**Q1. What is the purpose of Python's OOP?**
A-
The purpose of Python’s OOP is to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

**Q2. Where does an inheritance search look for an attribute?**
A-
An inheritance search looks for an attribute first in the instance object, then in the class the instance was created from, then in all higher superclasses, progressing from left to right (by default).

The search stops at the first place the attribute is found.

**Q3. How do you distinguish between a class object and an instance object?**
A-
Class variables are declared inside a class but outside of any function. Instance variables are declared inside the constructor which is the __init__method.

Updating the class variables will affect all the instances of the class. Class variables affect the entire class. Thus, when a class variable is updated, all the instances are updated. Global existence of class variables are shared variables.
Instance variables take unique values for each instance and are tightly coupled with the object itself. We assign the values for them when creating the instance.

**Q4. What makes the first argument in a class’s method function special?**
A-
In object-oriented programming, whenever we define methods for a class, we use self as the first parameter in each case.

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")
```

```
# Creating instance of the class
cat1 = Cat('Andy', 2)
cat2 = Cat('Phoebe', 3)
```

The self keyword is used to represent an instance (object) of the given class.
In this case, the two *Cat* objects *cat1* and *cat2* have their own name and age attributes.
If there was no *self* argument, the same class couldn't hold the information for both these objects.

However, since the class is just a blueprint, *self* allows access to the attributes and methods of each object in python. This allows each object to have its own attributes and methods. Thus, even long before creating these objects, we reference the objects as self while defining the class.

**Q5. What is the purpose of the init method?**
A-
__*init*__ acts as the constructor in python. Constructors are used to initialize the object's state.

The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.

Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.

It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)


p = Person('Human')
p.say_hi()
```

**Q6. What is the process for creating a class instance?**
A-
To create instances of a class, you call the class using the class name and pass in whatever arguments its __init__ method accepts.

Ex; In above code we created an instance of *Person* class by passing name as an argument.
*p = Person('Human')*
*p.say_hi()*

**Q7. What is the process for creating a class?**
A-
Class creates a user-defined data structure, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
A class is like a blueprint for an object.

Classes are created by keyword *class*.
Attributes are the variables that belong to a class. Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

**Q8. How would you define the superclasses of a class?**
A-
A superclass is the class from which many subclasses can be created. The subclasses inherit the characteristics of a superclass. The superclass is also known as the parent class or base class.

```
class Robot:
    def __init__(self, name):
        self.name = name
    def say_hi(self):
        print("Hi, I am " + self.name)

class PhysicianRobot(Robot):
    Pass

x = Robot("Marvin")
y = PhysicianRobot("James")

print(x, type(x))
print(y, type(y))
y.say_hi()
```

PhysicianRobot is a subclass of Robot, it inherits, in this case, both the method __*init*__ and *say_hi*.

Inheriting these methods means that we can use them as if they were defined in the *PhysicianRobot* class. When we create an instance of *PhysicianRobot*, the __*init*__ function will also create a name attribute.

We can apply the say_hi method to the *PhysisicianRobot* object y as well.

**Q9. What is the relationship between classes and modules?**
A-

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables.

A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Modules are like singleton objects — there can be only one. Anywhere where you import the module, it is the same object.

Classes can be also subclassed / extended / modified using inheritance. Modules can not.

**Q10. How do you make instances and classes?**
A-

```
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)


p = Person('Human')
p.say_hi()
```

Classes are created by keyword *class*. We define constructor of the class using __init__.
We create instances of the class using syntax: *object_name = Class_name(arguments)*

**Q11. Where and how should class attributes be created?**
A-
Class attributes belong to the class itself; they will be shared by all the instances. Such attributes are defined in the class body parts usually at the top.

```
class sampleclass:
    count = 0     # class attribute

    def increase(self):
        sampleclass.count += 1

# Calling increase() on an object
s1 = sampleclass()
s1.increase()
print(s1.count)

# Calling increase on one more
# object
s2 = sampleclass()
s2.increase()
print(s2.count)

print(sampleclass.count)
```

**Q12. Where and how are instance attributes created?**
A-
Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

Output-
>>> std = Student('Bill',25)
>>> std.name
'Bill'
```

**Q13. What does the term "self" in a Python class mean?**
A-
The self keyword is used to represent an instance (object) of the given class.

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")
# Creating instance of the class
cat1 = Cat('Andy', 2)
cat2 = Cat('Phoebe', 3)
```

In this case, the two *Cat* objects *cat1* and *cat2* have their own name and age attributes.
If there was no *self* argument, the same class couldn't hold the information for both these objects.

*self* allows access to the attributes and methods of each object in python.

**Q14. How does a Python class handle operator overloading?**
A-
Operator Overloading means giving extended meaning beyond their predefined operational meaning.

For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class.

Also, the same built-in operator or function shows different behavior for objects of different classes, this is called Operator Overloading.

When we use an operator(Ex;  +), the corresponding magic method ( Ex; __add__ ) is automatically invoked,  in which the operation for + operator is defined. Thereby changing this magic method's code, we can give extra meaning to the + operator.

```
# Python Program to perform addition
# of two complex numbers using binary
# + operator overloading.

class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

     # adding two objects
    def __add__(self, other):
        return self.a + other.a, self.b + other.b

Ob1 = complex(1, 2)
Ob2 = complex(2, 3)
Ob3 = Ob1 + Ob2
print(Ob3)
```

**Q15. When do you consider allowing operator overloading of your classes?**
A-
When we want to  change the meaning of an operator in Python depending upon the operands used.

Ex; For performing arithmetic operations on complex numbers, we can overload operators like +, -, *, /, etc.

**Q16. What is the most popular form of operator overloading?**
A-
A very popular and convenient example is the Addition (+) operator. Just think how the '+' operator operates on two numbers and the same operator operates on two strings.

It performs "Addition" on numbers whereas it performs "Concatenation" on strings.

**Q17. What are the two most important concepts to grasp in order to comprehend Python OOP code?**
A-
The most important concept of Object-Oriented Programming is the notion of keeping the data and the related code close.

Object-oriented programming has four basic concepts: encapsulation, abstraction, inheritance and polymorphism.

**Q18. Describe three applications for exception processing.**
A- **<Question Unclear>**
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.

**Q19. What happens if you don't do something extra to treat an exception?**
A-
Exceptions do not stop the execution of the program, however, it changes the normal flow of the program. Hence it is important to handle them.

**Q20. What are your options for recovering from an exception in your script?**
A-
The suspicious code is to be placed in a *try:* block.

After the *try:* block, include an *except:* statement, which handles the Exception by and has the code which is supposed to be run if an exception occurs.

*else* statement has the code which is supposed to be run if the exception does not occur.

At last there is *finally:* block which runs regardless if an exception occurs or not.

**Q21. Describe two methods for triggering exceptions in your script.**
A-
One easier way could be to raise a ZeroDivision exception by dividing a number by 0.
Another way is to try to access the element at an index which is greater than size of the iterable i.e. KeyError

**Q22. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.**
A-
For specifying actions to be executed at termination time, regardless of whether or not an exception exists, these actions should be included in the *finally* clause.

**Q23. What is the purpose of the try statement?**
A-
The code which can fail during execution (Ex; Invalid input, or connection issue during DB connection, etc) is placed in try: clause. This way if the code fails and throws an exception then that exception can be caught and this way the code execution will not fail.

**Q24. What are the two most popular try statement variations?**
A-
Following are the popular try variations:

Try/Except/Else-
When attaching an else statement to the end of a try/except, this code will be executed after the try has been completed, but only if no exceptions occur.
Try/Except/Finally-
When attaching a finally statement to the end of a try/except, this code will be executed after the try has been completed, regardless of exceptions.

**Q25. What is the purpose of the raise statement?**
A-
Python raise Keyword is used to raise exceptions or errors. The raise keyword raises an error and stops the control flow of the program.
It is used to bring up the current exception in an exception handler so that it can be handled further up the call stack.

Syntax of the raise keyword :

*raise {name_of_ the_ exception_class}*

Ex;
```
s = 'apple'

try:
   num = int(s)
except ValueError:
   raise ValueError("String can't be changed into integer")
```

Raising an exception Without Specifying Exception Class-
When we use the raise keyword, there’s no compulsion to give an exception class along with it.

When we do not give any exception class name with the raise keyword, it reraises the exception that last occurred.

```
s = 'apple'

try:
   num = int(s)
except:
   raise
```

Advantages of the raise keyword:

- It helps us raise exceptions when we may run into situations where execution can't proceed.
- It helps us reraise an exception that is caught.
- Raise allows us to throw one exception at any time.
- It is useful when we want to work with input validations.

**Q26. What does the assert statement do, and what other statement is it like?**
A-
Assertions are statements that assert or state a fact confidently in your program.
For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert the divisor is not equal to zero.

Syntax for using Assert in Pyhton:

*assert <condition>,<error message>*

assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.

assert statements can also have an optional  error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

**Q27. What is the purpose of the with/as argument, and what other statement is it like?**
A-
In Python, the with statement replaces a try-catch block with a concise shorthand. More importantly, it ensures closing resources right after processing them.

A common example of using the with statement is reading or writing to a file. A function or class that supports the with statement is known as a context manager.

A context manager allows you to open and close resources right when you want to.
For example, the open() function is a context manager. When you call the open() function using the with statement, the file closes automatically after you've processed the file.

```
with open("example.txt", "w") as file:
    file.write("Hello World!")
```

Under the hood, the with statement replaces this kind of try-catch block:

```
file = open("example.txt", "w")

try:
    file.write("hello world")
finally:
    file.close()
```

**Q28. What are *args, **kwargs?**
A-
There are two special symbols used for passing arguments:-
*args (Non-Keyword Arguments)
**kwargs (Keyword Arguments)

We use the “wildcard” or “*” notation like this – *args OR **kwargs – as our function’s argument when we have doubts about the number of arguments we should pass in a function.

***<u>*args</u>* :**
The special syntax *args in function definitions are used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

The syntax is to use the symbol * to take in a variable number of arguments; by convention, it is often used with the word args.

*args allows us to take in more arguments than the number of formal arguments that you previously defined.

With *args, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

For example, we want to make a multiply function that takes any number of arguments and is able to multiply them all together. It can be done using *args.

Using the *, the variable that we associate with the * becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as map and filter, etc.

```
def myFun(arg1, *args):
    print("First argument :", arg1)
    for arg in args:
        print("Next argument through *argv :", arg)
```

```
 # Function Call
myFun('Hello', 'Hi', 'to', 'Welcome', ‘This’, ‘is’, ‘good’)
```

Output:

```
First argument : Hello
Next argument through *argv : Hi
Next argument through *argv : to
Next argument through *argv : Welcome            …. good
```

****kwargs**

The special syntax **kwargs in function definitions in python are used to pass a keyworded, variable-length argument list.

We use the name kwargs with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

A keyword argument is where you provide a name to the variable as you pass it into the function.

We can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn’t seem to be any order in which they were printed out.

```
def connect_db(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))

# Driver code
myFun(url = 'xyz.db', user_name = 'alpha_user', password = 'alpha_user@123')
```

**Q29. How can I pass optional or keyword parameters from one function to another?**
A-

There are two main ways to pass optional parameters in python
- Without using keyword arguments.
    - The order of parameters should be maintained i.e. the order in which parameters are defined in function should be maintained while calling the function.
    - The value of the default arguments can be either passed or ignored.

- By using keyword arguments.

```
# Here b is predefined and hence is optional.
def func(a, b=1098):
    return a+b

# Using optional arguments
print(func(2)) # Here b will have 1098 as a value by default

# Without using keyword arguments
print(func(2, 5)) # Here 5 will be passed to optional argument ‘b’

# using keyword arguments
print(func(2, b = 5)) # Here 5 will be passed to optional argument ‘b’ using keyword
```

**Q30. What are Lambda Functions?**
A-
Lambda functions are inline functions with only the logical representation.

They are also called Anonymous functions and don’t need to be defined using the def keyword.

Syntax-

*lambda <arguments> : expression*

*Ex;*
*add_ten = lambda x, y = 10 : x + y*

**Q31. Explain Inheritance in Python with an example?**
A -
Inheritance is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class.

Inheritance is the capability of one class to derive or inherit the properties from another class

```
class Person(object):

  # Constructor
  def __init__(self, name, id):
     self.name = name
     self.id = id

  # To check if this person is an employee
  def Display(self):
     print(self.name, self.id)
```

```
# Driver code
emp = Person("Satyam", 102) # An Object of Person
emp.Display()

class Emp(Person):

  def print(self):
    print("Emp class called")

emp_details = Emp("Mayank", 103)

# calling parent class function
emp_details .Display()

# Calling child class function
emp_details.print()
```

**Q32. Suppose class C inherits from classes A and B as class C(A,B).Classes A and B both have their own versions of method func(). If we call func() from an object of class C, which version gets invoked?**

A-

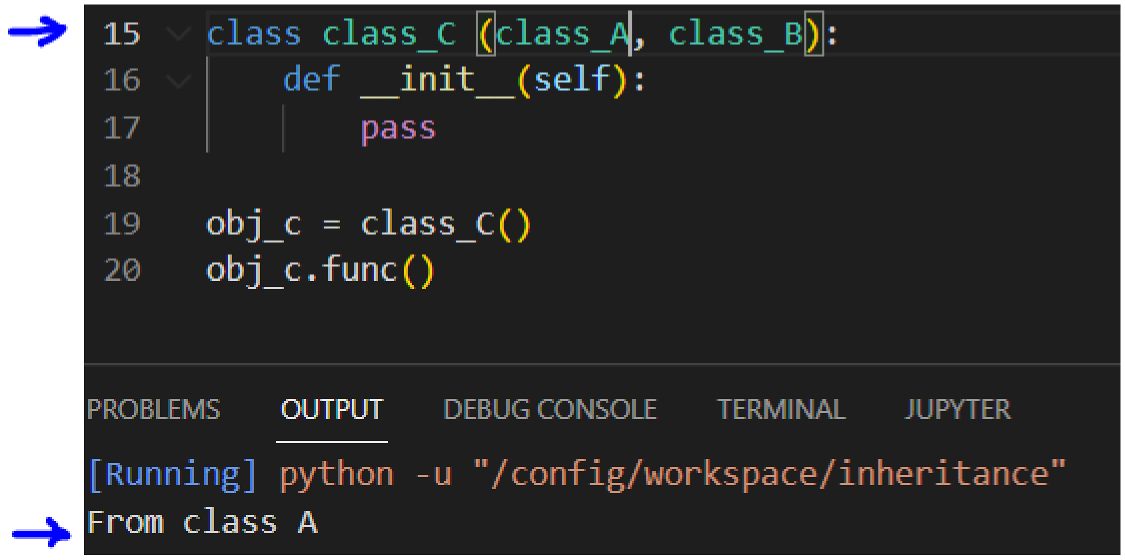
In Such case, In  A class definition, where a child class SubClassName inherits from the parent classes BaseClass1, BaseClass2, BaseClass3, and so on, and both BaseClasses have same function then the function of the BaseClass which was specified first in the childClass definition will be executed.

Ex;

Consider 2 base classes having same method named func():

```
class class_A:
   def __init__(self):
       pass

   def func(self):
       print("From class A")

class class_B:
   def __init__(self):
       pass

   def func(self):
       print("From class B")
```

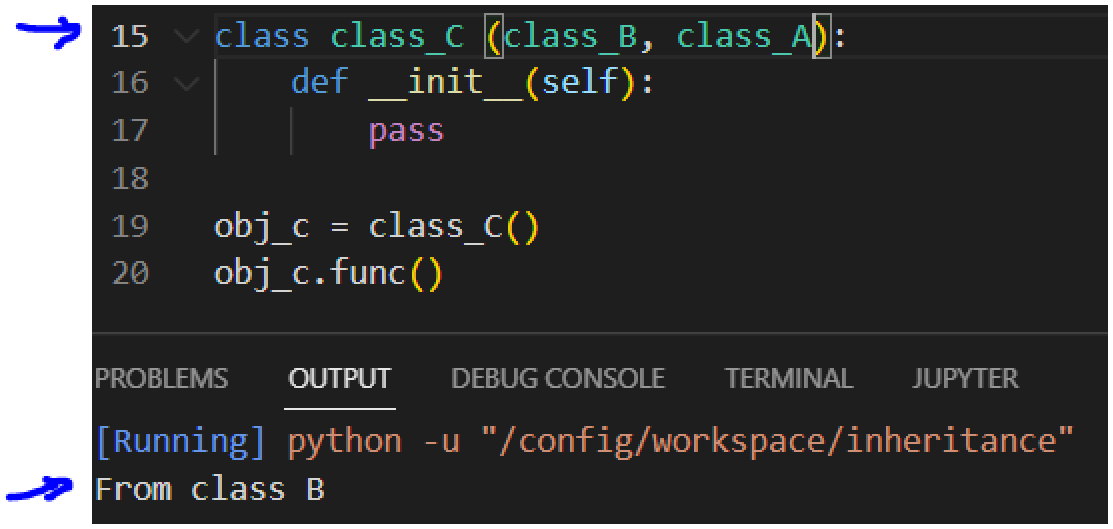
Here while defining the child class C when we specify class_A first then func() of class_A will be executed:



```
class class_C (class_A, class_B):
    def __init__(self):
        pass

obj_c = class_C()
obj_c.func()
```

```
[Running] python -u "/config/workspace/inheritance"
From class A
```

Similarly, when we specify class_B first then func() of class_B will be executed:



```
class class_C (class_B, class_A):
    def __init__(self):
        pass

obj_c = class_C()
obj_c.func()
```

```
[Running] python -u "/config/workspace/inheritance"
From class B
```

**Q33. Which methods/functions do we use to determine the type of instance and inheritance?**

A-

**isinstance() and issubclass()**

The isinstance() method checks whether an object is an instance of a class whereas issubclass() method asks whether one class is a subclass of another class (or other classes).

**isinstance(object, classinfo)**
Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof.

```
sample_list = ["Emma", "Stevan", "Brent"]
res = isinstance(sample_list, list)
print(res)
```

```
Output - true
```

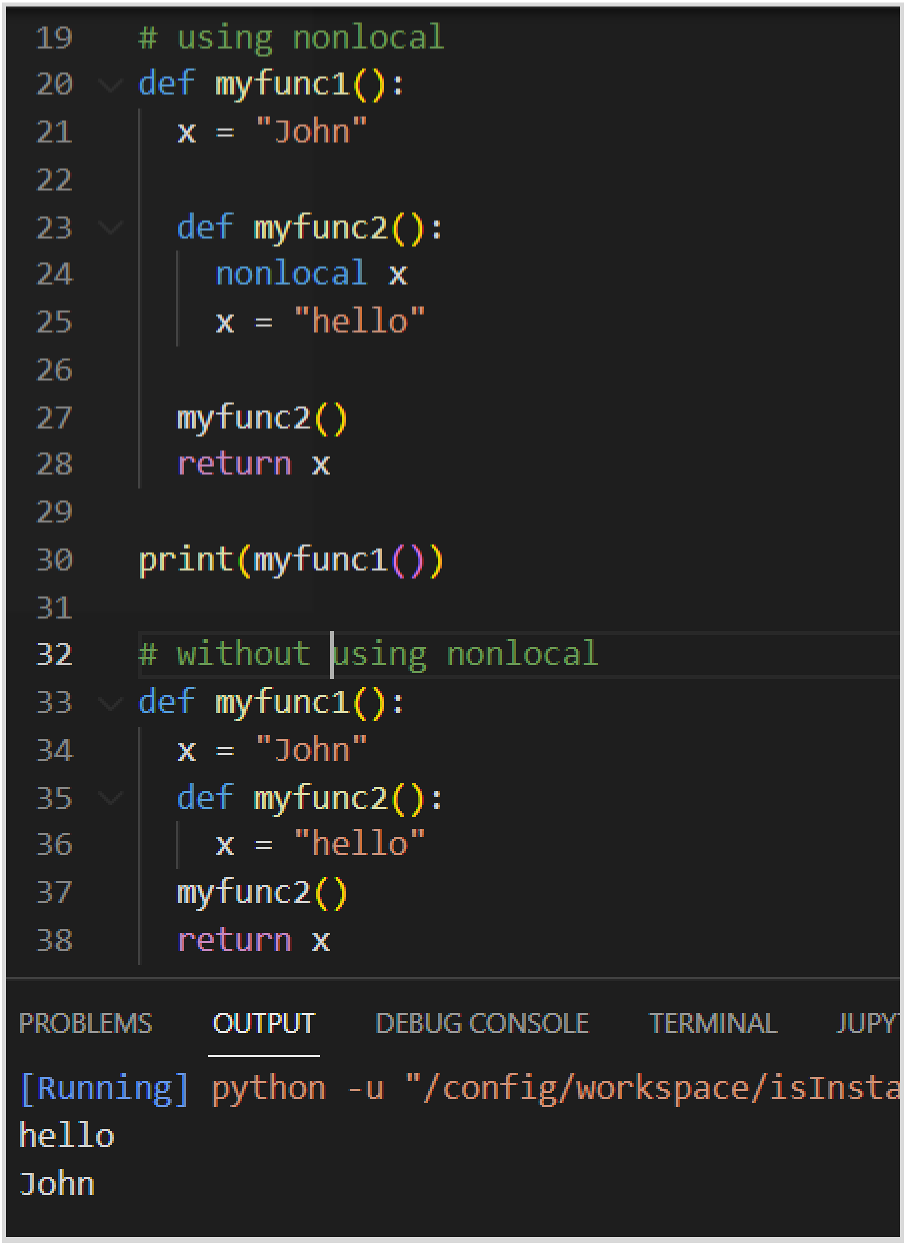
**issubclass(class, classinfo)**
Return true if class is a subclass (direct, indirect or virtual) of classinfo. A class is considered a subclass of itself.

```
class MyClass(object):
  pass

class MySubClass(MyClass):
  pass

obj_subclass = MySubClass()

print(isinstance(obj_subclass, MySubClass))
print(issubclass(MySubClass, MyClass))

print(isinstance(MySubClass, MyClass))
```

```
Output-
True
True
False
```

**Q34.Explain the use of the 'nonlocal' keyword in Python.**
A-
The nonlocal keyword is used to work with variables inside nested functions, where the variable should not belong to the inner function.

Use the keyword nonlocal to declare that the variable is not local.

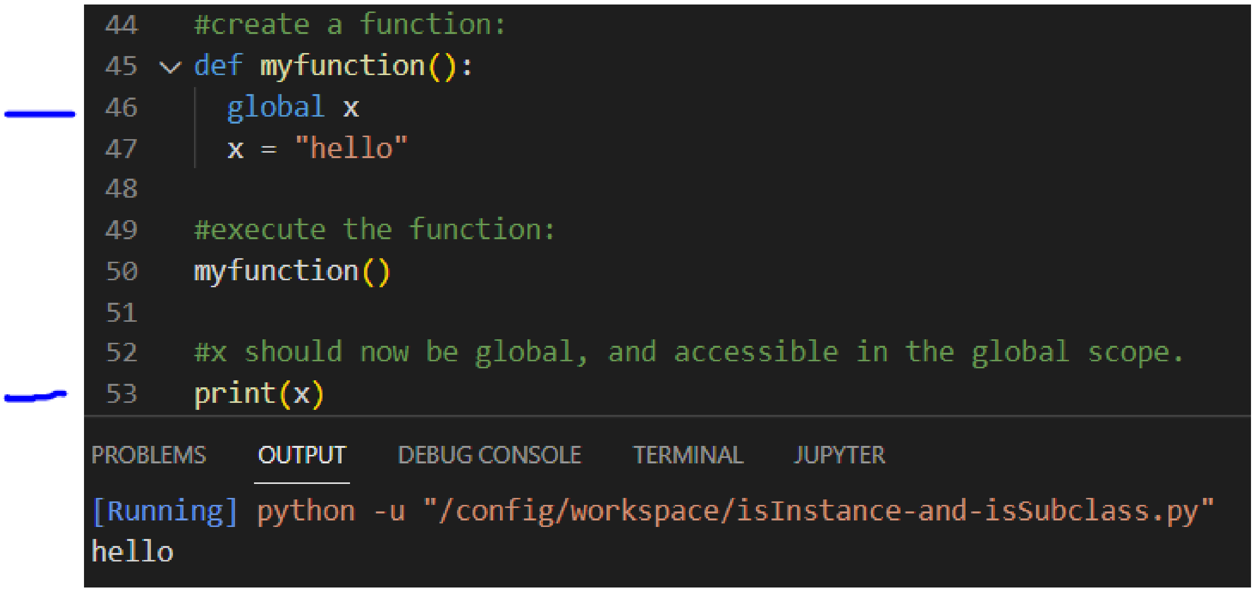


```python
# using nonlocal
def myfunc1():
    x = "John"

    def myfunc2():
        nonlocal x
        x = "hello"

    myfunc2()
    return x

print(myfunc1())

# without using nonlocal
def myfunc1():
    x = "John"
    def myfunc2():
        x = "hello"
    myfunc2()
    return x
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPY

```
[Running] python -u "/config/workspace/isInsta
hello
John
```

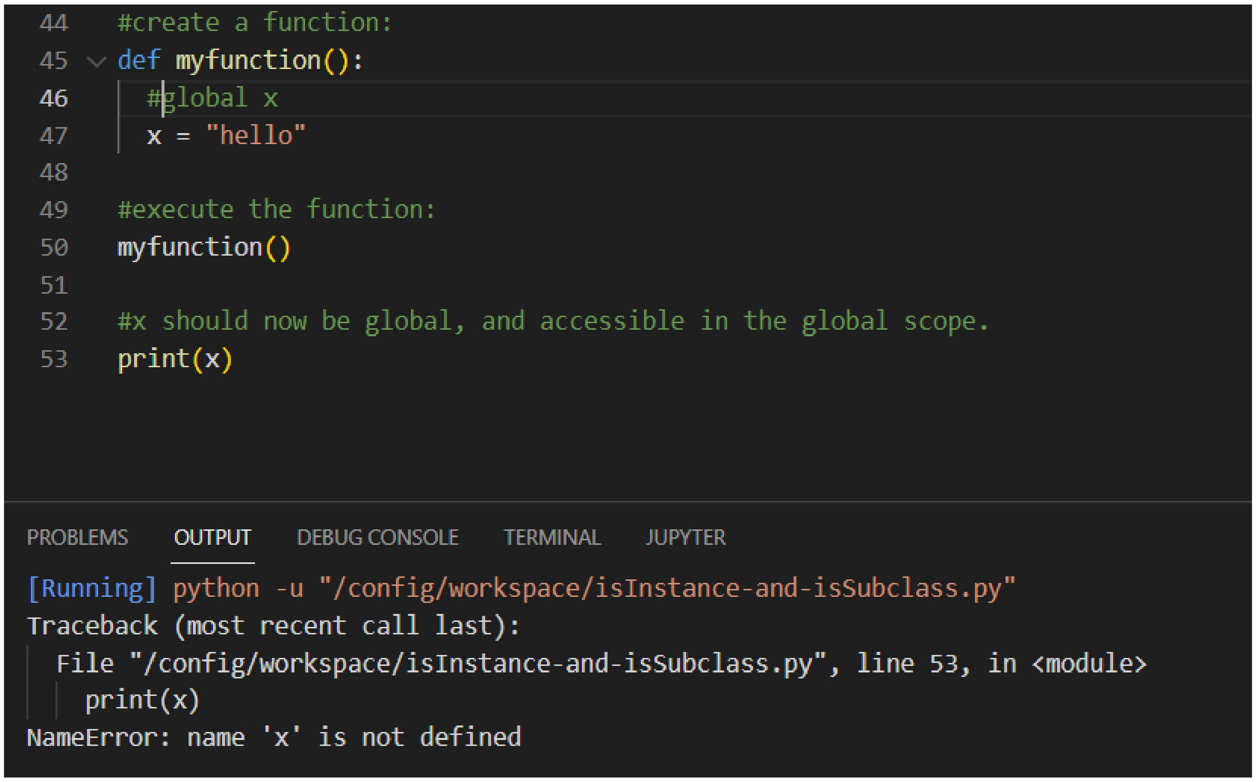
**Q35. What is the global keyword?**
A-
The global keyword is used to create global variables from a no-global scope, e.g. inside a function.



```python
#create a function:
def myfunction():
    global x
    x = "hello"

#execute the function:
myfunction()

#x should now be global, and accessible in the global scope.
print(x)
```

```
[Running] python -u "/config/workspace/isInstance-and-isSubclass.py"
hello
```

Without using global



```python
#create a function:
def myfunction():
    #global x
    x = "hello"

#execute the function:
myfunction()

#x should now be global, and accessible in the global scope.
print(x)
```

```
[Running] python -u "/config/workspace/isInstance-and-isSubclass.py"
Traceback (most recent call last):
  File "/config/workspace/isInstance-and-isSubclass.py", line 53, in <module>
    print(x)
NameError: name 'x' is not defined
```