Understanding Python Namespaces and Variable Scope

Namespaces

A namespace is a space that holds names(identifiers).Programmatically speaking, namespaces are dictionary of identifiers(keys) and their objects(values)

There are 4 types of namespaces:
* Builtin Namespace
* Global Namespace
* Enclosing Namespace
* Local Namespace

Scope and LEGB Rule (Local, Enclosing, Global, Builtin)

A scope is a textual region of a Python program where a namespace is directly accessible.

The interpreter searches for a name from the inside out, looking in the local, enclosing, global, and finally the built-in scope. If the interpreter doesn't find the name in any of these locations, then Python raises a NameError exception.

Understanding variable scope differences between local and global Python variables

Explanation

* The code demonstrates the fundamental concept of variable scope in Python where variables defined outside any function are global and accessible throughout the program
* Variables created inside a function are local to that function and cannot be accessed from outside the function scope
* The global variable 'a' is successfully printed after the function call because it exists in the global namespace
* The local variable 'b' is only accessible within the temp() function and would cause an error if referenced outside of it
* This illustrates how Python resolves variable names through the LEGB rule (Local, Enclosing, Global, Built-in) when looking up variable references

```
# Local and Global
```

Understanding the distinction between local and global variable scopes in Python

Explanation

* The variable `a` is defined globally with a value of 2.
* Inside the function `temp()`, a local variable `b` is created with a value of 3.
* The line `a = 4` creates a new local variable `a` that shadows the global `a`, meaning it does not affect the global variable.
* The function prints the local `a` (which is 4) and the local `b` (which is 3).
* After calling `temp()`, the global `a` remains unchanged and prints as 2.

```
# Local and Global -> same name
```

Output

```
4
3
```

Understanding the distinction between local and global variables in Python

Explanation

* The comment indicates that there is a difference between local and global variables in

Python.
* Local variables are defined within a function and are not accessible outside of it.
* Global variables are defined outside of any function and can be accessed from anywhere in the code.
* The snippet suggests that the local variable does not exist, while the global variable does, highlighting scope visibility.

```
# Local and Global -> local does not have but global has
```

Understanding the distinction between local and global variables in Python functions

Explanation

* The variable `a` is defined outside the function `temp()`, making it a global variable accessible throughout the module.
* Inside the function `temp()`, the variable `b` is defined, which is a local variable and can only be accessed within that function.
* The function `temp()` prints the values of both `a` and `b`, demonstrating that the global variable `a` can be accessed inside the function while the local variable `b` cannot be accessed outside of it.
* After calling `temp()`, the global variable `a` is printed again, confirming its value remains unchanged and accessible outside the function.

```
# Local and Global -> same name
```

Output

```
2
3
```

Understanding the Scope of Variables in Python with Global and Local Modifications

Explanation

* The variable `a` is defined as a global variable with an initial value of 2.
* Inside the function `temp()`, an attempt is made to modify `a` using `a = a + 1`, which will raise an `UnboundLocalError` because Python treats `a` as a local variable due to the assignment.
* The `print(a)` statement inside the function will not execute successfully, leading to an error before it can output any value.
* The final `print(a)` outside the function will correctly output the global variable's value, which remains 2, since the function did not successfully modify it.
* To modify the global variable within the function, the `global` keyword should be used before the variable name.

```
# Local and Global -> editing global
```

Output

```
---------------------------------------------------------------------------UnboundLocalE
rror                         Traceback (most recent call last)Cell In[8], line 9
                                                                                      6
```

Understanding the impact of global variable modification within a function in Python

Explanation

* The variable `a` is defined as a global variable with an initial value of 2.
* The function `temp()` declares `a` as a global variable, allowing it to modify the global `a`

instead of creating a local one.
* Inside the function, `a` is incremented by 1, changing its value from 2 to 3.
* The function prints the updated value of `a`, which is 3, before returning control to the main program.
* Finally, the global variable `a` is printed again, confirming its new value of 3, highlighting the risks of modifying global variables within functions.

```
# Local and Global -> editing global
```

Output

```
3
3
```

This code demonstrates the use of a global variable within a function to perform arithmetic operations.

Explanation

* A global variable `a` is defined with a value of 2, accessible throughout the module.
* The function `temp()` calculates a new variable `b` by adding 25 to the global variable `a`.
* The value of `b` is printed when the function is called, resulting in an output of 27.
* After calling `temp()`, the global variable `a` is printed, which remains unchanged at 2.
* This illustrates how global variables can be utilized within functions without modification.

```
# Local and Global -> editing global
```

Output

```
27
```

```
2
```

Understanding the use of global variables within a local function in Python

Explanation

* The function `temp()` defines a global variable `a` using the `global` keyword, allowing it to be accessed outside the function's scope.
* Inside the function, `a` is assigned the value of `1` and printed, which outputs `1` when the function is called.
* After calling `temp()`, the global variable `a` retains its value, allowing it to be printed again outside the function, resulting in the same output of `1`.
* This code demonstrates how local function definitions can modify global variables, impacting their accessibility and value throughout the program.

```
# Local and Global -> global created inside local
```

Output

```
1
```

```
1
```

Understanding Local and Global Variables in Python Functions

Explanation

* The function `temp(z)` takes a parameter `z`, which is a local variable within the function's scope.
* Inside the function, `print(z)` outputs the value of the local variable `z`, which is passed as an argument when the function is called.
* The variable `a` is defined globally with a value of `5`, but it is not affected by the function `temp(z)`.
* Calling `temp(5)` prints `5`, while `print(a)` outputs the global variable `a`, which is also `5`.

* This demonstrates the distinction between local and global variables, where changes to local variables do not impact global ones.

```
# Local and Global -> function parameter is local
```

Output

```
5
```

5

This code snippet demonstrates the use of Python's built-in scope to print a string to the console.

Explanation

* The `print()` function is a built-in function in Python that outputs text to the console.
* The string `'hello'` is passed as an argument to the `print()` function.
* When executed, this code will display the text "hello" in the terminal or console window.
* This snippet illustrates a basic usage of Python for outputting information.

```
# built-in scope
```

Output

```
hello
```

Displaying all built-in functions and variables in Python using the builtins module

Explanation

* The code imports the `builtins` module, which contains a collection of built-in functions, exceptions, and other objects available in Python.
* The `dir()` function is called with `builtins` as an argument, which returns a list of names in the module's namespace.

* The `print()` function outputs this list to the console, allowing users to see all built-in identifiers available for use in their Python environment.
* This is useful for developers to quickly reference built-in capabilities without needing to consult external documentation.

```
# how to see all the built-ins
```

Output

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup',
'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
'PythonFinalizationError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
'Warning', 'WindowsError', 'ZeroDivisionError', '_IncompleteInputError', '__IPYTHON__',
'__build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__',
'__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin',
'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod',
'enumerate', 'eval', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset',
'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'range', 'repr', 'reversed', 'round', 'runfile', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

This code snippet finds the maximum value in a list of integers.

Explanation

* A list `L` is created containing three integers: 1, 2, and 3.
* The `max()` function is called with the list `L` as its argument.
* The `max()` function evaluates the elements in the list and returns the highest value, which is

3.
* The result is printed to the console using the `print()` function.

```
L = [1,2,3]
```

Output

```
3
```

This code snippet demonstrates the renaming of a built-in function in Python.

Explanation

* The code defines a function named `max`, which overrides the built-in `max` function in Python.
* Inside the custom `max` function, it simply prints the string 'hello' when called.
* When the `max()` function is invoked, it executes the print statement instead of returning the maximum value from a list or iterable.
* This practice is generally discouraged as it can lead to confusion and errors in code that relies on the original functionality of the built-in `max` function.
* It highlights the importance of avoiding naming conflicts with built-in functions to maintain code clarity and functionality.

```
# renaming built-ins
def max():
```

Output

```
hello
```

This code snippet finds and prints the maximum value from a list in Python.

Explanation

* The `max()` function is used to determine the largest element in the iterable provided, which in this case is the list `L`.
* The result of `max(L)` is the highest value contained within the list.
* The `print()` function outputs this maximum value to the console.

* Ensure that `L` is not empty, as calling `max()` on an empty list will raise a `ValueError`.
* This code is useful for quickly identifying the peak value in a dataset represented as a list.

```
print(max(L))
```

Output

```
---------------------------------------------------------------------------TypeError
                            Traceback (most recent call last)Cell In[3], line 1
                                                                                ----> 1 pr
```

Understanding nested functions and their scopes in Python

Explanation

* The `outer` function defines a nested `inner` function, showcasing the concept of enclosing scope.
* The `inner` function is called within `outer`, allowing it to access its local scope and print "inner function".
* After calling `inner`, the `outer` function prints "outer function", demonstrating that it can execute its own code after the nested function call.
* Finally, the `outer` function is invoked, followed by a print statement in the global scope, which outputs "main program".
* This code illustrates the hierarchy of scopes: local (inner), enclosing (outer), and global (main program).

```
# Enclosing scope
```

Output

```
inner function
                                                                                outer func
```

Understanding variable scope in nested functions with Python's nonlocal

behavior

Explanation

* The `outer` function defines a local variable `a` with a value of 3 and contains a nested function `inner`.
* Inside `inner`, a new local variable `a` is defined with a value of 4, which shadows the `a` from the enclosing scope.
* When `inner` is called, it prints the local `a`, resulting in "inner 4".
* After `inner` completes, control returns to `outer`, which prints the enclosing `a`, resulting in "outer 3".
* Finally, the global variable `a` with a value of 1 is printed, showing the distinction between local, enclosing, and global scopes.

```
# Enclosing scope
```

Output

```
inner 4
outer 3
```

Understanding variable scope in nested functions in Python

Explanation

* The `outer` function defines a local variable `a` with a value of 3.
* Inside `outer`, the `inner` function is defined, which accesses the variable `a` from the enclosing scope of `outer`.
* When `inner` is called, it prints the value of `a` from the `outer` function, demonstrating access to the enclosing scope.
* After calling `inner`, `outer` prints its own `a`, confirming that it can access its local variable.
* Finally, the global variable `a` with a value of 1 is printed, showcasing the hierarchy of

variable scope: Local -> Enclosing -> Global -> Built-in.

```
# Enclosing scope
```

Output

```
inner 3
outer 3
```

Understanding variable scope and the order of resolution in nested functions

Explanation

* The code defines a nested function `inner()` inside the `outer()` function, demonstrating how Python handles variable scope.
* The variable `a` is defined in the global scope, allowing it to be accessed in both the `outer()` and `inner()` functions.
* When `inner()` is called, it prints the value of `a`, which is local to `inner()`, but since `inner()` does not have its own `a`, it looks for `a` in the enclosing scope (the `outer()` function).
* After calling `inner()`, the `outer()` function prints `a`, which is still accessible as it is in the enclosing scope.
* Finally, the global variable `a` is printed in the main program, illustrating the order of variable resolution: Local -> Enclosing -> Global -> Built-in.

```
# Enclosing scope
```

Output

```
inner 1
outer 1
```

Understanding the use of the nonlocal keyword in nested functions in Python

Explanation

* The `outer` function defines a variable `a` with a value of 3.
* The `inner` function attempts to modify `a` by adding 4, but it will raise an error because it tries to reference `a` before declaring it as nonlocal.
* The `print` statement in `inner` is intended to show the modified value of `a`, but it will fail due to the unbound local error.
* The `print` statement in `outer` correctly accesses the nonlocal `a`, printing its value of 3.
* The final `print` statement in the main program outputs the global variable `a`, which is set to 1.

```
# nonlocal keyword
```

Output

```
---------------------------------------------------------------------------UnboundLocalE
rror                         Traceback (most recent call last)Cell In[12], line 12
```

Understanding the Use of the Nonlocal Keyword in Nested Functions

Explanation

* The `outer` function defines a variable `a` initialized to 3 and contains a nested function `inner`.
* The `inner` function uses the `nonlocal` keyword to modify the variable `a` from the enclosing scope of `outer`, adding 4 to it.
* When `inner` is called, it prints the updated value of `a`, which becomes 7.
* After calling `inner`, the `outer` function prints the value of `a`, which reflects the change made by `inner`, showing 7.
* Finally, the global variable `a`, initialized to 1, is printed, demonstrating the scope hierarchy and the impact of the `nonlocal` keyword.

```
# nonlocal keyword
```

Output

```
inner 7
```

```
outer 7
```

Decorators

A decorator in python is a function that receives another function as input and adds some functionality(decoration) to and it and returns it.

This can happen only because python functions are 1st class citizens.

There are 2 types of decorators available in python

* `Built in decorators` like `@staticmethod`, `@classmethod`, `@abstractmethod` and `@property` etc
* `User defined decorators` that we programmers can create according to our needs

Demonstrating first-class functions in Python by assigning a function to a variable

Explanation

* The function `func` is defined to print the string 'Hello' when called.
* The variable `a` is assigned the function `func`, illustrating that functions can be treated as first-class citizens.
* Calling `a()` invokes the original `func`, resulting in 'Hello' being printed to the console.
* This showcases the ability to pass functions around as variables, enabling higher-order functions and callbacks.

```
# Python is a First class function -> We can do any operations
```

Output

```
Hello
```

This code demonstrates the deletion of a function and attempts to call it afterward.

Explanation

* The code defines a function `func()` that prints "Hello" when called.
* The `del func` statement deletes the reference to the function `func`, effectively removing it from the current namespace.
* The subsequent call to `func()` will raise a `NameError` because `func` no longer exists after deletion.
* This snippet illustrates the concept of first-class functions in Python, where functions can be assigned, passed, or deleted like any other object.

```
# Python is a First class function -> We can do any operations
```

Output

```
---------------------------------------------------------------------------NameError
                           Traceback (most recent call last)Cell In[18], line 7
                                                                              4
```

This code demonstrates how to pass a function as an argument to modify its behavior.

Explanation

* The `modify` function takes two parameters: a function `func` and a number `num`.
* It calls the passed function `func` with the argument `num` and returns the result.
* The `square` function computes the square of a given number by multiplying it by itself.
* When `modify(square, 2)` is called, it applies the `square` function to the number 2, resulting in 4.

```
def modify(func,num):
                                                                          return
```

Output

```
4
```

This code demonstrates the use of a decorator to enhance a function's output with additional print statements.

Explanation

* Defines a decorator function `my_decorator` that takes another function `func` as an argument.
* Inside `my_decorator`, a nested function `wrapper` is created, which prints asterisks before and after calling the original function.
* The decorator returns the `wrapper` function, effectively replacing the original function with the enhanced version.
* The `hello` function is defined to simply print "Hello".
* The decorator is applied to `hello`, and when the decorated function `a` is called, it outputs asterisks surrounding the "Hello" message.

```
# simple example
```

Output

```
**********
Hello
```

Understanding Python Decorators and Closures through a Simple Function Wrapper

Explanation

* Defines a decorator function `my_decorator` that takes another function as an argument and wraps it with additional behavior.
* The `wrapper` function prints asterisks before and after calling the original function, demonstrating how decorators can modify function behavior.
* Two functions, `hello` and `display`, are created to showcase the decorator's functionality by printing different messages.
* The decorator is applied to both functions, creating new callable objects `a` and `b`, which retain the original functions' behavior while adding the wrapper's print statements.
* Highlights the concept of closures, where the inner `wrapper` function retains access to its enclosing scope, allowing it to remember the original function even after execution.

```
# simple example
```

Output

```
**********
Hello
```

This code demonstrates the concept of closures in Python by encapsulating a variable within a nested function.

Explanation

* The `outer` function defines a variable `a` with a value of 5 and contains a nested function `inner`.
* The `inner` function, when called, prints the value of `a` from the enclosing scope of `outer`.
* The `outer` function returns the `inner` function, allowing it to be called later.
* When `b = outer()` is executed, `b` becomes a reference to the `inner` function.
* Calling `b()` executes the `inner` function, which accesses and prints the value of `a`, demonstrating closure behavior.

```
def outer():
    a = 5
```

Output

```
5
```

Closures in Python are like "memory-equipped" functions. They allow a function to remember values from the environment in which it was created even if that environment no longer exists. Closures are used in functional programming, event handling and callback functions where you need to retain some state without using global variables.

How Closures are Created

A closure is

A function is defined inside another function (nested function).

The inner f

Understanding closures in Python through nested functions and variable retention

Explanation

* The `outer_function` takes a parameter `x` and defines an inner function `inner_function` that takes another parameter `y`.
* The inner function returns the sum of `x` (from the outer function) and `y`, demonstrating how closures can capture variables from their enclosing scope.
* When `outer_function` is called with `x = 10`, it returns the `inner_function`, effectively

creating a closure that retains the value of `x`.
* The closure can then be called with different values of `y`, allowing it to compute results based on the retained value of `x`.
* In the provided calls to `closure`, it outputs `15` for `closure(5)` and `30` for `closure(20)`, showcasing the closure's functionality.

```
def outer_function(x):
                                                    # Oute
```

Output

```
15
                                                    30
```

This code demonstrates the use of a decorator to enhance a function's output with additional formatting.

Explanation

* Defines a decorator function `my_decorator` that takes a function `func` as an argument.
* Inside the decorator, a nested function `wrapper` is created, which adds pre- and post-execution print statements.
* The original function `func` is called within the `wrapper`, allowing it to execute while being wrapped by the additional functionality.
* The decorator is applied to the `hello` function using the `@my_decorator` syntax, modifying its behavior when called.
* When `hello()` is invoked, it prints decorative lines before and after the original "Hello" message.

```
# Better syntax
```

Output

```
**********
```

```
Hello
```

This code implements a decorator to measure the execution time of functions in Python.

Explanation

* The `timer` function is a decorator that wraps another function to measure its execution time.
* Inside the `wrapper` function, the current time is recorded before and after the execution of the original function.
* The time taken for the function to execute is calculated and printed to the console.
* The `@timer` syntax is used to apply the decorator to the `hello` and `display` functions, allowing their execution time to be measured when called.
* When `hello()` and `display()` are invoked, the output includes the execution time for each function.

```
import time
```

Output

```
hello
time taken
```

This code implements a decorator to measure the execution time of functions in Python.

Explanation

* The `timer` function is a decorator that wraps another function to measure its execution time.
* Inside the `wrapper` function, the current time is recorded before and after the execution of the wrapped function.
* The time taken for the function to execute is calculated and printed to the console.
* The `@timer` syntax is used to apply the decorator to the `hello`, `display`, and `square` functions.
* The `square` function does not print its result, so its execution time will be displayed but not its output.

```
import time
```

Output

```
hello
                                                                                time taken
```

This Python code implements a decorator to measure and display the execution time of functions.

Explanation

* The `timer` function is a decorator that wraps another function to measure its execution time.
* It uses the `time` module to capture the start time before calling the wrapped function.
* After the function execution, it calculates the elapsed time and prints it along with the function name.
* The `@timer` syntax is used to apply the decorator to the `hello`, `display`, and `square` functions.
* Each decorated function will output its execution time when called, providing insights into performance.

```
import time
```

Output

```
hello
```

```
time taken
```

This function calculates the square of a number but may raise an error for non-numeric inputs.

Explanation

* The `square` function takes a single argument `num` and returns its square by multiplying it by itself.
* When called with a numeric input like `2`, it correctly returns `4`.
* If called with a non-numeric input such as a string (`'hehehe'`), it will raise a `TypeError` since multiplication is not defined between a string and an integer.
* This code demonstrates the importance of input validation when performing mathematical operations.

```
def square(num):
```

```
    return
```

Output

```
---------------------------------------------------------------------------TypeError
                          Traceback (most recent call last)Cell In[40], line 5
```

```
      2
```

Implementing decorators with arguments to enforce data type checks in Python functions

Explanation

* Defines a decorator `sanity_check` that takes a `data_type` argument to validate the type of input for the decorated function.
* The `outer_wrapper` function wraps the target function, while `inner_wrapper` checks if the first argument matches the specified `data_type`.
* If the type matches, the original function is called; otherwise, a `TypeError` is raised with a descriptive message.
* The `@sanity_check(int)` and `@sanity_check(str)` decorators are applied to the `square` and `greet` functions, respectively, enforcing type checks on their inputs.
* The code demonstrates the functionality by calling `square(2)` and `greet('madhu')`, which succeed, while `square('hehehe')` raises a `TypeError` due to a type mismatch.

```
# decorators with arguments
```

Output

```
4
```

```
hello madh
```

Namespace and Scope

###`Q1:` Write `Person` Class as given below and then display it's namespace.

```
Class Name - Person
```

This Python code defines a Person class with private attributes and methods for managing personal information.

Explanation

* The `Person` class initializes with `name` and `state` attributes, while `city` and `age` are private attributes initialized to `None`.
* Setter methods (`set_city` and `set_age`) allow controlled modification of the private attributes `__city` and `__age`.

* Getter methods (`get_city` and `get_age`) provide access to the private attributes, ensuring encapsulation.
* The `address` method formats and returns a string representation of the person's address using their name, city, and state.
* The final loop iterates over the class's dictionary to print all attributes and methods defined in the `Person` class.

```
#Write your code here
```

Output

```
__module__
__firstlin
```

###`Q2:` Write a program to show namespace of object/instance of above(Person) class.

This code demonstrates the creation and manipulation of a Person object in

Python.

Explanation

* A `Person` object named `p` is instantiated with the name "Madhu" and state "AP".
* The `set_city` method is called to update the city attribute of the `Person` object to "Vizag".
* The `set_age` method is used to set the age of the person to 30.
* The `address` method is called to print the address of the person, which is likely a formatted string based on the object's attributes.
* A loop iterates over the `__dict__` attribute of the `Person` object, printing each attribute name, which provides insight into the object's internal state.

```
# Write your code here
```

Output

```
Address:  Madhu, Vizag, AP
```

```
name
```

###`Q3:` Write a recursive program to to calculate `gcd` and print no. of function calls taken to find the solution.

```
gcd(5,10)
```

This code calculates the greatest common divisor (GCD) using recursion and tracks the number of recursive calls.

Explanation

* The `gcd` function computes the GCD of two integers `a` and `b` using the Euclidean algorithm.
* A global variable `counter` is incremented with each recursive call to count how many times the function is invoked.
* The base case for recursion is when `b` equals zero, in which case it returns `a` as the GCD.
* If `b` is not zero, the function calls itself with `b` and the remainder of `a` divided by `b`.

* Finally, the result of the GCD calculation and the total count of recursive calls are printed.

```
#Write your code here
```

```
# O(log(mi
```

Output

```
5 2
```

Itterator And Generator

###`Q4:` Create MyEnumerate class,

Create you
need to ret
(starting wi
structure. T

```
for index, letter in MyEnumerate('abc'):
```

```
    print(
```

Output:

```
0 : a
```

Custom enumeration class in Python to iterate over a sequence with index tracking

Explanation

* Defines a class `MyEnumerate` that mimics the behavior of Python's built-in `enumerate` function.
* The `__init__` method initializes the object with the data to be enumerated and sets the starting index to 0.

* Implements the `__iter__` method to return the iterator object itself, allowing it to be used in a loop.
* The `__next__` method retrieves the current index and corresponding value from the data, increments the index, and raises `StopIteration` when the end of the data is reached.
* The for loop demonstrates the usage of `MyEnumerate` to print each character in the string 'abc' along with its index.

```
#Write your code here
```

Output

```
0 : a
1 : b
```

###`Q5:` Iterate in circle

Define a cl
idea is tha
greater tha
an another

```
c = Circle('abc', 5)
d = Circle
```

Output

```
[a, b, c,
```

This code defines a circular iterator that cycles through a given data set a

specified number of times.

Explanation

* The `Circle` class initializes with a data set and a maximum number of iterations.
* The `__iter__` method returns the iterator object itself, allowing it to be used in a loop.
* The `__next__` method retrieves the next value from the data set, cycling back to the start if the end is reached, and raises `StopIteration` when the maximum iterations are exceeded.
* Two instances of `Circle` are created with different maximum iterations, demonstrating how the iterator behaves with varying limits.
* The `print` statements convert the iterators to lists, showing the output of the circular iteration for each instance.

```
#Write your code here
```

Output

```
['a', 'b', 'c', 'a', 'b']
```

###`Q6:` Generator time elapsed

Write a ge
will return
many seco
next item fr

Note that the timing should be relative to the previous iteration, not when the

generator w

```
for t in elapsed_since('abcd'):
    print(
```

Output:

```
(0.0, 'a')
```

Note: Your output may differ because of diffrent system has different processing configuration.

This code measures and yields the time elapsed since the last iteration for each item in a sequence.

Explanation

* The `elapsed_since` function uses `time.perf_counter()` to track high-resolution time intervals.
* It iterates over each item in the provided `data`, calculating the time difference (`delta`) from the last recorded time.
* The function yields a tuple containing the elapsed time and the current item, allowing for real-time tracking of intervals.
* In the loop, each item is printed with a 2-second pause (`time.sleep(2)`), demonstrating the elapsed time for each iteration.
* This approach is useful for performance monitoring or timing events in a sequence.

```
#Write yor code
```

Output

```
(5.00003807246685e-07, 'a')
```

```
(2.0007447
```

Decorators

###`Q7:` Write a Python program to make a chain of function decorators (bold, italic, underline etc.) on a given function which prints "hello world"

```
def hello():
    return
```

```
bold - wrap string with <b> tag. <b>Str</b>
italic - w
```

This code demonstrates the use of decorators to format text with HTML tags in Python.

Explanation

* The code defines three decorators: `make_bold`, `make_italic`, and `make_underline`, each of which wraps a function to add corresponding HTML tags around its output.
* Each decorator defines an inner function `wrapped` that calls the original function and modifies its return value by adding the appropriate HTML formatting.
* The `hello` function is decorated with all three decorators, meaning its output will be wrapped in bold, italic, and underline tags when called.
* The final output of `print(hello())` will be the string "hello world" enclosed in `<b><i><u></u></i></b>` tags, resulting in a bold, italicized, and underlined text when rendered in HTML.

```
#Write your code here
```

Output

```
<b><i><u>hello world</u></i></b>
```

###`Q8:` Write a decorator called `printer` which causes any decorated function to print their return values. If the return value of a given function is `None`, printer should do nothing.

Python decorator that prints function return values with preserved function metadata

Explanation

* The code defines a decorator called `printer` that wraps around functions to automatically print their return values
* The `@wraps(func)` decorator preserves the original function's metadata like name and docstring when creating the wrapper function
* When `hello("hello")` is called, the decorator intercepts the function call, executes it, and prints the return value "hello"
* The decorator checks if the return value exists before printing, avoiding unnecessary output for functions that return None
* The `inner` function accepts any number of positional and keyword arguments using `*args` and `**kwargs` to make the decorator flexible for different function signatures

```
#Write your code here
```

Output

```
hello
```

###`Q9:` Make a decorator which calls a given function twice. You can assume the functions don't return anything important, but they may take arguments.

```
#Lets say
```

Output:

```
hello
```

This code defines a decorator that calls a function twice when invoked.

Explanation

* The `double` function is a decorator that takes another function `func` as an argument.
* Inside `double`, an inner function `inner` is defined, which calls `func` twice with the same arguments.
* The `@wraps(func)` decorator is used to preserve the metadata of the original function `func`.
* The `hello` function, when decorated with `@double`, will print its input string two times when called.
* The call `hello("hello")` results in "hello" being printed twice to the console.

```
#Write your cod here
```

Output

```
hello
```

```
hello
```

`Q10:` Write a decorator which doubles the return value of any function. And test that decoratos is working correctly or not using `asert`.

```
add(2,3) -> result in 10. Without decorator it should be 5.
```

This code demonstrates a decorator that doubles the result of a function.

Explanation

* The `double` function is a decorator that takes another function `func` as an argument and returns a new function `inner`.
* Inside `inner`, the original function `func` is called with any arguments and keyword arguments, and its result is multiplied by 2 before being returned.
* The `add_withDeco` function is decorated with `@double`, meaning its return value will be doubled when called.
* The `add` function simply returns the sum of two numbers without any modification.
* An assertion checks if doubling the result of `add` matches the output of `add_withDeco`, and if they match, it prints a confirmation message.

```
# Write your code here
```

Output

```
Values are matching
```