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

Object Oriented Programming (OOP) and Process Oriented Programming (POP) are two distinct programming paradigms. OOP differs from POP in that the former is a bottom-up approach which revolves around data abstraction. The programs are divided into objects. The latter is a top-down approach which revolves around procedural abstractions and breaks down programs into functions. Python fits within OOP.

Here are some of the key differences in their approaches:

OOP	POP
Bottom-up approach	Top-down approach
Emphasis on objects- divided into these	Emphasis on functions- divided into these
Each object controls its own data	Functions share global variables
Data is accessed by the functions of an object	Data moves freely between functions
Access modifiers (i.e., Private, Public etc.)	No access modifiers
Data hiding is possible, enhancing security	Data hiding methods are not available
Additions can be added without revisions to the existing program.	The existing program must be revised if new functions or data is to be added.
C++, Java, Python, C#	C, Pascal, FORTRAN and COBOL

2. What's polymorphism in OOP?

Polymorphism refers to the ability to define methods which can take on multiple forms (i.e., methods of the same name can be implemented differently in classes). There are two types of polymorphism. Dynamic occurs during runtime and means a method is featured in both a parent and child class but may be implemented differently (i.e., different action). Static occurs during compiling and refers to cases where more than one method sharing the same name, but different arguments, co-exist within the same class. Python is dynamically typed.

These can be defined through functions and objects, class methods, and inheritance (as seen in the examples below). In the first version, the classes share method names and can be called upon through a function; once fed in as objects, they will produce their respective outputs/actions. In the second, both classes once again share methods of the same name but produce different outputs. A for loop will iterate through a tuple of objects and the methods will be called and outputs stored in the students variable. In the third example, the child class inherits the features of the parent class but they are amended to fit the particular child classes. This is the process of method overriding- we are able to access the overwrites despite the derived class sharing the same method name as the parent.

Example: **Polymorphism with function and objects**:

```
class Abyss():
    def platform(self):
        print("Netflix.")

    def origin_language(self):
        print("Korean.")

    def type(self):
        print("Series.")

class Great_Greek_Myths():
    def platform(self):
        print("Amazon Prime.")

    def origin_language(self):
        print("English.")

    def type(self):
        print("Series.")

def func(obj):
    obj.platform()
    obj.origin_language()
    obj.type()

obj_Aby = Abyss()
obj_GGM = Great_Greek_Myths()

func(obj_Aby)
func(obj_GGM)
```

Example: **Polymorphism with class methods**:

```
class Student:
    def __init__(self, name, stream):
        self.name = name
        self.stream = stream
    def instructor(self):
        print("Aleena")
    def project(self):
        print("Whodunnit game")

class Student_2:
    def __init__(self, name, stream):
        self.name = name
        self.stream = stream
    def instructor(self):
        print("Anyi")
    def project(self):
        print("Spotify project")

obj_stu = Student("Karan", "Software")
obj_stu2 = Student_2("Someone", "Software")
```

```
for students in (obj_stu, obj_stu2):
   students.instructor()
   students.project()
```

Example: **Polymorphism with inheritance**:

```
class Haunting_of_Hill_House:
   def intro(self):
       print("This is a show featured on Netflix surrounding a scary
house.")

   def survived(self):
       print("Not everybody survived the entirety of the show. Hope this
isn't a spoiler")

class Nell(Haunting_of_Hill_House):
   def survived(self):
       print("Nell Crain did not survive.")


class Shirley(Haunting_of_Hill_House):
   def survived(self):
       print("Shirley Crain did survive.")

obj_HHH = Haunting_of_Hill_House()
obj_Nell = Nell()
obj_Shirley = Shirley()

obj_HHH.intro()
obj_HHH.survived()
obj_Nell.intro()

obj_Nell.survived()
obj_Shirley.survived()
```

3. What's inheritance in OOP?

Inheritance refers to the process of deriving or extending the properties of another class; having derived these, the 'child' class will acquire the attributes and methods of the base/'parent' class so that they now share these in common. It can serve to make a program more reliable through enabling reusability and removes the need to revisit and modify a class to add features. A hierarchy can be created through different levels of inheritance, as demonstrated below in the examples. Where we do not wish for the child class to inherit all the features of the parent class, we can make them private through using '__' prior (e.g. `self.__staff.id = 87898`).

Example: **Single inheritance-** the child class (Employee) inherits from a parent class (Person). The information from the parent class can now be accessed through calling the child too.

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

```
        self.staff_id = staff_id

    def retrieve_info(self):
        print (self.name)
        print (self.staff_id)

class Employee(Person):
    def __init__(self, name, staff_id, level, department):
        super(Employee, self).__init__(name, staff_id)
        self.level = level
        self.department = department

    def get_info(self):
        print("{} is a {} in the {} department".format(self.name,
self.level, self.department))

Bob = Employee('Bob', 2234, 'Senior', 'Design')
Bob.get_info()
Bob.retrieve_info()
```

Example: **Multiple inheritance-** a child class (Employee) inherits from two or more parent classes (Person and Department).

```
# Parent class
class Person:
    def person_info(self, name, age, company):
        print('Name:', name, 'Age:', age, 'Company:', company)

# Parent class 2
class Department:
    def department_info(self, department_name, team_size):
        print('Name:', department_name, 'No. of members in team:',
team_size)

# Child class
class Employee(Person, Department):
    def Employee_info(self, salary, level):
        print('Salary:', salary, 'Level:', level)

emp = Employee()
emp.person_info('Karan', 21, 'EMSOU')
emp.department_info('Data Team', 10)
emp.Employee_info(17000, 'Intern')
```

Example: **Multilevel inheritance-** this refers to the ability of a derived class to inherit from another derived class (e.g., Daughter can inherit from Mother, a derived class of the parent/base class, Grandmother)

```
# Base class
class Grandmother:
  def __init__(self, grandmother_term):
     self.grandmother_term = grandmother_term

# Middle class
```

```
class Mother(Grandmother):
  def __init__(self, mother_term, grandmother_term):
     self.mother_term = mother_term

     # invoking constructor of Grandmother class
     Grandmother.__init__(self, grandmother_term)

# Last/derived class
class Daughter(Mother):
  def __init__(self, daughter_name, mother_term, grandmother_term):
     self.daughter_name = daughter_name

     # invoking constructor of Mother class
     Mother.__init__(self, mother_term, grandmother_term)

  def print_names(self):
     print("Grandmother :", self.grandmother_term)
     print("Mother :", self.mother_term)
     print("Daughter's name :", self.daughter_name)

me = Daughter('Karan', 'Mumsy', 'Bibi Ji')
me.print_name()
```

(2nd example)

```
# Base class
class Uni_Student:
   def student_info(self, name, age, uni_name, year):
       print('Name:', name, 'Age:', age, 'University:', uni_name, 'Year:',
year)

# Middle class
class school(Uni_Student):
   def uni_info(self, school_name, head, campus):
       print('School:', school_name, 'Head of School:', head, 'Campus:',
campus)

# Last/derived class
class Summer_Modules(school):
   def modules_info(self, module_1, module_2, module_3):
       print('Module 1:', module_1, 'Module 2:', module_2, 'Module 3:',
module_3)

uni_m = Summer_Modules()

uni_m.student_info('Karan,', 21, 'University of Nottingham,', 'Graduate')
uni_m.uni_info('Sociology and Social Policy,', 'Rachel Fyson,', 'Uni Park')
uni_m.modules_info('Dissertation,', 'Crimes and Harms of the Powerful,',
'Cyber Crime')
```

Example: **Hierarchical inheritance-** this is when more than one child/derived class is created from a single parent/base class (Movies).

```
class Movies:
   def info(self):
```

```
        print("There are many movies of different genres and audios")

class ten_things(Movies):
    def ten_things_info(self, name, genre, audio):
        print(name + ' is a ' + genre + ' that is originally spoken in ' +
audio)

class K3G(Movies):
    def K3G_info(self, name, genre, audio):
        print(name + ' is a ' + genre + ' that is originally spoken in ' +
audio)

class Parasite(Movies):
    def Parasite_info(self, name, genre, audio):
        print(name + ' is a ' + genre + ' that is originally spoken in ' +
audio)

obj_1 = K3G()
obj_1.info()
obj_1.K3G_info('Khabhi Khushi Kabhi Gham', 'Bollywood Melodrama', 'Hindi')

obj_2 = Parasite()
obj_2.info()
obj_2.Parasite_info('Parasite', 'Thriller', 'Korean')
```

4. 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?

The first step would be to do a quick overview of the key requirements, potential problems, and tools to make the program.

Key requirements:

- Ensuring each staff ID is only associated with 3 distinct votes- cannot vote twice for the same person and can only vote 3 times.
- Need to keep a count of the number of votes each person casts and the number of votes each member has received
- Ensure the nominees have valid staff IDs to prove the list is limited to those in the office.
- Return results from the SQL DB must be in desc order and limited to 3 to retrieve the top 3 funniest people.

Potential problems:

- Late votes
- Potential for false staff IDs to be input
- Votes could be casted on others behalf.
- Handling draws?
- Users not casting all 3 votes in one sitting?

Tools:

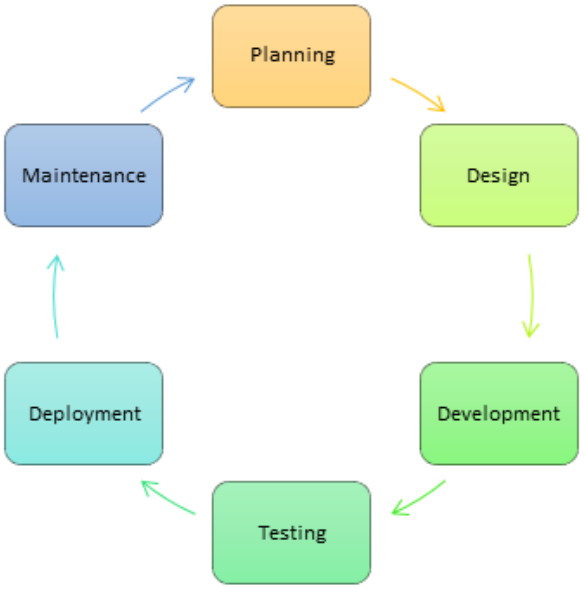
- SQL database with tables containing: the names and IDs of the staff members in the office, a table to store employee IDs and their names and the votes they have, and

potentially another table to store employee IDs and the number of votes they have casted.
- .py file- MySQL connector and config details to link to the SQL database.
- Cursor to handle SQL queries
- .py file containing code for get and put requests to retrieve the list of potential nominees and to push data into the tables regarding number of casted votes and how many votes each person has received.
- .py file- main script containing user interface- will ask for their ID and their nominee.

The general idea is that a staff member would be asked to provide their staff ID before a get request retrieves information from an SQL DB table containing all the possible nominees. The user will have the opportunity to vote for 3 distinct people. The put requests will update a column +1 for each nominee they have voted for. It will also do the same for another column containing information regarding how many times the user has already casted a vote. Once the voting session ends, the get request will pull information from the scores table regarding the top 3 funniest people in the office.

5. What's the software development cycle?

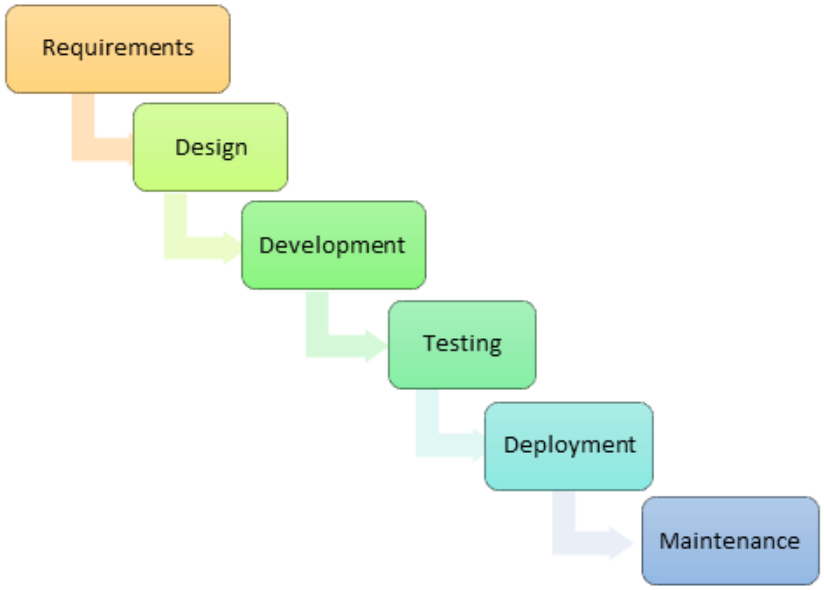
The software development cycle is the term applied to the typical phases undergone to develop software from start to finish. There is a general agreement that this framework consists of around six stages: planning (analysis), design, development, testing, development, deployment, and maintenance (as seen in the diagram below).



The initial stage consists of **planning**, which may also be considered the requirements or analysis phase. During this stage, decisions are made or information is received regarding what the software needs to do- this will include finding out about the purpose of the product, the end users, their expectations of how it should work, among other important details the client will have. Resource allocation may also be featured under the planning phase. The **design stage** involves creating the software architecture for the product.The overall purpose may be broken into smaller stories or aspects, including the visual appearance, UI etc. Wireframes may be designed to gain a better insight into how the end users will interact with

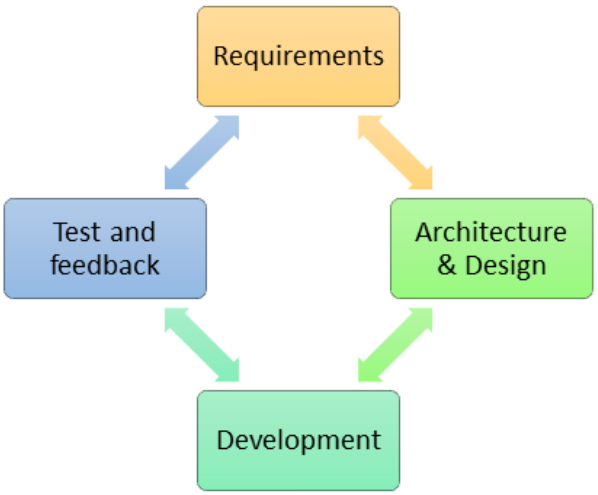
the product and what is to be expected. Diagrams and pseudocode may be useful tools in determining the overall look and programming behind the software.
At the **development** stage, the design becomes more tangible through the source code created by the team of software developers using a high-level programming language. These tasks are likely to be broken into smaller stories to be tackled across the team. These modules/programs will be released to be **tested** for a number of things including bugs and issues, functionality, usability etc. This can be preventative as potential issues are flagged up prior to release and so it can be modified. Performing unit tests, among others, minimises the chances of an unsuccessful deployment i.e., one that essentially fails to meet client/customer expectations. Once rigorously tested, the software is ready to be **deployed** to be used by end users. There are different boundaries to this- for example, some products are released to a smaller sample of users to allow for feedback and further modifications before a Beta release to a larger population. Following the deployment of software, continuous **maintenance** is still required; this may be corrective (fixing bugs that arise), perfective (adding additional features), or adaptive (making changes based on new conditions). As users begin to utilise software, feedback may uncover new requirements or issues to be worked on. Moreover, as technology continues to keep advancing, the software may be updated to account for these new developments.

6. What's the difference between agile and waterfall?
Waterfall and agile are two approaches to the software development cycle. The former is a more linear, sequential version, in which it is generally accepted that one phase will be completed before the next is to take place.The more distinct step-by-step method requires a clear understanding of the requirements, schedule, milestones/deadlines etc at the outset of the development cycle. So long as there is little deviation from these, this approach can help ensure there is little chance of derailing from schedule or miscalculating the scope of the project. Due to its rigidity, it is best implemented when there are already clear, fixed requirements and expectations since there is less room for feedback that may lead to big changes or amendments.
Waterfall:



On the other hand, the agile approach to software development is much more flexible. As displayed in the diagram below, there is greater opportunity to move between phases, allowing for continuous feedback, revisions and amendments. This flexibility means the initial list of requirements need not be fully complete, unlike the waterfall model. While goals are set at the outset of a sprint (short two week periods approx), the feedback obtained from the client at scheduled review meetings can lead to changes in the original plans or the discovery of newer requirements which are added as additional features to be tackled in the next sprint. This may be the preferred method to waterfall if the set of requirements is not fixed; there is a more collaborative effort throughout the cycle as the software is built incrementally, tested, and reviewed continuously without a dependency on either stage being fully completed first.

Agile:



7. What is a reduced function used for?

The reduce function is featured in the functools module and essentially processes all the elements of a sequence and reduces them to a singular value according to a given function (making reduce() a higher order function). While a for loop could be used to receive the same output, the reduce() function is much more concise.
The function takes in two required parameters: any function, so long as it accepts two arguments, and a sequence, which can be any form of an iterable (list, tuples etc). There is also a third optional parameter, an initialiser. If a value is given to this parameter, reduce() will be called with this value as its first argument and the first item in the sequence. This optional parameter is especially useful when processing a potentially empty sequence; a TypeError can be avoided by providing a default value to be returned in this case.
The process is as follows:

- The given function is applied to the first two items in a given sequence to return a result
- The function is then called again using this first result and the next element in the sequence to generate another result.
- This process continues until the function has been applied to every item in the iterable to return a final singular cumulative value.

This is the syntax:

```
reduce(func, seq [, init])
```

Example: **Sum use case-** this example calculates the cumulative total of all the values in a given sequence.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

def add_two_numbers(x, y): #function with two arguments
   result = x + y
   return result

print(reduce(add_two_numbers, numbers, 300)) #since 300 is set as the
initialiser, the return will be 315

result2 = reduce(lambda x,y: x + y, numbers)#similar but using lambda and
no init value
print(result2)# return is 15
```

Example: **Product use case-** this example calculates the product of all the values in the given sequence.

```
def multiply_numbers(x, y):
   return x * y

print(reduce(multiply_numbers, numbers))#returns 120

from operator import mul #using mul()to achieve the same result

print(reduce(mul, numbers))#returns 120
```

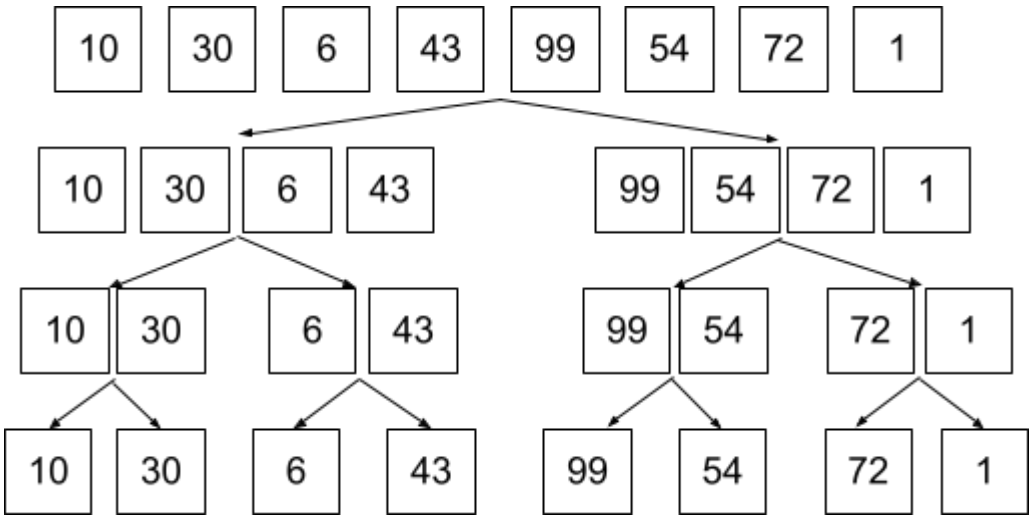
Example: **Boolean use case-** reduce() can also be used to test if any elements in a given sequence are True. If so, it will return True. Iif not, it will return False.

```
def are_any_true(x, y):
   return bool(x or y)

print(reduce(are_any_true, [0, None, 0, 1, 0]))#True
print(reduce(are_any_true, [0, [], 0, None])) #False

print(reduce(are_any_true, [1, False])) #True
```

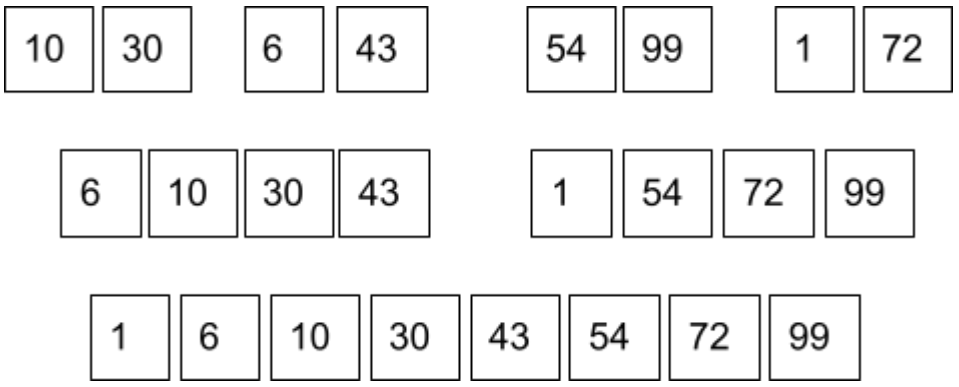
8. How does merge sort work?

Merge Sort is a sorting algorithm that utilises a divide-and-conquer approach to arrange unsorted elements of a traversable object (e.g., a list). 'Divide' refers to the process of dividing the elements of a given list into smaller parts/sublists. The function is repeatedly called upon itself as the array is continuously divided into halves, making it a recursive technique. This continues until every element is separated individually and can therefore no longer be divided (i.e, there is one element in each subsection.)

Dividing:



Conquering(aka merging):
'Conquering' refers to the process of comparison the individual elements will then undergo- if two elements are sorted, they are merged into an array. If not, they are first swapped. These pairs will then be compared, swapped (if needed), and merged in a similar fashion to form sub arrays. This continues until a final comparison and merging will result in a sorted array.



In regards to the code, the following steps must be taken:

- Storing the size/length of the traversable object- this will be used to find the midpoint of the list to determine the division.
- Calling upon the merge_sort() function to halve the sublist continuously.
- Begin the merging by setting up trackers for the leftmost element of each array, since these will be compared, as well as one for the merged array.
- The sub lists will continue to be merged until a final sorted array is fully merged.

Example:

```
def merge_sort(given_array):
   size = len(given_array)
   if size > 1:
       mid_point = size // 2 #formula to halve
       left_side = given_array[:mid_point]
       right_side = given_array[mid_point:]

       merge_sort(left_side)
       merge_sort(right_side)
```

```
        i = 0
        j = 0
        k = 0

        left_length = len(left_side)
        right_length = len(right_side)
        while i < left_length and j < right_length:
            if left_side[i] < right_side[j]:
                given_array[k] = left_side[i]
                i += 1
            else:
                given_array[k] = right_side[j]
                j += 1

            k += 1

        while i < left_length:
            given_array[k] = left_side[i]
            i += 1
            k += 1

        while j < right_length:
            given_array[k] = right_side[j]
            j += 1
            k += 1


nums = [10, 30, 6, 43, 99, 54, 72, 1]
merge_sort(nums)
print(nums)#[1, 6, 10, 30, 43, 54, 72, 99]- as seen in the diagram
```

9. 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?

Generators are a useful tool when creating programs that do not rely on the entire contents of a given object being processed and stored at once; therefore, memory does not need to be allocated for all the results at one point in time. Sequences can be looped/iterated through without storing the contents into memory, which is especially useful in use cases where programmers are working with a large file or quantity of data.

The major difference between generators and alternative methods is the use of 'yield' rather than 'return'. The latter works by exiting the function after the result value is returned to the caller. Items are stored at once and returned all at once, rather than one-by-one, which can potentially be a strain on memory. On the other hand, yield does not exit the function and the local variables are not discarded. Instead, the yielded value is returned and the state of the function is remembered and remains suspended until the next item is called for. So long as the generated values are not required more than once, generators can be the preferable option due to memory, conciseness, among other reasons.

Example: **Reading a large CSV file.**

One use case is when working with large files, such as CSV files. The code below shows a generator can be used to count the number of rows in a text file. The alternative method

would be to open the file, read all the lines, have them stored into memory into an array as the rows are split. While this is a solution, it can only work up to a certain extent- when a file is too large (i.e., over a few thousand lines), the program may result in a MemoryError as the entirety of the file exceeds the limit that can be read and loaded into memory. This is a non-issue with generators since a row can be yielded one at a time rather than returned, minimising the memory storage required.

```
def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row

csv_gener = csv_reader("experiment.txt")
row_count = 0

for row in csv_gen:
    row_count += 1

print('The row count is {}'.format(row_count))
```

Example: **Adding positive integers to a given *n***

The following example calculates the sum of the integers prior to the given *n* (e.g., if *n* is 3, then 1+2). An alternative to using a generator would be to build a list that is continuously appended to and then returned; while this is fine for a small integer, it will become an issue when given a very large *n* since all of the integers are kept in memory. Using a generator in this case allows us to reap the benefits of iterators but implement them in a simple, concise way and so the memory usage is minimised again.

```
def sum_pre_no(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_of_previous_n = sum(sum_pre_no(5))
print(sum_of_previous_n)
```

Example: **Generating an infinite sequence**

Another use case is in the generation of an infinite sequence. Since memory is not infinite too, a generator can be used to yield each number in its current state as it increments by 1 from the last yielded value. Calling next() displays the values one by one while it is manually asked for.

```
def infinite_seq():
    num = 0
    while True:
        yield num
        num += 1

gen = infinite_seq()
print(next(gen))
print(next(gen))
print(next(gen))
```

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

Decorators are functions that wrap around another function, essentially providing new functionality. There are several uses for decorators, some of which are provided below.

Example: **calculating-** an example of a nested function being used in a calculation. This program will add 2 to 12, as per the nested function, and then subtract 3.

```
def add_two(func):
    def add_two_inner(*args, **kwargs):
        return func(*args, **kwargs) + 2
    return add_two_inner

@add_two
def subtract_another_three(x):
    return x - 3

print(subtract_another_three(12))
```

Example: **exception handling-** decorators can also be used to handle exceptions e.g., TypeError.

```
def exception_handling(func):
    def inner_function(*args, **kwargs):
        try:
            func(*args, **kwargs)
        except TypeError:
            print('Please provide an integer')
    return inner_function

@exception_handling
def square_no(n):
  print(n * n)

@exception_handling
def subtracting_no(n):
  print(n - 1)

square_no(3)
square_no("Not a number")
subtracting_no(2)
subtracting_no("Not a number")
```

Example: **property decorator-** these can be used to act as getters (to retrieve values) and setters (to assign a value).

```
class Item:
    def __init__(self, price): #set initial price
        self._price = price

    @property #getter
    def price(self):
        return self._price
```

```
    @price.setter #setter
    def price(self, ammended_price):
        if ammended_price > 0 and isinstance(ammended_price, float):
            self._price = ammended_price
        else:
            print("Please enter a valid price- a float that is above 0")

item = Item(30.00)
item.price = 32.00
print(format(item.price, '.2f'))
```

Example: **memoize decorator-** a memoize decorator can assist in recording intermediate results into a memory assigned to a variable (cache_mem). This program calculates the factorial of a number.

```
def memoize_factorial(f):
    cache_mem = {}

    def inner(num):
        if num not in cache_mem:
            cache_mem[num] = f(num)
        return cache_mem[num]

    return inner

@memoize_factorial
def factor(num):
    if num == 1:
        return 1
    else:
        return num * factor(num - 1)

print(factor(5))
```

Example: **timer decorator-** decorators may also be applied to implement a timer. The following program shows an output of the execution time of the testing_time function.

```
from time import time

def timer(func):
    def wrap_func(*args, **kwargs):
        t1 = time()
        result = func(*args, **kwargs)
        t2 = time()
        print(f'This function executed in {(t2 - t1):.4f}s')#4 dp
        return result

    return wrap_func

@timer
def testing_time(n):
    for i in range(n):
```

```
        for j in range(20000):
            i * j

testing_time(3)
```

Example: **class decorator- using __call__ .** Using this method means the given instances behave like a function and can therefore be called upon like one.

```
class class_dec:
    def __init__(self, simple_function):
        self.function = simple_function

    def __call__(self, *args, **kwargs):
        print("Hello")
        self.function(*args, **kwargs)
        print("Goodbye")

@class_dec
def my_function(message1, message2):
    print("{} {}".format(message1, message2))

my_function("This example uses a:", "class decorator")
```

Example: **chained decorators-** more than one decorator can be applied to a function e.g. @brackets and @stars

```
def brackets(func):
    def inner(*args, **kwargs):
        print("[]" * 10)
        func(*args, **kwargs)
        print("[]" * 10)
    return inner

def stars(func):
    def inner(*args, **kwargs):
        print("*" * 20)
        func(*args, **kwargs)
        print("*" * 20)
    return inner

@brackets
@stars
def greeting(message):
    print(message)

greeting("Hi Aleena :)")
```

Abusive decorators/bad situations:

Decorators can lose their effectiveness when they lose their readability, which can make it difficult to debug as there is a lack of understanding regarding what the function actually does. A user online stated they came across the following code:

```
·     @text
·     @user
def create_post(user, text):
·         backend.callCreatePost(user, text)
·     create_post(request)
```

They bring up the valid point that interdependency between decorators and where they provide parameters (as displayed above) can also become problematic when a function predetermines an arrangement that may not be best suitable. In some cases, it can be best to create programs void of decorators that are more readable, if not slightly longer.

Decorators can also provide problems in that when they are applied to a function, the function is removed from the namespace- this makes it difficult to call its original form.

Using decorators on class methods may also result in an error stating the method is not callable. The location of the decorators can be of great importance in this situation. The wrappers created will not have the entirety of attributes copied.

The return of decorator type is a modified function or class (object).