(Abstraction and Inheritance)

Inheritance:

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called *derived* classes, the classes that we derive from are called *base* classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

inheritance.py

```
#!/usr/bin/env python

# inheritance.py

class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        super().__init__()

        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")

d = Dog()
d.whoAmI()
d.eat()
d.bark()
```

(Abstraction and Inheritance)

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class. The derived class inherits the functionality of the base class. It is shown by the eat method. The derived class modifies existing behavior of the base class, shown by the whoAmI method. Finally, the derived class extends the functionality of the base class, by defining a new bark method i.e.;

```
class Dog(Animal):
   def __init__(self):
      super().__init__()
      print("Dog created")
```

We put the ancestor classes in round brackets after the name of the descendant class. If the derived class provides its own __init__ method and we want to call the parent constructor, we have to explicitly call the base class __init__ method with the help of the super function.

```
$ ./inherit.py
Animal created
Dog created
Dog
Eating
Woof!
```

Abstraction:

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**

In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialing, etc., but we don't know how these operations are happening in the background. Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.

- In Python, an abstraction is used to hide the irrelevant data/class to reduce the complexity. It also enhances the application efficiency.
- It can be achieved by using abstract classes and interfaces.

(Abstraction and Inheritance)

- A class that consists of one or more abstract method is called the abstract class.
- Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass.
- Abstraction classes are meant to be the blueprint of the other class.
- An abstract class can be useful when we are designing large functions.
- An abstract class is also helpful to provide the standard interface for different implementations of components.
- Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

1. from abc import ABC
2. class ClassName(ABC):

We import the ABC class from the **abc** module.

```
    # Python program demonstrate

    # abstract base class work

from abc import ABC, abstractmethod

class Car(ABC):

   def mileage(self):

      pass

class Tesla(Car):

   def mileage(self):

      print("The mileage is 30kmph")

class Suzuki(Car):

   def mileage(self):

      print("The mileage is 25kmph ")
```

(Abstraction and Inheritance)

```
class Duster(Car):
    def mileage(self):
        print("The mileage is 24kmph ")


class Renault(Car):
   def mileage(self):
          print("The mileage is 27kmph ")


# Driver code
t= Tesla ()
t.mileage()


r = Renault()
r.mileage()


s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
```

Output:

(Abstraction and Inheritance)

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

Explanation -

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

(Abstraction and Inheritance)

1. **Predict the output:**

```
from abc import ABC
    class Polygon(ABC):
      # abstract method
      def sides(self):
        pass
    class Triangle(Polygon):

      def sides(self):
        print("Triangle has 3 sides")

    class Pentagon(Polygon):
      def sides(self):
        print("Pentagon has 5 sides")

    class Hexagon(Polygon):
     def sides(self):
        print("Hexagon has 6 sides")

    class square(Polygon):
     def sides(self):
        print("I have 4 sides")

    # Driver code
    t = Triangle()
    t.sides()

    s = square()
    s.sides()

    p = Pentagon()
    p.sides()
```

```
k = Hexagon()
k.sides()
```

Output:

```
Triangle has 3 sides
Square has 4 sides
Pentagon has 5 sides
Hexagon has 6 sides
```

2. **Predict the output:**

```
import abc
class parent:
   def hi(self):
      pass

class child(parent):
   def hi(self):
      print("child class")
print( issubclass(child, parent))
print( isinstance(child(), parent))
```

Output:

```
True

True
```

3. **Predict the output:**

```
import abc

from abc import ABC, abstractmethod

class R(ABC):

    def rk(self):

         print("The Base Class")
```

(Abstraction and Inheritance)

```
class K(R):
    def rk(self):
        super().rk()
        print("The subclass ")
r = K()
r.rk()
```

Output:

```
The Base Class
The subclass
```

4. **Predict the output**

```
import abc
from abc import ABC, abstractmethod
class parent(ABC):
  @abc.abstractproperty
  def met(self):
    return "parent class"
class child(parent):
  @property
  def met(self):
    return "child class"
```

```
try:
   r =parent()
   print( r.met)
except Exception as err:
   print (err)
r = child()
print (r.met)
```

Output:

```
Can't instantiate abstract class parent with abstract method met
child class
```

5. **Predict the output:**

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
```

```
d = Derived()

print(issubclass(Derived,Calculation2))

print(issubclass(Calculation1,Calculation2))
```

Output:

```
True

False
```

6. **Predict the output:**

```
class Base:

   def __init__(self):

   # Protected member

      self._a = 2

class Derived(Base):

   def __init__(self):

      Base.__init__(self)

      print("Calling protected member of base class: ")

      print(self._a)

obj1 = Derived()

obj2 = Base()

print(obj2._a)
```

Output:

```
Calling protected member of base class:

2

2
```

7. **Predict the output**

```
class MyClass:
    hiddenVariable = 10
myObject = MyClass()
print(myObject.hiddenVariable)
class MyClass:
    __hiddenVariable = 0
    def add(self, increment):
        self.__hiddenVariable += increment
        print (self.__hiddenVariable)
myObject = MyClass()
myObject.add(2)
myObject.add(5)
```

Output:

```
10

2

7
```

(Abstraction and Inheritance)

8. Predict the Output:

```
class Person(object):
   def __init__(self, name):
      self.name = name

   def getName(self):
      return self.name

   def isEmployee(self):
      return False

class Employee(Person):

   def isEmployee(self):
      return True

emp = Person("student1")
print(emp.getName(), emp.isEmployee())

emp = Employee("student2")
print(emp.getName(), emp.isEmployee())
```

Output:

```
student1 False
student2 True
```

9. Predict the Output:

```
class Person(object):
   def __init__(self, name):
      self.name = name
   def getName(self):
         return self.name
   def isEmployee(self):
      return False
class Employee(Person):
   def __init__(self, name, eid):
      super(Employee, self).__init__(name)
```

```
        self.empID = eid
    def isEmployee(self):
        return True
    def getID(self):
        return self.empID
emp = Employee("student", "E101")
print(emp.getName(), emp.isEmployee(), emp.getID())
```

Output:

```
student True E101
```

10. Write a program for following:

Write a Python class to find validity of a string of parentheses, '(', ')', '{', '}', '[' and ']. These brackets must be close in the correct order, for example "()" and "()[]{}" are valid, but "[)", "({[)]" and "{{{" are invalid.

Solution:

```
class py_solution:
   def is_valid_parenthese(self, str1):
        stack, pchar = [], {"(": ")", "{": "}", "[": "]"}
        for parenthese in str1:
            if parenthese in pchar:
                stack.append(parenthese)
            elif len(stack) == 0 or pchar[stack.pop()] != parenthese:
                return False
        return len(stack) == 0

print(py_solution().is_valid_parenthese("(){}[]"))
print(py_solution().is_valid_parenthese("()[{)}"))
print(py_solution().is_valid_parenthese("()"))
```