Design with classes

Objects, Classes, and Methods

- Every data value in Python is an *object*
- Every object is an instance of a *class*
- Built in classes include **`int`**, **`float`**, **`str`**, **`tuple`**, **`list`**, **`dict`**
- A class includes operations (*methods*) for manipulating objects of that class (**`append`**, **`pop`**, **`sort`**, **`find`**, etc.)
- Operators (**`==`**, **`[]`**, **`in`**, **`+`**, etc.) are "syntactic sugar" for methods

What Do Objects and Classes Do for Us?

- An object bundles together data and operations on those data
- A computational object can model practically any object in the real (natural or artificial) world
- Some classes come with a programming language
- Any others must be defined by the programmer

Programmer-Defined Classes

- The **`EasyFrame`** class is used to create GUI windows that are easy to set up
- The **`Image`** class is used to load, process, and save images
- Like the built-in classes, these classes include operations to run with their instances

Other Examples

- A **`Student`** class represents information about a student and her test scores
- A **`Rational`** class represents rational numbers and their operations
- A **`Die`** class represents dice used in games
- **`SavingsAccount`**, **`CheckingAccount`**, **`Bank`**, and **`ATM`** are used to model a banking system
- **`Proton`**, **`Neutron`**, **`Electron`**, and **`Positron`** model subatomic particles in nuclear physics

The student class

- Each student has a name and a list of test scores.
- We can use these as the attributes of a class named Student.
- The Student class should allow the user to view a student's name, view a test score at a given position (counting from 1), reset a test score at a given position, view the highest test score, view the average test score, and obtain a string representation of the student's information.

Student class

- When a Student object is created, the user supplies the student's name and the number of test scores
- Assuming that the Student class is defined in a file named student.py

The student class

```
>>> from student import Student
>>> s = Student("Maria", 5)
>>> print(s)
Name: Maria
Scores: 0 0 0 0 0
>>> s.setScore(1, 100)
>>> print(s)
Name: Maria
Scores: 100 0 0 0 0
>>> s.getHighScore()
100
>>> s.getAverage()
20
>>> s.getScore(1)
100
>>> s.getName()
'Maria'
```

Interface

Student Method	What It Does
`s = Student(name, number)`	Returns a **Student** object with the given **name** and **number** of scores. Each score is initially 0.
`s.getName()`	Returns the student's name.
`s.getScore(i)`	Returns the student's *i*th score, **i** must range from 1 through the number of scores.
`s.setScore(i, score)`	Resets the student's *i*th score to score, **i** must range from 1 through the number of scores.
`s.getAverage()`	Returns the student's average score.
`s.getHighScore()`	Returns the student's highest score.
`s.__str()__`	Same as **str(s)**. Returns a string representation of the student's information.

Table 9-1 The interface of the **Student** class

Specifying an Interface

- The user of a class is only concerned with learning the information included in the headers of the class's methods
- This information includes the method name and parameters
- Collectively, this information comprises the class's *interface*
- Docstrings describe what the methods do

Defining (Implementing) a Class

- The *definition* or *implementation* of a class includes completed descriptions of an object's data and the methods for accessing and modifying those data
- The data are contained in *instance variables* and the methods are called *instance methods*
- Related class definitions often occur in the same module

Syntax Template for a Simple Class Definition

```
<imports of other modules used>

class <name>(<parent class name>):
    <docstring for the class>

    <method definitions>
```

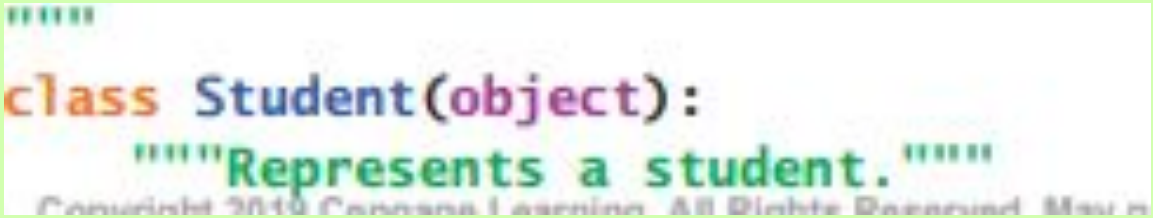
Basically a header followed by several method definitions

class

- The parent class name refers to another class.
- All Python classes, including the built-in ones, are organized in a tree-like class hierarchy.
- At the top, or root, of this tree is the most abstract class, named object, which is built in.
- Each class immediately below another class in the hierarchy is referred to as a subclass, whereas the class immediately above it, if there is one, is called its parent class..

Student class

- If the parenthesized parent class name is omitted from the class definition, the new class is automatically made a subclass of object



Student class

```
def __init__(self, name, number):
    """Constructor creates a Student with the given
    name and number of scores and sets all scores
    to 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)

def getName(self):
    """Returns the student's name."""
    return self.name

def setScore(self, i, score):
    """Resets the ith score, counting from 1."""
    self.scores[i - 1] = score

def getScore(self, i):
    """Returns the ith score, counting from 1."""
    return self.scores[i - 1]
```

Student class

```
def getAverage(self):
    """Returns the average score."""
    return sum(self.scores) / len(self.scores)

def getHighScore(self):
    """Returns the highest score."""
    return max(self.scores)

def __str__(self):
    """Returns the string representation of the
    student."""
    return "Name: " + self.name + "\nScores: " + \
           " ".join(map(str, self.scores))
```

The Class Header

```
class Student(object):
    <docstring for the class>

    <method definitions>
```

By convention, programmer-defined class names are capitalized in Python

Built-in class names, like **`str`**, **`list`**, and **`object`**, are not

All Python classes are *subclasses* of the **`object`** class

Method Definition

- All of the method definitions are indented below the class header
- When a method is called with an object, the interpreter binds the parameter self to that object so that the method's code can refer to the object by name.

Method defnition

- Thus, for example, the code

  s.getScore(4)

- binds the parameter self in the method getScore to the Student object referenced by the variable s.
- The code for getScore can then use self to access that individual object's test scores.

Setting the Initial State

```
def __init__(self, name, number):
    """All scores are initially 0."""
    self.name = name
    self.scores = []
    for count in range(number):
        self.scores.append(0)
```

The **`__init__`** method (also called a *constructor*) is automatically run when an object is instantiated; this method usually sets the object's initial state

- **`s = Student("ram",5)`**) is run, Python automatically runs the constructor or __init__ method of the Student class.
- The purpose of the constructor is to initialize an individual object's attributes

Instance variable

- The attributes of an object are represented as instance variables.
- Each individual object has its own set of instance variables.
- These variables serve as storage for its state.
- The scope of an instance variable (including self) is the entire class definition
- The lifetime of an instance variable is the lifetime of the enclosing object
- the instance variables self.name and self.scores are initialized to a string and a list,

str method

- Many built-in Python classes usually include an __str__ method.
- This method builds and returns a string representation of an object's state.
- When the str function is called with an object, that object's __str__ method is automatically invoked to obtain the string that str returns.

str method

- For example, the function call str(s) is equivalent to the method call s.__str__(), and is simpler to write.
- The function call print(s) also automatically runs str(s) to obtain the object's string representation for output.
- Here is the code for the __str__ method in the Student class:

str method

```
def __str__(self) :
    """Returns the string representation of the student."""
    return "Name: " + self.name + "\nScores: " + \
           " ".join(map(str, self.scores))
```

Mutator and accessor methods

- Methods that allow a user to observe but not change the state of an object are called **accessors**
- Methods that allow a user to modify an object's state are called **mutators**.(eg: setScore)
- The Student class has just one mutator method.
- It allows the user to reset a test score at a given position.
- The remaining methods are accessors.

Rule of thumb for defining classes

1. Before writing a line of code, think about the behavior and attributes of the objects of the new class. What actions does an object perform, and how, from the external perspective of a user, do these actions access or modify the object's state?
2. Choose an appropriate class name, and develop a short list of the methods available to users. This interface should include appropriate method names and parameter names, as well as brief descriptions of what the methods do. Avoid describing how the methods perform their tasks.
3. Write a short script that appears to use the new class in an appropriate way. The script should instantiate the class and run all of its methods. Of course, you will not be able to execute this script until you have completed the next few steps, but it will help to clarify the interface of your class and serve as an initial test bed for it.
4. Choose the appropriate data structures to represent the attributes of the class. These will be either built-in types such as integers, strings, and lists, or other programmer-defined classes.

Rule of thumb for defining classes

5. Fill in the class template with a constructor (an **__init__** method) and an **__str__** method. Remember that the constructor initializes an object's instance variables, whereas **__str__** builds a string from this information. As soon as you have defined these two methods, you can test your class by instantiating it and printing the resulting object.
6. Complete and test the remaining methods incrementally, working in a bottom-up manner. If one method depends on another, complete the second method first.
7. Remember to document your code. Include a docstring for the module, the class, and each method. Do not add docstrings as an afterthought. Write them as soon as you write a class header or a method header. Be sure to examine the results by running **help** with the class name.

Data modelling examples: Rational numbers

- The interface of the Rational class includes a constructor for creating a rational number, an str function for obtaining a string representation, and accessors for the numerator and denominator
- Python allows the programmer to overload many of the built-in operators for use with new data types

Rational numbers

```
"""
File: rational.py
Resources to manipulate rational numbers.
"""

class Rational(object):
    """Represents a rational number."""

    def __init__(self, numer, denom) :
        """Constructor creates a number with the given
        numerator and denominator and reduces it to lowest
        terms."""
        self.numer = numer
        self.denom = denom
        self._reduce()

    def numerator(self):
        """Returns the numerator."""
        return self.numer
```

Rational numbers

```
def denominator(self):
    """Returns the denominator."""
    return self.denom

def __str__(self):
    """Returns the string representation of the
    number."""
    return str(self.numer) + "/" + str(self.denom)

def _reduce(self):
    """Helper to reduce the number to lowest terms."""
    divisor = self._gcd(self.numer, self.denom)
    self.numer = self.numer // divisor
    self.denom = self.denom // divisor

def _gcd(self, a, b):
    """Euclid's algorithm for greatest common
    divisor (hacker's version)."""
    (a, b) = (max(a, b), min(a, b))
    while b > 0:
        (a, b) = (b, a % b)
    return a
```

Rational Number Arithmetic and Operator Overloading

- For a built-in type such as int or float, each arithmetic operator corresponds to a special method name.
- You will see many of these methods by entering dir(int) or dir(str) at a shell prompt
- The object on which the method is called corresponds to the left operand, whereas the method's second parameter corresponds to the right operand.
- Thus, for example, the code x + y is actually shorthand for the code x.__add__(y)

Rational Number Arithmetic and Operator Overloading

Operator	Method Name
+	`__add__`
-	`__sub__`
*	`__mul__`
/	`__div__`
%	`__mod__`

Table 9-3 Built-in arithmetic operators and their corresponding methods

Rational Number Arithmetic and Operator Overloading

- To overload an arithmetic operator, you just define a new method using the appropriate method name. The code for each method applies a rule of rational number arithmetic

Type of Operation	Rule
Addition	$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1) / d_1d_2$
Subtraction	$n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1) / d_1d_2$
Multiplication	$n_1/d_1 * n_2/d_2 = n_1n_2 / d_1d_2$
Division	$n_1/d_1 / n_2/d_2 = n_1d_2 / d_1n_2$

Table 9-4 Rules for rational number arithmetic

Rational number addition

```
def __add__(self, other):
    """Returns the sum of the numbers.
    self is the left operand and other is
    the right operand."""
    newNumer = self.numer * other.denom + \
               other.numer * self.denom
    newDenom = self.denom * other.denom
    return Rational(newNumer, newDenom)
```

Rational number addition

- Note that the parameter self is viewed as the left operand of the operator, whereas the parameter other is viewed as the right operand.
- Operator overloading is another example of an abstraction mechanism.
- In this case, programmers can use operators with single, standard meanings even though the underlying operations vary from data type to data type

Comparison Methods

- You can compare integers and floating-point numbers using the operators ==, !=, , <=, and >=.
- When the Python interpreter encounters one of these operators, it uses a corresponding method defined in the float or int class.
- Each of these methods expects two arguments. The first argument, self, represents the operand to the left of the operator, and the second argument represents the other operand.

Comparison Methods

Operator	Meaning	Method
==	Equals	__eq__
!=	Not equals	__ne__
<	Less than	__lt__
<=	Less than or equal	__le__
>	Greater than	__gt__
>=	Greater than or equal	__ge__

Table 9-5 The comparison operators and methods

- The simplest way to compare two rational numbers is to compare the product of the extremes and the product of the means.
- The extremes are the first numerator and the second denominator, whereas the means are the second numerator and the first denominator.
- Thus, the comparison 1/6 , 2/3 translates to 1 * 3 , 2 * 6.

Comparison

```
def __lt__(self, other):
    """Compares two rational numbers, self and other,
    using <."""
    extremes = self.numer * other.denom
    means = other.numer * self.denom
    return extremes < means
```

Equality Method

```
def __eq__(self, other):
    """Tests self and other for equality."""
    if self is other:                       # Object identity?
        return True
    elif type(self) != type(other):         # Types match?
        return False
    else:
        return self.numer == other.numer and \
               self.denom == other.denom
```

Equality

- The method first tests the two operands for object identity using Python's is operator. The is operator returns True if self and other refer to the exact same object.
- If the two objects are distinct, the method then uses Python's type function to determine whether or not they are of the same type. If they are not of the same type, they cannot be equal.
- Finally, if the two operands are of the same type, the second one must be a rational number, so it is safe to access the components of both operands to compare them for equality in the last alternative

Structuring classes with Inheritance and polymorphism

- In Python, all classes automatically extend the built-in object class, which is the most general class possible

```
class <new class name>(<existing parent class name>):
```

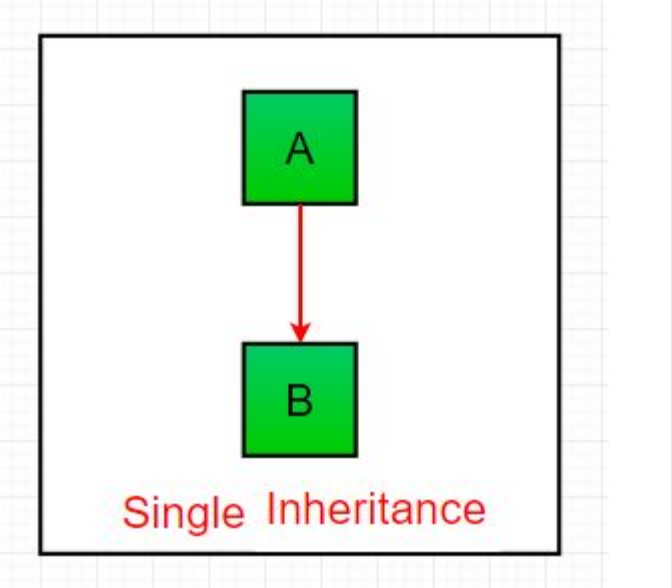
Inheritance

- Inheritance is defined as the capability of one class to derive or inherit the properties from some other class and use it whenever needed. Inheritance provides the following properties:
- It **represents real-world relationships** well.
- It **provides reusability** of code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is **transitive i**n nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Inheritance

- The class which inherits another class is called subclass or child class, and the other class is the parent class
- Inheritance is categorized based on the hierarchy followed and the number of parent classes and subclasses involved.
- There are five types of inheritances:
- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

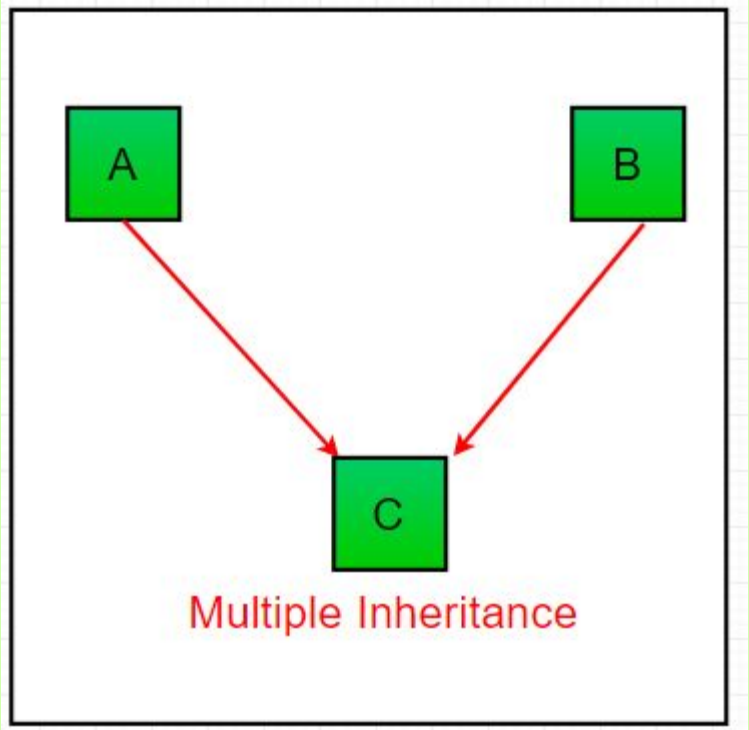
Single Inheritance



Single Inheritance

- This type of inheritance enables a subclass or derived class to inherit properties and characteristics of the parent class, this avoids duplication of code and improves code reusability.

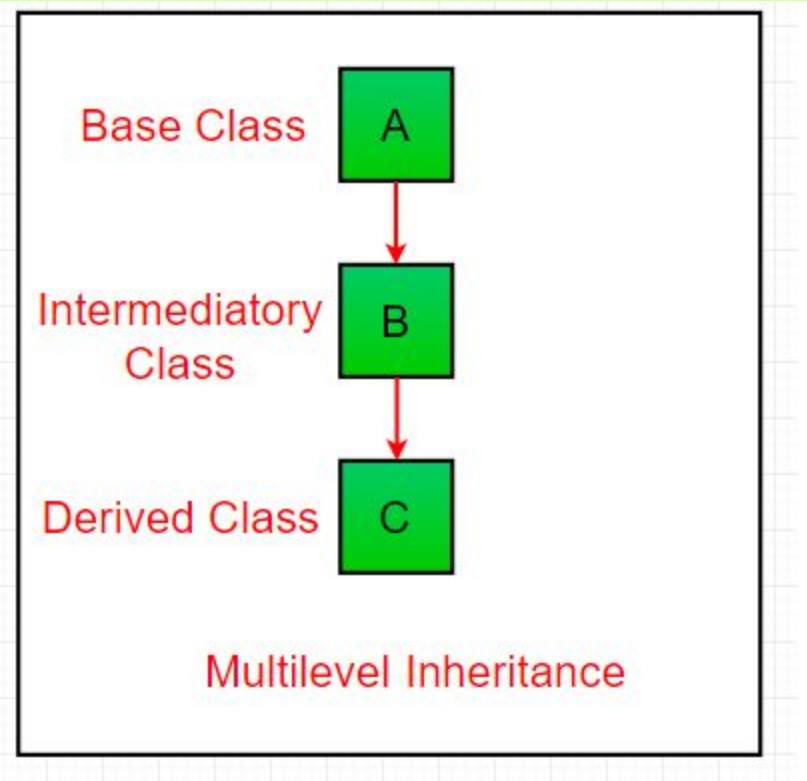
Multiple Inheritance



Multiple Inheritance

- When a class can be derived from more than one base class this type of inheritance is called multiple inheritance.
- In multiple inheritance, all the features of the base classes are inherited into the derived class.

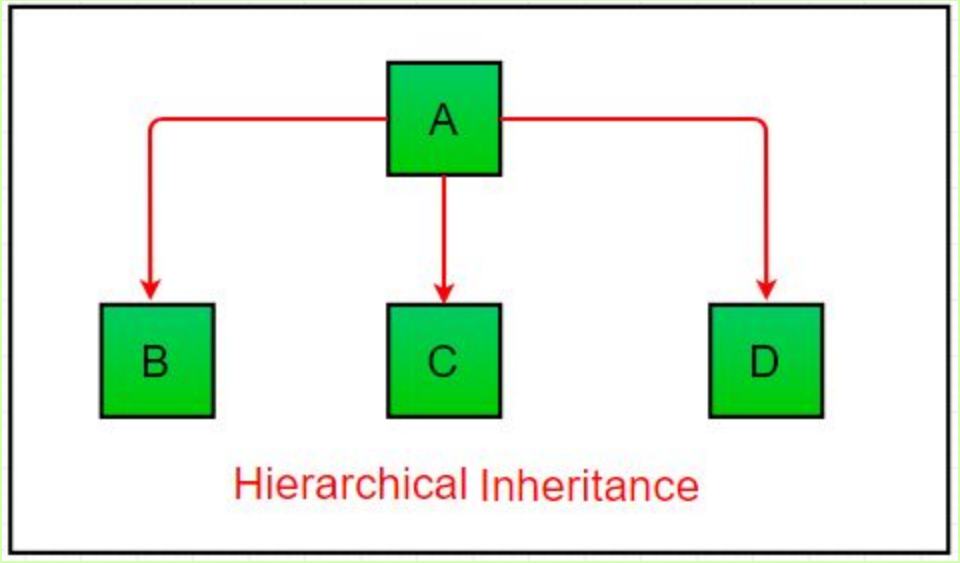
Multilevel Inheritance



Multilevel Inheritance

- In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class.
- This is similar to a relationship representing a child and grandfather.

Hierarchical Inheritance



Hierarchical Inheritance

- When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance.
- In this program, we have a parent (base) class and two child (derived) classes.

Hybrid Inheritance

- Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Polymorphic Methods

- The word polymorphism means having many forms.
- In programming, polymorphism means the same function name (but different signatures) being used for different types.

Polymorphic functions

```
#len() being used for a string
print(len("geeks"))

# len() being used for a list
print(len([10, 20, 30]))
```

Example 2

```
def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Example 3

```
class India():
        def capital(self):
                print("New Delhi is the capital of India.")
        def language(self):
             print("Hindi is the most widely spoken language of
India.")

        def type(self):
                 print("India is a developing country.")
```

Example 3

```
class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")
```

Example 3

```
obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Abstract class

- An abstract class can be considered as a blueprint for other classes.
- It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- An abstract method is a method that has a declaration but does not have an implementation.

Abstract class

- Abstract classes may or may not contain abstract methods, i.e., methods without body.
- But, if a class has at least one abstract method, then the class must be declared abstract.

Abstract class

- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC

Abstract class

- So we use an abstract class as a template and according to the need, we extend it and build on it before we can use it.
- Due to the fact, an abstract class is not a concrete class, it cannot be instantiated
- When we create an object for the abstract class it raises an *error*.

Exception handling

- Exceptions are raised when the program is syntactically correct, but the code resulted in an error.
- This error does not stop the execution of the program, however, it changes the normal flow of the program.
- **Try and except** statements are used to catch and handle exceptions in Python.
- Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Common exceptions

- **ZeroDivisionError:** Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.

Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

Example 2

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b;
    print("a/b = %d"%c)
except:
    print("can't divide by zero")
else:
    print("Hi I am else block")
```

Handling multiple exceptions

- A try statement can have more than one except clause, to specify handlers for different exceptions.
- At most one handler will be executed.

```
try:
    # statement(s)
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)
```

Example 1

```
def fun(a):
        if a < 4:
                # throws ZeroDivisionError for a = 3
                b = a/(a-3)
        # throws NameError if a >= 4
        print("Value of b = ", b)

try:
        fun(3)
        fun(5)

except ZeroDivisionError:
        print("ZeroDivisionError Occurred and Handled")
except NameError:
        print("NameError Occurred and Handled")
```

Example 2

```
string = input()
try:
    num = int(input())
    print(string+num)
 except (TypeError, ValueError) as e:
    print(e)
```

Finally

- Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception

```
try:
    # Some Code....
except:
    # optional block # Handling of exception (if required)
else:
    # execute if no exception
finally:
    # Some code .....(always executed)
```

Example

```
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed  regardless of exception     generation.
    print('This is always executed')
```

Interfaces

- An interface acts as a template for designing classes.
- Interfaces also define methods the same as classes, but abstract methods, whereas class contains nonabstract methods.
- Abstract methods are those methods without implementation or which are without the body.
- So the interface just defines the abstract method without implementation.
- The implementation of these abstract methods is defined by classes that implement an interface.

Cntd:

- There are two ways in python to create and implement the interface, which are –

1. Informal Interfaces
2. Formal Interfaces

- An informal interface in Python is a class. It defines methods that can be overridden but without force enforcement.
- An **informal interface in python** is termed as a protocol because it is informal and cannot be enforced formally. :

Formal Interface

- A formal Interface is an interface which enforced formally
- To create a formal interface, we need to use ABCs (Abstract Base Classes).
- The ABCs is explained as we define a class that is abstract in nature, we also define the methods on the base class as abstract methods.
- Any object we are deriving from the base classes is forced to implement those methods.

Cntd:

- Note that the interface cannot be instantiated, which means that we cannot create the object of the interface.
- So we use a base class to create an object, and we can say that the object implements an interface.

Cntd:

```
import abc
 class Myinterface( abc.ABC ):
        @abc.abstractclassmethod
         def disp( ):
               pass
class Myclass(Myinterface):
         def disp():
               print(" Hello from Myclass ")
o1=Myclass()
o1.disp()
```

Interface vs Abstract class

Python interface	**Python abstract class**
An interface is a set of methods and attributes on that object.	We can use an abstract base class to define and enforce an interface.
All methods of an interface are abstract	An abstract class can have abstract methods as well as concrete methods.
We use an interface if all the features need to be implemented differently for different objects.	Abstract classes are used when there is some common feature shared by all the objects as they are.
The interface is slow as compared to the abstract class.	Abstract classes are faster.

exercises

2. The name used to refer to the current instance of a class within the class definition is

   a. `this`
   b. `other`
   c. `self`

3. The purpose of the `__init__` method in a class definition is to

   a. build and return a string representation of the instance variables
   b. set the instance variables to initial values

4. A method definition

   a. can have zero or more parameter names
   b. always must have at least one parameter name, called `self`

5. The scope of an instance variable is

   a. the statements in the body of the method where it is introduced
   b. the entire class in which it is introduced
   c. the entire module where it is introduced

6. An object's lifetime ends

   a. several hours after it is created
   b. when it can no longer be referenced anywhere in a program
   c. when its data storage is recycled by the garbage collector

7. A class variable is used for data that

   a. all instances of a class have in common
   b. each instance owns separately

8. Class `B` is a subclass of class `A`. The `__init__` methods in both classes expect no arguments. The call of class `A`'s `__init__` method in class `B` is

   a. `A.__init__()`

1. An instance variable refers to a data value that
   a. is owned by a particular instance of a class and no other
   b. is shared in common and can be accessed by all instances of a given class

10. A polymorphic method
   a. has a single header but different bodies in different classes
   b. creates harmony in a software system