohesion.

  To determine whether an OO solution adheres to SRP, simply attempt to describe a class without using the conjunction "and".

Read

https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

Activity 2

Explain the benefit of applying SRP to OO solution.

- Open-Closed Principle (OCP)

  Software entities should be open for extension, but closed for modification.

  –Bertrand Meyer

  This principle suggests that existing classes must not be changed (closed for modification), but rather, new subclasses (or extension) should be implemented for the new variation of behaviour.

  When there is a variation of behaviour, the variation in behavior is specified in a new class which either extends an existing abstract class through inheritance and method overriding or implements a common interface.

  Abstraction and polymorphism are the key concepts for this principle.

  - Abstraction tells us what behaviour an object has, without specifying the how.

  - Polymorphism means that a program statement has many interpretations as the method bound in the statement at runtime, depends on the *class* of the object referenced in the statement, at runtime.

  From the abstraction of an object, we know what behaviour (method) the object has, and with polymorphism, we get variations of the behaviour with objects from the different classes.

  A side note: interface is not applicable to Python although a Python abstract class with only abstract methods can be thought of as an interface. An abstract class with only abstract methods is a pure abstract class. Interface is necessary in languages that do not support multiple inheritance.

  To determine whether an OO solution adheres to OCP, check whether runtime type identification is performed. Having if-else statements which check for the object type before invoking their methods is a sign that the OO solution does not adhere to OCP.

Activity 3

Explain the benefit of applying OCP to OO solution.

Read

https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view

- Liskov Substitution Principle (LSP)

  Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

  -Barbara Liskov

  Rephrased as:

  What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

  – Robert Martin

  This principle suggests that subclass objects should be able to substitute its superclass objects *perfectly*, and so a function q using the subclass object for the superclass object in a function q should not be aware of the difference.

  To determine whether an OO solution adheres to LSP, check that the inheritance relationship is truly an is-a relationship in terms of the objects' behaviour. For example, there should not be an inheritance relationship between a square and a rectangle as the behaviour of a square is different from a rectangle that a square or a rectangle cannot substitute each other perfectly.

Activity 4

Explain the benefit of applying LSP to OO solution.

Read

https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view

- Interface Segregation Principle (ISP)

  Clients should not be forced to depend on methods they do not use.

  –Robert C. Martin

This principle favours having many thin cohesive interfaces or superclasses over few fat non-cohesive interfaces or superclasses.

Implementing many thin interfaces or inheriting from multiple superclasses that adhere to SRP means that a class is less likely to be forced to implement methods it does not need.

Furthermore, fat interfaces and subclasses are also more likely not to be cohesive and therefore, not adhere to SRP.

ISP is related also to LSP in that a fat subclass makes inheritance less likely to be perfectly an is-a relationship.

Besides multiple inheritance, delegation via object composition can be used to implement an OO solution that does not violate ISP.

To determine whether an OO solution adheres to ISP, check that the interfaces and superclasses are cohesive and adhere to SRP.

Activity 5

Explain the benefit of applying ISP to OO solution.

Read

https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view

- Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

–Robert C. Martin

This principle favours the use of interfaces so that classes do not depend on one another but on the interfaces or pure abstract classes in the case of Python.

When classes depend only on interfaces, it means classes that implement the same interface are interchangeable. For this reason, an OO solution that adheres to DIP is less coupling, making it less fragile, rigid and immobile. The banana-gorilla-forest problem is a result of not applying DIP. To resolve the problem, three interfaces should be defined: a banana interface, a gorilla interface and a forest interface.

To determine whether an OO solution adheres to DIP, check whether the interfaces are used in place of classes.

Activity 5

Explain the benefit of applying DIP to OO solution.

Read

https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQoNC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view

1.2 Design Patterns and SOLID Principles

Design patterns are best practices for the solution design to common occurring programming problems. As best practices, design patterns incorporate good software design principles.

In this section, we look at three design patterns: template method pattern, strategy pattern and model-view-controller pattern and relate them to SOLID principles.

- Template Method pattern

  Recall that the template method pattern is applied whenever an algorithm has both invariant and variant steps.

  The design for such an algorithm is to place the invariant steps in a method of a superclass and the variant steps in a method of its subclasses, each subclass method implementing a variation of the algorithm.

  The method of the superclass will make a polymorphic method call to invoke the variant steps of the algorithm in a method of one of the subclasses.

  The template method pattern applies both

- Open-Closed Principle

  New variation of the algorithm can be implemented without changing the method in the superclass.

- Liskov Substitution Principle

  The objects of the subclasses substitute the superclass object perfectly in the invariant steps of the algorithm.

- Strategy pattern

  Recall that the strategy pattern applies delegation with the aim of reducing the interface of the superclass. An interface or a pure abstract class is first defined with only the required methods to implement a strategy, and each subclass implements the required strategy. When a particular strategy is required, an object from the class that implements the strategy is delegated to perform the strategy.

  The strategy pattern applies Interface Segregation Principle by introducing a new interface or superclass with just the necessary methods for the purpose of implementing variations of the strategies.

- Model-View-Controller pattern

  The model-view-controller pattern is applied in GUI application for the separation of concerns. Each class has a single responsibility. Thus, the model-view-controller pattern applies Single Responsibility Principle.

  Some implementations of the model-view-controller pattern use interface dependencies instead of class dependencies. These implementations define an interface for the model, an interface for the views and an interface for the controller. Models that implement the same interface are interchangeable. Likewise, views and controllers that implement the same interface are interchangeable. These implementations, thus, apply Dependency Inversion Principle.

Summary

Poor object-oriented (OO) program designs causes programs to be fragile, rigid and immobile. Thus, many principles such as SOLID have been proposed to help programmers write better OO solutions that are easy to extend and maintain.

The first of the five SOLID principles is Single Responsibility Principle (SRP). This principle implies that a class should model one thing or is responsible for only one task.

The second SOLID principle is Open-Closed Principle (OCP) which recommends that when there is a variation of behaviour, the variation in behavior should be specified in a new class.

The third SOLID principle is Liskov Substitution Principle (LSP). This principle requires that subclass objects be able to substitute its superclass objects *perfectly*.

The fourth SOLID principle is Interface Segregation Principle (ISP) which favours having many thin cohesive interfaces or superclasses over few fat non-cohesive interfaces or superclasses.

The fifth and also the last SOLID principle is Dependency Inversion Principle (DIP) proposes that interfaces be used so that classes do not depend on one another but on the interfaces.

Design patterns such as template method pattern, strategy pattern and model-view-controller pattern, are best practices for the solution design to common occurring programming problems. Design patterns incorporate good software design principles, including SOLID principles.

References

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view

https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view

https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view

https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view

Formative Assessment

1. State the design principle described by this statement:

Introduce interface so that classes depend on interface instead of classes

A) Single Responsibility Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

*B) Dependency Inversion Principle
@ Correct. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

C) Interface Segregation Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

D) Liskov Substitution Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

2. State the design principle described by this statement:

You should be able to describe the class in a sentence without using “and”

*A) Single Responsibility Principle
@ Correct. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

B) Dependency Inversion Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

C) Interface Segregation Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

D) Liskov Substitution Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

3. State the design principle described by this statement:

A fat interface means a class depends on methods it does not need.

A) Single Responsibility Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

B) Dependency Inversion Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

*C) Interface Segregation Principle
@ Correct. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

D) Liskov Substitution Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

4. State the design principle described by this statement:

The inheritance relationship must be an is-a relationship.

A) Single Responsibility Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

B) Dependency Inversion Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

*C) Liskov Substitution Principle
@ Correct. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

D) Open-closed principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

5. State the design principle described by this statement:

Apply polymorphism so that a new behaviour does not change the existing classes.

A) Single Responsibility Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

B) Dependency Inversion Principle
@ Incorrect. This is not the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

C) Liskov Substitution Principle

@ Incorrect. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

*D) Open-closed principle
@ Correct. This is the correct design principle. Refer to Section 1.1 Introduction to SOLID, Study Unit 6.

Solutions or Suggested Answers

Activity 1

Bad OO design
The problem is caused by high coupling.

Activity 2

highly cohesive as class models only "one thing"

easier to understand and maintain as class is small

low coupling and more reusable

solution is less fragile

Activity 3

new code for extension without modification to existing code
testing is reduced to new code and integration

Activity 4

Cohesive inheritance structure
No anomaly when substituting objects within the class hierarchy

Activity 5

High cohesion
Low coupling

Activity 6

Low coupling
Allows for interchangeable implementations with the same interface