**THEORY QUESTIONS ASSIGNMENT**

Software Stream



KEY NOTES

* This assignment to be completed at student’s own pace and submitted before given deadline.
* There are 10 questions in total and each question is marked on a scale 1 to 10. The maximum possible grade for this assignment is 100 points.
* Students are welcome to use any online or written resources to answer these questions.
* The answers need to be explained clearly and illustrated with relevant examples where necessary. Your examples can include code snippets, diagrams or any other evidence-based representation of your answer.

| **Theory questions** | **10 point each** |
| --- | --- |

1. How does Object Oriented Programming differ from Process Oriented Programming?

Process Oriented Programming (POP) is the programming model whose structure originates in structured programming which has basis in the concept of calling procedure. Procedures can be defined as an ordered sequence of computational steps to be executed. Whilst a program is being executed, at any point a given procedure may be called, including by other procedures or itself.

Examples of POP are FORTRAN, C, Basic, Pascal etc.

Object Oriented Programming (OOP) is a programming model whose structure is based on the concept of objects. Data is contained within objects in the form of attributes whilst code is contained in the form of methods.

Examples of OOP are C#, Java, C++, Python, etc.

In OOP, programs are split into small segments named *objects* whilst in POP, programs are split into small segments named *functions.*

The *bottom up approach* is adopted by OOP whilst the *top down approach* is adopted by POP.

There are access specifiers like public, private, protected etc. in OOP, whilst no such thing exists in POP. There are no access specifiers in POP.

Adding new data and function in OOP is much easier than in POP.

There is functionality for data hiding provided in OOP thus it is more secure. In contrast, there is no proper method for data hiding provided by POP thus it is less secure.

It is possible to achieve overloading in OOP, meanwhile overloading is not possible in POP.

Data holds more significance than function in OOP, whilst function holds more significance than data in POP.

The basis of OOP is from the *real world* whilst the basis of POP is from the *unreal world*.

**SOURCES:**

* https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/#:~:text=In%20procedural%20programming%2C%20program%20is,into%20small%20parts%20called%20objects.&text=Adding%20new%20data%20and%20function%20is%20easy.,so%20it%20is%20less%20secure.

1. What's polymorphism in OOP?

Polymorphism is one of the 4 basic concepts from OOP. Polymorphism refers to an object being able to be of many forms.

In the context of parent and child classes, any class reference can have a child class assigned to it in its parent hierarchy, as well as itself.

There are 2 types of polymorphism which are frequently referenced: Dynamic Polymorphism and Static Polymorphism.

Dynamic Polymorphism refers to the existence of many forms occurring in different classes.

To explain simply, say there methods of the same method signature in classes within a class hierarchy, these methods exist in different forms i.e. method overriding. When a class reference has an object assigned to it and when a method of the object is called, the method from the object’s class will be executed - not the method found in the reference class (if the reference is a parent class) .

As at run-time is when the object is created, the form of method to be executed can only be decided at run-time.

Static Polymorphism is where the existence of many forms is occurring in the same class. To explain it simply, this is where there is more than 1 method with the same method name but containing different arguments (return type may or may not be identical). When the methods compiler is called, the compiler will decide which method to call, determined by the parameters passed when calling. This occurs at compile-time.

**SOURCES:**

* https://medium.com/@shanikae/polymorphism-explained-simply-7294c8deeef7

1. What's inheritance in OOP?

Inheritance is 1 of the 4 basic concepts of OOP; concepts which serve to optimize work performed by programmers. Inheritance permits the creation of class hierarchies where classes and objects inherit properties (attributes) and behaviours (methods) from their parent (or super) class. A subclass or child class refers to a class inheriting from a parent (or super) class. Child objects refer to objects that through inheritance receive properties and behaviors from a parent.

One of the most important factors of the usefulness of inheritance is the resulting reusability of code, and is important with the coding principle of DRY (Don’t Repeat Yourself). Inheritance provides a means for child classes which share the same attributes or methods to all inherit from a parent class, rather than repeating the same code in all of the child classes.

Another important aspect of the usefulness of inheritance is the limiting of the possibility of human error permitted by inheritance. Errors that might otherwise have been propagated by manually inputting these shared functions into every child class, can be avoided through having all the code easily accessible from the parent class where the software engineer can easily work through and debug errors from 1 place rather than every single child class sharing the same functions.

**SOURCES:**

* https://medium.com/@andrewkoenigbautista/inheritance-in-object-oriented-programming-d8808bca5021

1. If you had to make a program that could vote for the top three funniest people in the office, how would you do that? How would you make it possible to vote on those people?

I would create a program where the user was presented with a list for all the people in the office, and I would ask them to rank the top 3 funniest people in the office where 1 is the funniest and 3 is the least funny of the 3. By assigning numbers to the people; they would be voting for them.

You can use dictionaries to achieve this and then nest the dictionaries. I would create a dictionary for each individual’s response.

I would then use the Python collections counter module to count up all the people, where 1st place would be the most who received the most 1’s, 2nd place would be the most who received the most 2’s and 3rd place would be the person who received the most 3’s.

It would be computationally expensive but I would go through the values of the nested dictionary, which correspond to answers of the funniest people. I would then create a mapping of the values of the nested dictionary to count the occurrences of 1, 2 and 3 - you would have then figured out the 3 funniest people in the office by collating the data. I only had time to code the first part; I didn’t have time to code the second part (the mapping). However, it is by mapping or by other means.

It is possible to map multiple values to a key [https://www.oreilly.com/library/view/python-cookbook/0596001673/ch01s06.html], I would then map the multiple rankings to each key which was the person from the office. The keys would be the person in the office and they would have multiple values representing their rankings.

It is then possible to count the frequency of 1, 2 and 3 for each individual, and then calculate the 3 funniest people. You could this by creating another dictionary with all these unique values and calling the max() function to find the maximum and find the funniest person.

Alternatively, the keys could be 1, 2 and 3 and the values could be the individuals. By mapping all the individuals to the 1, 2 and 3 keys - you can then use the Counter method to count the names which occur the most frequently for each of the 1, 2 and 3 keys. You would then be able to to find the 3 funniest people in the office.

| # example program  from collections import Counter, ChainMap  # present list of people from the office  # this is to be answered by each individual  list\_people = ['Nasian', 'Claire', 'Chizu', 'Sophie', 'Nikita', 'Georgia']  print('For the following questions, please only use names supplied in {}'.format(list\_people))  funny\_1 = input("Who do think is the funniest person in the office? ").lower()  funny\_2 = input("Who do think is the second funniest person in the office? ").lower()  funny\_3 = input("Who do think is the third funniest person in the office? ").lower()  dict\_funny = {}  dict\_funny.update({funny\_1: 1, funny\_2: 2, funny\_3: 3})  # this is data from somebody else's ranking  dict\_funny\_2 = {'nasian': 1, 'claire': 2, 'chizu': 3}  nested\_dict = {'user\_1\_funny\_list': dict\_funny, 'user\_2\_funny\_list': dict\_funny\_2}  nested\_dict\_values = (nested\_dict.values())  # once you have collected all the data from everyone, use counter to calculate people's answers  # iterate through the values of the nested dictionary to find the frequency of 1, 2, 3 |
| --- |

1. What's the software development cycle?

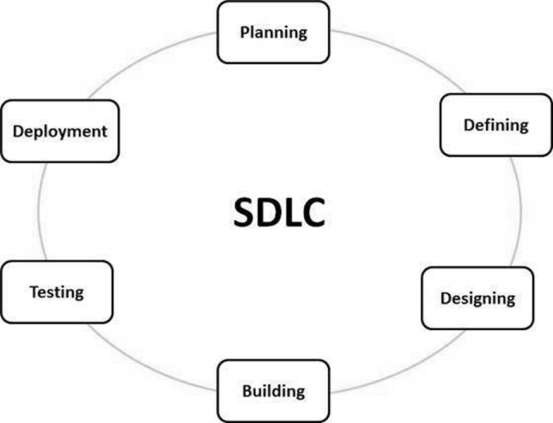
The Software Development Life Cycle (SDLC) serves to design, develop and test high quality softwares, and is a process that is utilized by the software industry. The objectives of the SDLC are to create a high-quality software that satisfies or surpasses customer expectations, capable of reaching completion within times and cost estimates.

SDLC is otherwise known as the Software Development Life Cycle and is a framework that defines tasks executed at each stage in the software development process. For software life-cycle processes, **ISO/IEC 12207** is the international standard. It has the objective of being the standard defining all tasks necessary for developing and maintaining software.

So how does the SLDC work, you ask?

In a software organization, for a software project the process that is followed is the SLDC. It contains a detailed plan that outlines the development, maintenance, replacement and alterations or enhancements of the specific software. The methodology for the improvement of the quality of the software and the overall development process is defined within the life cycle.

The following Figure illustrates the stages of the SDLC well:



*Figure 1.* The stages of the SDLC. Taken from [<https://www.tutorialspoint.com/sdlc/sdlc_overview.htm>]

The following stages of the SDLC are:

**Stage 1: Planning and Requirement Analysis**

This stage is executed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. The basic project approach is then planned with this information, as well as the product feasibility study in the economical, operational and technical regions.

**Stage 2: Defining Requirements**

The product requirements are defined and documented after the completion of the requirement analysis. The product requirements will then be sent for approval from the customer or the market analysts. This is all executed through an **SRS (Software Requirement Specification)** document.

**Stage 3: Designing the Product Architecture**

Using the SRS, a DDS - Design Document Specification documents the proposed design approaches for the product architecture.

**Stage 4: Building or Developing the Product**

The commencing of the development and the building of the product occurs during this stage of the SDLC.

**Stage 5: Testing the Product**

In modern SDLC models, this stage is generally a subset of all the stages as testing is usually necessary in all stages of the SDLC. This stage refers solely to testing the product.

**Stage 6: Deployment in the Market and Maintenance**

The product is formally released in the suitable market at this stage, after it has been tested and certified by those necessary as ready to be deployed.

After receiving feedback, the product can possibly be released as it is or it may be appropriately improved for the target market demographic. The maintenance for the existing customer base is finished once the product has been released in the market.

My own example to understand this: Based on this information, I would liken this to the release of software updates for Android OS (which is what I use), whereby updates to the software are released and based on feedback - changes or improvements can be made.

**SOURCES:**

* https://www.tutorialspoint.com/sdlc/sdlc\_overview.htm

1. What's the difference between agile and waterfall?

Waterfall and Agile are 2 unique methodologies regarding the process of the completion or work items or projects.

Agile consists of a set of principles and values, witten by 17 leaders in technology in 2001. The methodology of Agile is iterative in nature and it contains both collaborative and cyclic processes. The product delivery methods and frameworks within Agile include those such as Six Sigma, Lean, Scrum and Kanban. The methods and frameworks found within Agile serve to source a solution that is both flexible and iterative, so that it may be adapted according to demands resulting from the situation.

The methodology of waterfall is sequential in nature where it is possible to be collaborative, however, the process of handling tasks is much more linear. Waterful is much more linear than Agile and has a primary focus on up front planning with requirements fully defined before the start of a project. As an ode to its name, work cascades, akin to a waterfall, through different project phases. The completion of each phase is required before the next phase can start.

When comparing the 2, Agile is much more flexible and facilitates the opportunity for stakeholders and team members to observe, test and contribute throughout the process thus the project can evolve and change in time to meet changing demands and necessary requirements. The principles of collaboration, teamwork, accountability and self-organisation are encouraged through the Agile process thus helping with the general commitment and overall motivation to the goals and outcomes of a project.

However, the flexibility of Agile can become problematic as there is a risk of resulting in issues in regions such as organization due to priorities shifting due to changes not being appropriately managed or not communicated succinctly.

For well-defined projects, Waterfall is especially efficient. Planning and design becomes much easier as project stakeholders agree upfront of the delivery of the project. The much more linear approach of Waterfall is so that team members are required to only be available for their specific required project phases and thus are able to focus in other regions.

However, for more complex or longer-term projects the upfront comprehensive requirements required of Waterfall can become very challenging. Waterfall is much more rigid than Agile due to its sequential nature and reliance on pre-planning thus it is very difficult to evolve and make changes to the project whilst it is progressing.

**SOURCES:**

* <https://www.forecast.app/faqs/what-is-the-difference-between-agile-and-waterfall>
* https://www.aipm.com.au/blog/agile-vs-waterfall-whats-the-difference

1. What is a reduced function used for?

The **reduce(function, iterable[, initializer])** function can take in 3 arguments, where the first 2 arguments **(function, iterable)**  are compulsory to give. The function argument itself takes in 2 arguments and the iterable argument consists of an iterable such as a list or tuple. The third optional argument is the **initializer**, the default value for this argument is **None**. If a value is to be passed into the initializer then this value will be utilized as the first *x* value by the reduce function - rather than the first element of the iterable being the first *x* value. All subsequent steps i.e. all the steps after this first x value, will be the same i.e. what the result of the code would have been without the initializer. Before the elements of our iterable in the calculation, is where the initializer value is placed.

The purpose of the **reduce(fun,seq)** function is to apply a specific function passed in the argument to all of the list elements contained in the sequence being passed into the argument. This definition of this function is found in the “functools” module.

The **reduce()** function will store the immediate results but will only return the final summation value. The **reduce()** function is then a very useful function for the purposes of applying a function to an iterable so as to reduce it to a single cumulative value. The concept behind this **reduce()** function is to take an existing function which will subsequently be applied cumulatively to all the items in an iterable so that a single value can be formed.

The **reduce()** function is very useful as it allows for the processing of iterables without the explicit use of for loops, which are computationally expensive. As the **reduce()** function is written in C, its internal loop can be faster than an explicit Python *for* loop.

**SOURCES:**

* <https://www.geeksforgeeks.org/reduce-in-python/>
* <https://realpython.com/python-reduce-function/>
* https://towardsdatascience.com/using-reduce-in-python-a9c2f0dede54

1. How does merge sort work?

Merge Sort is a Divide and Conquer algorithm. The input array is divided into 2 halves, the algorithm then calls itself for the 2 halves and subsequently merges the 2 sorted halves. The merging of the two halves occurs through the use of the **merge()** function.

The following is the C implementation [taken from <https://www.geeksforgeeks.org/merge-sort/>] :

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

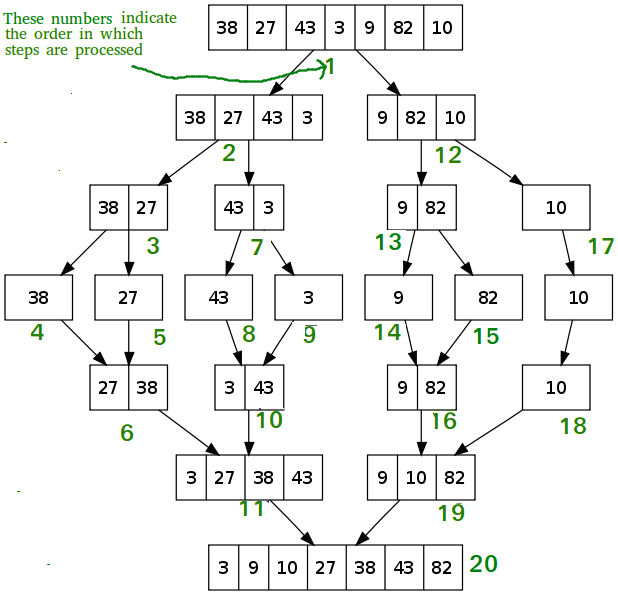
**3.** Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following image is taken from [<https://www.geeksforgeeks.org/merge-sort/>] which took it from Wikipedia - it demonstrates the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.



The Merge Sort algorithm is a recursive algorithm and so a recurrence relation can be used to describe the time complexity: T(n) = 2T(n/2) + θ(n). The solution of the recurrence is θ(nLogn) and so the time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as the array is always divided into 2 halves by merge sort and the merging of the 2 halves takes linear time.

Key applications of the Merge Sort algorithm include the sorting of linked lists in O(nLogn) time, Inversion Count Problem and External Sorting.

The primary cons of the Merge Sort algorithm include:

* For smaller tasks, compared to other sort algorithms; Merge Sort algorithm is slower
* For the temporary array, an additional memory space of 0(n) is required for the Merge Sort algorithm
* Even in cases where the array is sorted, the Merge Sort algorithm will go through the entire process

**SOURCES:**

* https://www.geeksforgeeks.org/merge-sort/

1. Generators - Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop. What is the use case?

An iterator must have the magic methods: **\_\_iter\_\_** and **\_\_next\_\_**, so when creating a generator function these two magic methods must be included.

A reason to use generator functions over an iterator is to save memory space. Iterators compute data by lazy evaluation and so they only compute the value when requested, rather than computing the value of each item at time of instantiation.

For really large data sets, this is really useful as it is possible to then immediately begin to use the data whilst the entire data set is in the process of being computed. However with small data sets, it can be much more useful to use generator functions so that the values are computed at the point of instantiation - small data sets will not take a long time to be processed unlike big data sets therefore generator functions become very useful.

When computing data with generators over iterators, it is much more simple to compute the data with the generator functions than with iterators. The **yield** statement is introduced when using generator functions; it is similar to the **return** command as the **yield** command returns a value. However, unlike the **return** command, the **yield** command will save the state of the function. When the function is next called, execution will **continue from the point at which it left off**, with the **same variable values** it had before yielding.

An even better way of dealing with data than generator function is generator expressions - which are synonymous with list/dictionary comprehensions. Generator expressions' functionality is identical to that of a list comprehension, where instead of surrounding an expression with [] the expression is surrounded with ().

**To summarise:**

**Lazy evaluation** is permitted by **iterators** and thus the next element of an iterable object is only generated upon request, which is particularly useful for **large data sets**.

**Generator functions** permit the creation of functions that behave as **iterators** and thus in a very **‘pythonic’** style. It is generally considered that **generator functions** are **better** than **iterators**. For **simple cases**, **generator expressions** are considered to be better than **iterators**.

Both **generators** and **iterators** are permitted to only be **iterated over once**. If the user attempts to iterate over a sequence once it has been exhausted with either iterators/generators - **no value** will be returned/yielded. The behaviour will be that of an empty list.

The following is code I wrote after our iterators class to practice writing generator functions and demonstrates the functionality of generator functions. The class has magic methods **\_\_iter\_\_** and **\_\_next\_\_**, and the function makes use of the **yield** statement.

| *'''*  *Practicisng the coding problem from Corey Schafer's YouTube channel on creating your own iterators*  *'''*  # creating an iterable with a class  class Sentence:  # i originally used str as my second argument but this is bad practice as str is how you declare a string in python  def \_\_init\_\_(self, sentence):  self.sentence = sentence  self.n = 0  self.seperate = self.sentence.split()  self.max\_len = len(self.seperate)  # you need the \_\_iter\_\_ and \_\_next\_\_ methods to create an object that is an iterable  def \_\_iter\_\_(self):  return self  def \_\_next\_\_(self):  if self.n >= self.max\_len:  raise StopIteration  index = self.n  result = self.seperate[index]  self.n += 1  return result  # creating an iterable with a generator function  def my\_gen(sentence):  n = 0  seperate = sentence.split()  max\_len = len(seperate)  while n < max\_len:  # yield is the keyword needed to make this a generator  # use of the yield gives this \_\_iter\_\_ and \_\_next\_\_ methods without having to write out all that ugly notation  yield seperate[n]  n += 1  # iterable using the class method  my\_sentence = Sentence('This is a test')  for word in my\_sentence:  print(word)  print('\n BREAK \n')  # iterable using the generator method  sentence\_2 = my\_gen('This is a test')  for word in sentence\_2:  print(word) |
| --- |

The following code is code I wrote during our class to do with itertools to demonstrate how generator functions differ with the itertools cycle. When using the generator functions, once you have passed through it, it will be exhausted and you will get an error if you try to go through it again. However, when using the itertools cycle you overcome this issue and you are able to cycle through.

| # generator function I created yesterday  def my\_gen(sentence):  n = 0  seperate = sentence.split()  max\_len = len(seperate)  while n < max\_len:  # yield is the keyword needed to make this a generator  # use of the yield gives this \_\_iter\_\_ and \_\_next\_\_ methods without having to write out all that ugly notation  yield seperate[n]  n += 1  '''  When I use my generator function and run the next method on it, after it has exausted the iterable, it will  throw an error and won't allow me to print it again.  It throws the error: StopIteration  '''  sentence\_2 = my\_gen('My name is Nasian')  print(next(sentence\_2))  print(next(sentence\_2))  print(next(sentence\_2))  # after this one it will throw an error  print(next(sentence\_2))  print(next(sentence\_2))  print(next(sentence\_2))  ############################################  ############################################  ############################################  '''  I had to copy the function over, as it processed the next in the sentence\_2 which was left as Nasian and it  kept cycling over Nasian.  Very interesting behaviour!  '''  def my\_gen(sentence):  n = 0  seperate = sentence.split()  max\_len = len(seperate)  while n < max\_len:  # yield is the keyword needed to make this a generator  # use of the yield gives this \_\_iter\_\_ and \_\_next\_\_ methods without having to write out all that ugly notation  yield seperate[n]  n += 1  sentence\_2 = my\_gen('My name is Nasian')  import itertools  cycle\_def = itertools.cycle(sentence\_2)  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  print(next(cycle\_def))  from itertools import cycle  # remove the itertools as it throws an error, you have imported cycle - no need to reference the module  counter = cycle(['my', 'name', 'is', 'nasian'])  print(next(counter))  print(next(counter))  print(next(counter))  print(next(counter))  print(next(counter))  print(next(counter))  print(next(counter))  '''  By using the cycle itertools, we can overcome this issue as it allows us to cycle through the iterable and  does not throw an error. Instead, it cycles through and keeps printing the iterable.  This is different as we able to use the cycle itertool on the generator function I have produced and keep using the  next method on the cycle itertool, and it won't throw a StopIteration error.  ''' |
| --- |

**SOURCES:**

* https://www.freecodecamp.org/news/how-and-why-you-should-use-python-generators-f6fb56650888/

1. Decorators - A page for useful (or potentially abusive?) decorator ideas. What is the return type of the decorator?

A Python decorator will take in a function, subsequently add some additional functionality to the function and finally return it. These decorators serve to add additional functionality to existing code, which is otherwise known as **metaprogramming** as a part of the program attempts at modifying another part of the program at compile time.

Python decorators are thus a very powerful tool for programmers as it permits them to change the behaviour of a class or function without changing the original code of the class or function. The original function is wrapped by the decorator to modify or extend the behaviour of the wrapped function, whilst the original function has not been permanently changed.

Python functions are first class objects thus functionals are able to be utilized or passed as arguments.

Some properties of first class functions:

* A function is an instance of the Object type.
* Functions can be stored in a variable
* The function can be passed as a parameter to another function
* The function can be returned from a function
* First class functions can be stored in data structures such as lists, hash tables etc

**So for Python decorators, functions are taken as the argument into another function and then called inside the wrapper function.**

The syntax for decorators are as follows and has been taken from [<https://www.geeksforgeeks.org/decorators-in-python/>]:

| @gfg\_decorator  def hello\_decorator():  print("Gfg")  '''Above code is equivalent to -  def hello\_decorator():  print("Gfg")  hello\_decorator = gfg\_decorator(hello\_decorator)''' |
| --- |

The **gfg\_decorator** is a callable function which will subsequently add some code on the top of some other callable function, **hello\_decorator function** and finally return the wrapper function.

The following is code taken from [<https://www.geeksforgeeks.org/decorators-in-python/>] which really succinctly outlines how Python decorators work:

| # defining a decorator  def hello\_decorator(func):  # inner1 is a Wrapper function in  # which the argument is called  # inner function can access the outer local  # functions like in this case "func"  def inner1():  print("Hello, this is before function execution")  # calling the actual function now  # inside the wrapper function.  func()  print("This is after function execution")  return inner1  # defining a function, to be called inside wrapper  def function\_to\_be\_used():  print("This is inside the function !!")  # passing 'function\_to\_be\_used' inside the  # decorator to control its behavior  function\_to\_be\_used = hello\_decorator(function\_to\_be\_used)  # calling the function  function\_to\_be\_used() |
| --- |

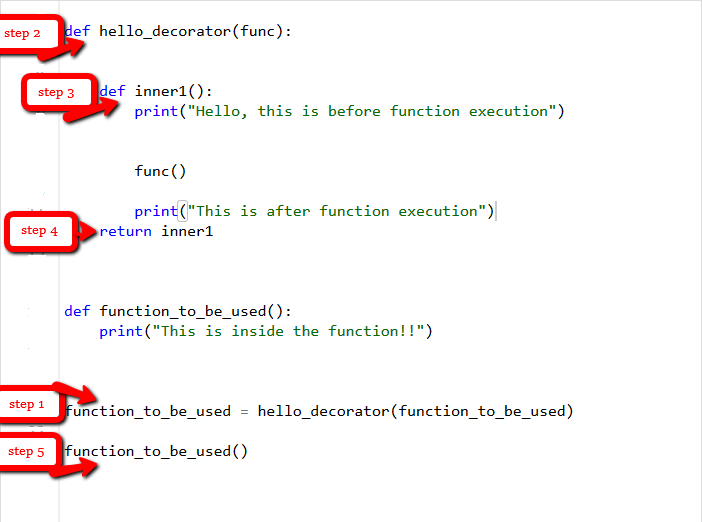
The outcome is as follows:

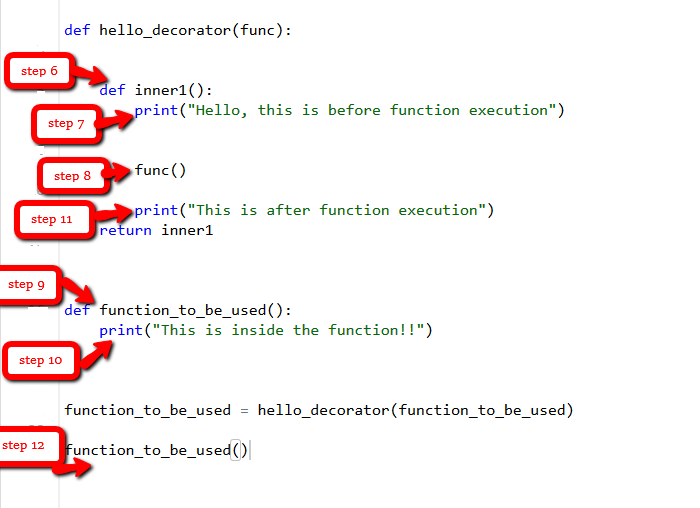
Hello, this is before function execution

This is inside the function !!

This is after function execution

The following pictures taken from [<https://www.geeksforgeeks.org/decorators-in-python/>] really successfully describe what is happening:





In the above examples, the function was not returning anything but in cases where the function is returning something or an argument is being passed to the function - it is good practice to make sure of the \*args and \*\*kwargs and pass these as the arguments in the inner functions e.g. **inner1(\*args, \*\*kwargs)** and **func(\*args, \*\*kwargs)**. \*args and \*\*kwargs allow the passing of a tuple of positional arguments or a dictionary of keyword arguments of any length. This then makes the decorator a general decorator that is able to decorate a function having any number of arguments.

**Chaining decorators** refers to the decoration of a function with multiple decorators.

**SOURCES:**

* <https://www.programiz.com/python-programming/decorator>
* https://www.geeksforgeeks.org/decorators-in-python/