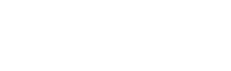
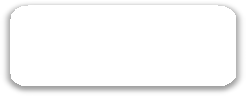
THEORY QUESTIONS ASSIGNMENT

Software Stream



**Maximum score: 100**

KEY NOTES

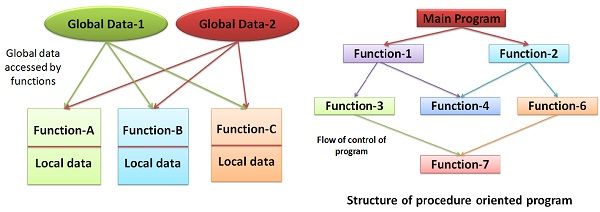
* This assignment to be completed at student’s own pace and submitted before given deadline.
* There are 10 questions in total and each question is marked on a scale 1 to 10. The maximum possible grade for this assignment is 100 points.
* Students are welcome to use any online or written resources to 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** |

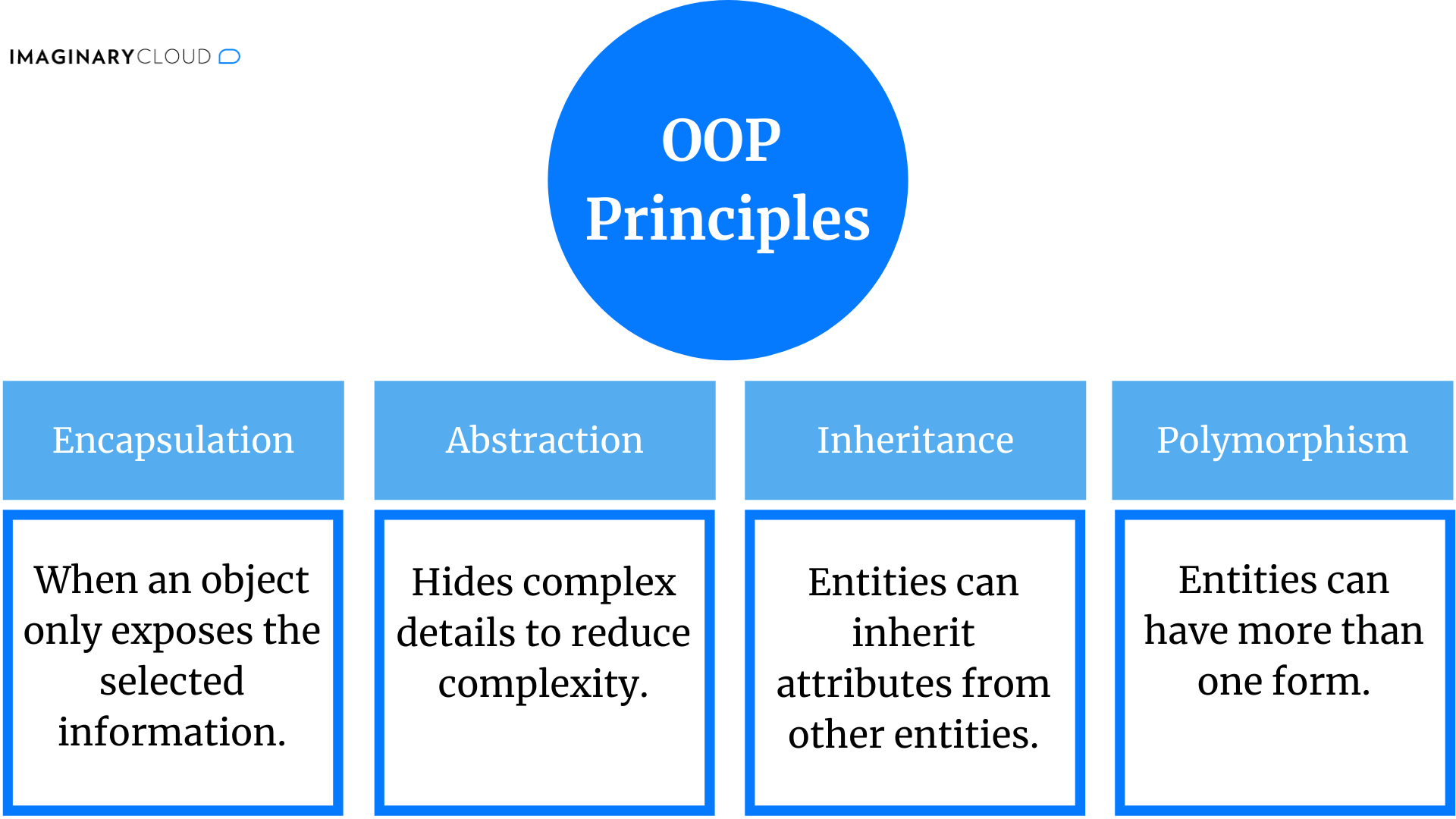
1. **How does Object Oriented Programming differ from Procedural Oriented Programming?**

Object Oriented Programming (OOP) and Procedural Oriented Programming (POP) are two examples of programming paradigms (approaches) to solving problems and developing software. All programming languages follow at least one programming paradigm (some use multiple paradigms) and each has its own strengths and weaknesses depending on the requirements of the application.

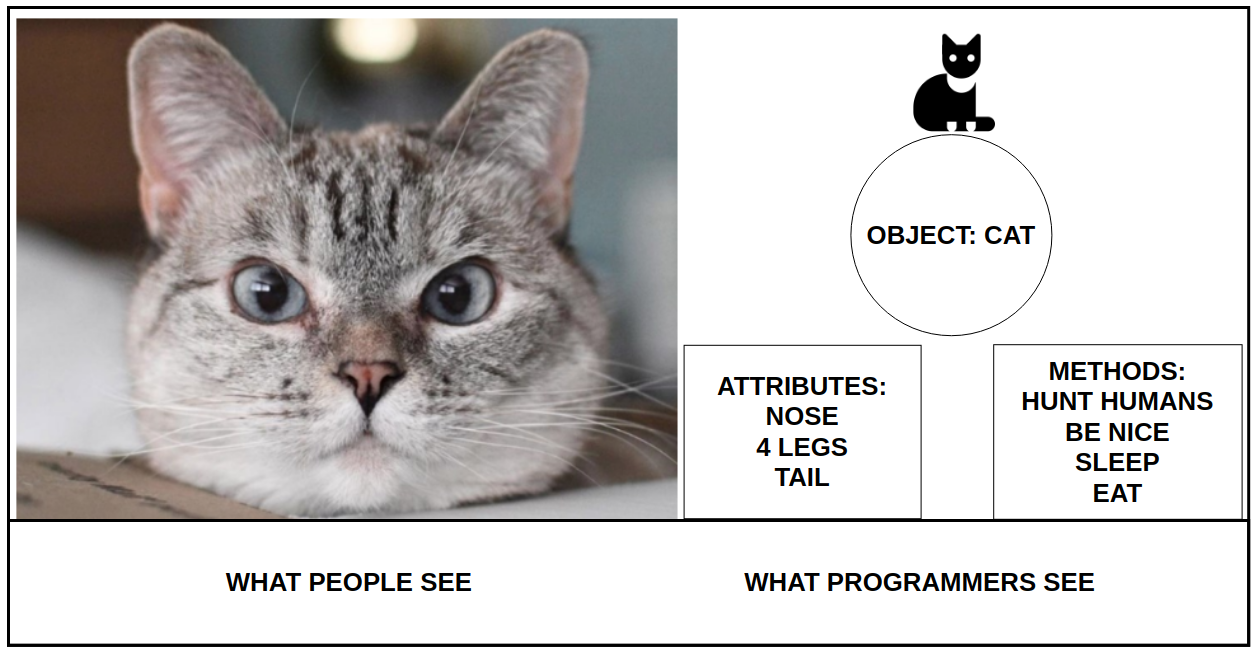
Procedural Oriented Programming (POP) is derived from structured programming and takes a top-down approach, completing tasks in sequential order. It is based on procedures (routines, subroutines, functions) – sets of step-by-step instructions and computations to be carried out. When a program is executed, any given procedure can be called at any point. Large programs are structured in small units that share global data. This is a data security concern as unintentional changes can be made in the program by functions. Languages used with PP – FORTRAN, ALGOL, COBOL, BASIC, Pascal and C.



Object Oriented Programming (OOP) is based on the concept of objects that relate to the real world. They contain data (attributes) and behaviours (methods), which interact with other elements in the application. For example, an object could be a cat. That cat would have a name (a property of the object) and would know how to ‘meow’ (a method). A method in OOP is similar to a procedure in PP, the difference being that the method belongs to a particular object. The most popular OOP languages are class-based – a class can be considered as a blueprint for an object; therefore, objects are said to be instances of classes. Pure object-oriented languages follow the four core principles: encapsulation, abstraction, inheritance, and polymorphism. Languages used with OOP – Java, C++, C#, Python, PHP, JavaScript, Ruby, Perl, Objective-C, Dart, Swift, Scala.

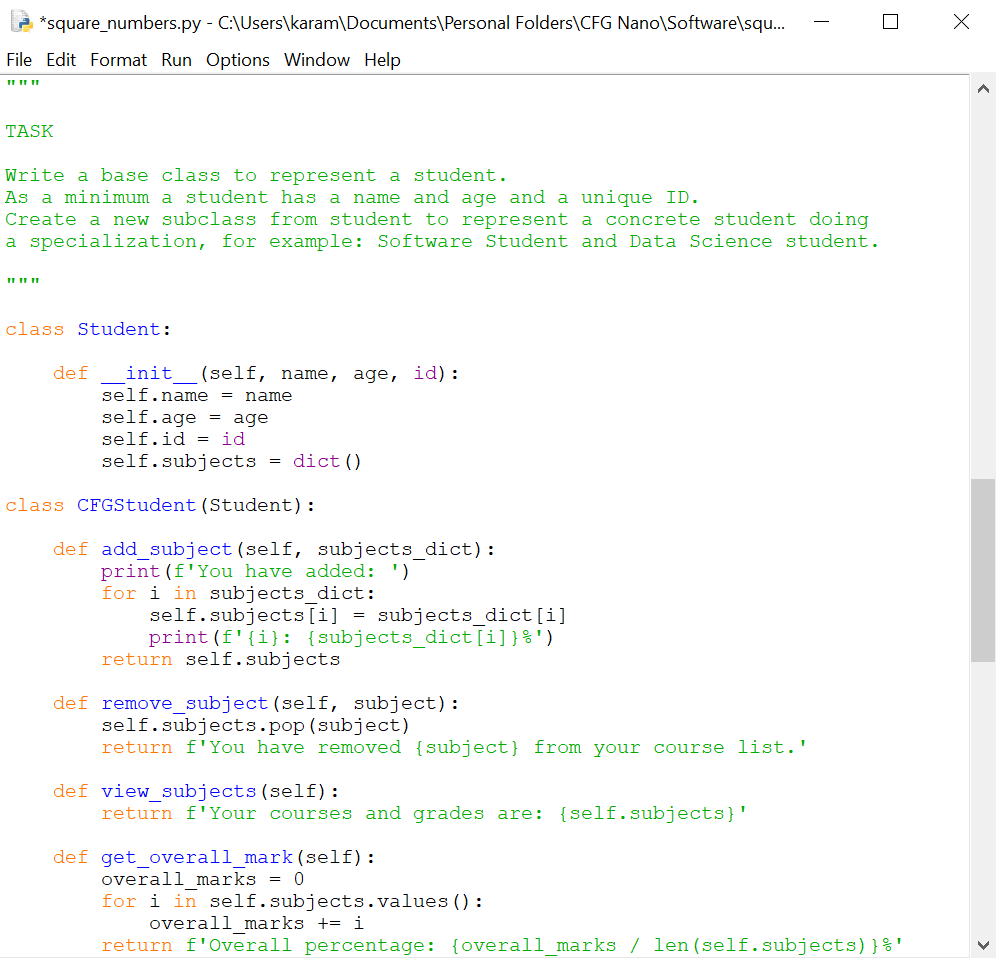


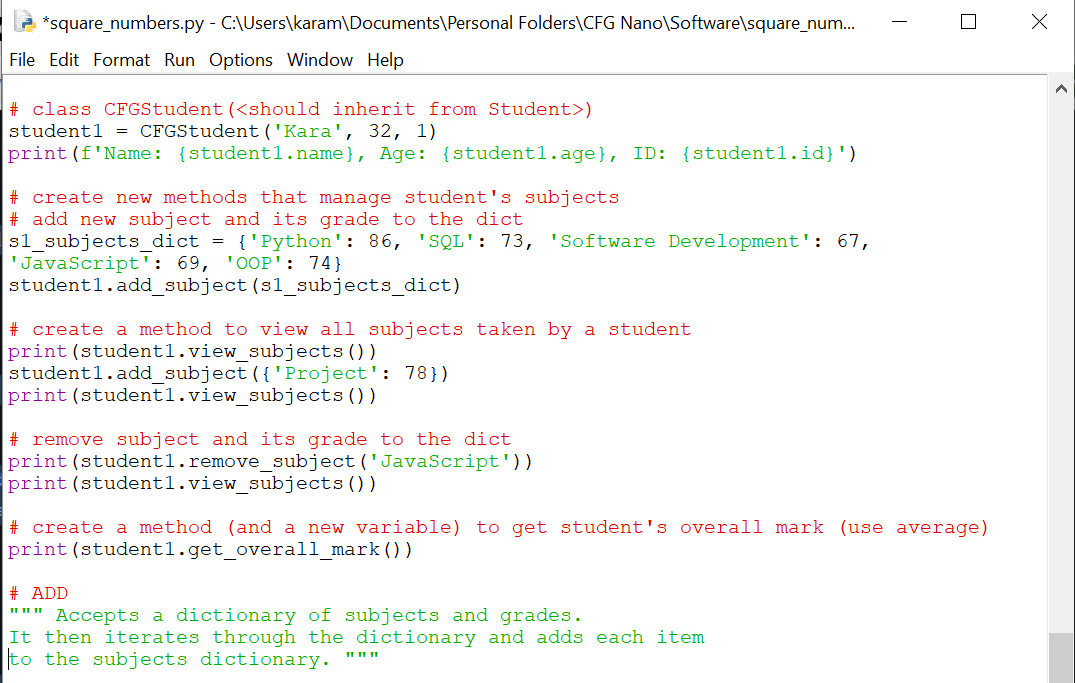
OOP can have several advantages over POP, especially if the task is highly complex. As data is grouped with the functions that operate on it, it is more secure. Access to certain attributes can also be restricted. Code is more reusable with OOP as once one class is created, multiple instances (objects) can be created. Lengthy POP programs are prone to flaws and can be difficult to read and understand if not organised carefully, and accessibility of data throughout the program can make it vulnerable.

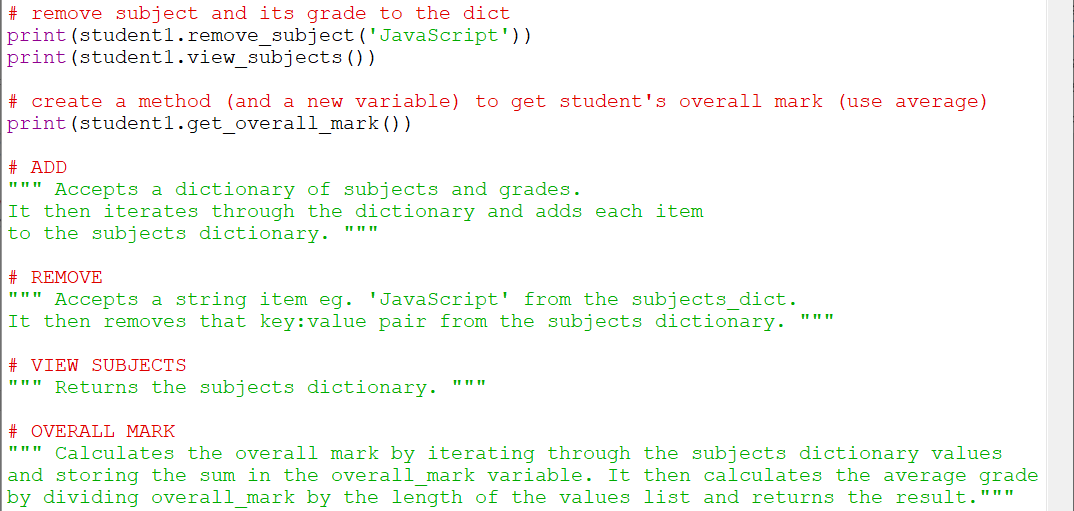


Example of OOP:

*Student class with name, age, ID, subject/grade (dict) attributes. CFGStudent subclass inherits attributes from Student class and has its own methods for adding, removing, and viewing subjects and for getting the overall mark (average grade). The Student class is like a template from which several different types of students could be created.*







| **BASIS FOR COMPARISON** | **POP** | **OOP** |
| --- | --- | --- |
| Basic | Procedure/Structure oriented. | Object oriented. |
| Approach | Top-down. | Bottom-up. |
| Basis | Main focus is on "how to get the task done" i.e., on the procedure or structure of a program. | Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class. |
| Division | Large program is divided into units called functions. | Entire program is divided into objects. |
| Entity accessing mode | No access specifier observed. | Access specifier are "public", "private", "protected". |
| Overloading or Polymorphism | Neither it overloads functions nor operators. | It overloads functions, constructors, and operators. |
| Inheritance | There is no provision of inheritance. | Inheritance achieved in three modes public private and protected. |
| Data hiding & security | There is no proper way of hiding the data, so data is insecure | Data is hidden in three modes public, private, and protected. hence data security increases. |
| Data sharing | Global data is shared among the functions in the program. | Data is shared among the objects through the member functions. |
| Friend functions or friend classes | No concept of friend function. | Classes / function can become a friend of another class with keyword "friend". (used only in c++) |
| Virtual classes or virtual function | No concept of virtual classes. | Concept of virtual function appear during inheritance. |
| Example | C, VB, FORTRAN, Pascal | C++, JAVA, VB.NET, C#.NET. |

**Advantages**

POP (Procedure Oriented Programming)

* Ability to reuse the same code in various places
* Facilitates tracking of program flow
* Ability to construct modules

OOP (Object Oriented Programming)

* Objects assist with task partitioning in the project
* Data hiding allows secure programs to be built
* Can potentially map the objects
* Categorisation of objects into various classes
* Object-oriented systems can be upgraded easily
* Inheritance eliminates redundant code
* Codes can be extended using reusability
* Greater modularity can be achieved
* Increased reliability due to data abstraction
* Flexible due to the dynamic binding concept
* Information hiding decouples the essential specification from its implementation

**Disadvantages**

POP (Procedure Oriented Programming)

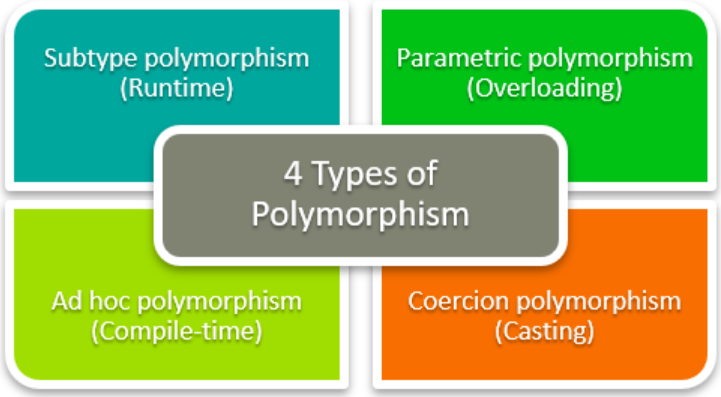
* Global data is vulnerable
* Data can move freely within a program
* Can be difficult to verify the data position
* Functions are action-oriented
* Functions are not capable of relating to the elements of the problem
* Real-world problems cannot be modelled
* Parts of code are interdependent
* Lack of reusability – one application code may not be able to be used in other application
* Data is transferred by using the functions

OOP (Object Oriented Programming)

* More resources are required
* Dynamic behaviour of objects requires RAM storage
* Detection and debugging can be challenging in complex applications when the message passing is performed
* Inheritance makes classes tightly coupled, which can affect the reusability of objects

1. **What's polymorphism in OOP?**

Polymorphism means taking many forms. In programming, it refers to the use of a single entity (method, operator or object) to represent different types in different scenarios. For example, the addition operator (+) has several uses. It is used for arithmetic addition with integers and concatenation with strings.



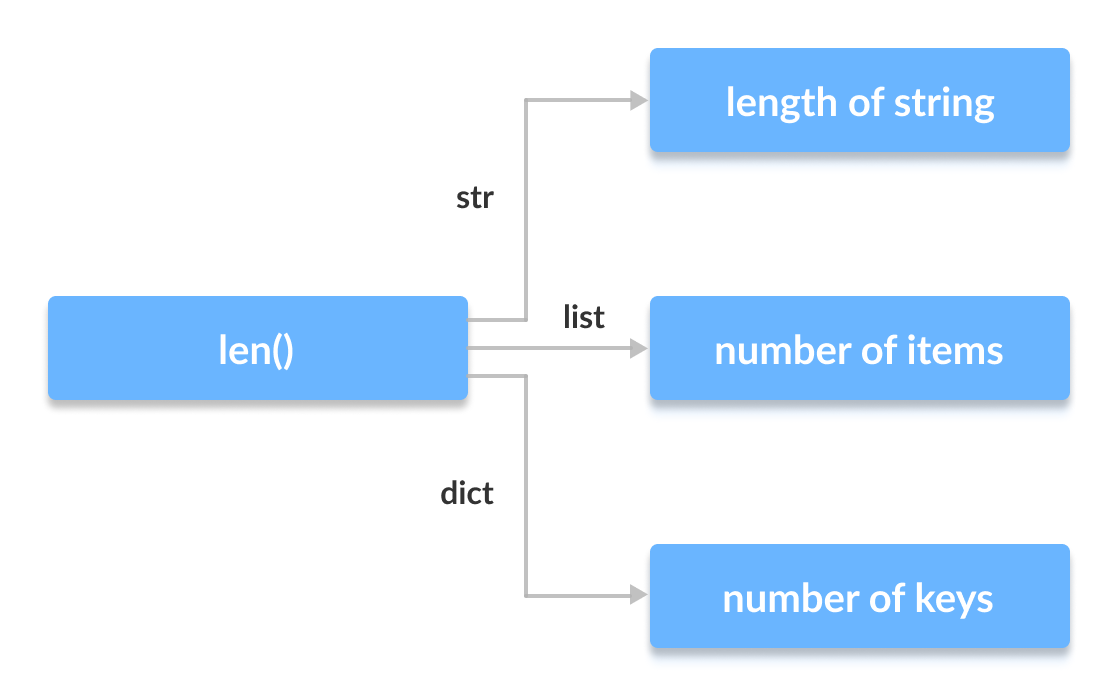
**Function polymorphism**

Some functions are compatible with various data types, such as len(). The len() function can be used with a string, list, tuple, set, or dictionary but return specific information about each.

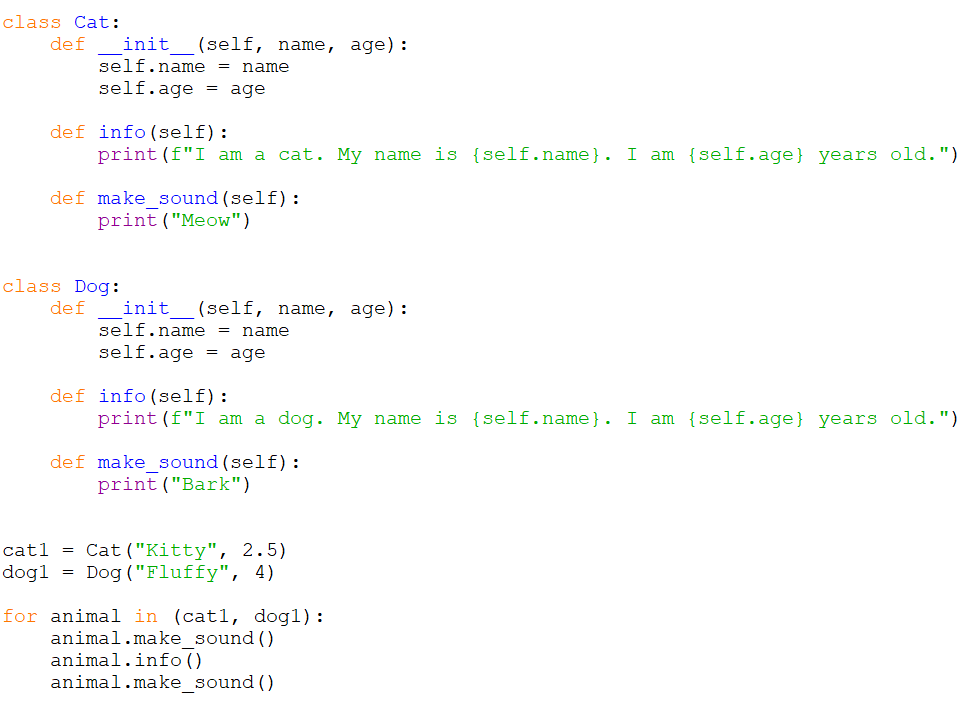
**print(len("Hello World!"))**

**print(len(["Python", "Java", "C"]))**

**print(len({"Name": "Kara", "Address": "London"}))**



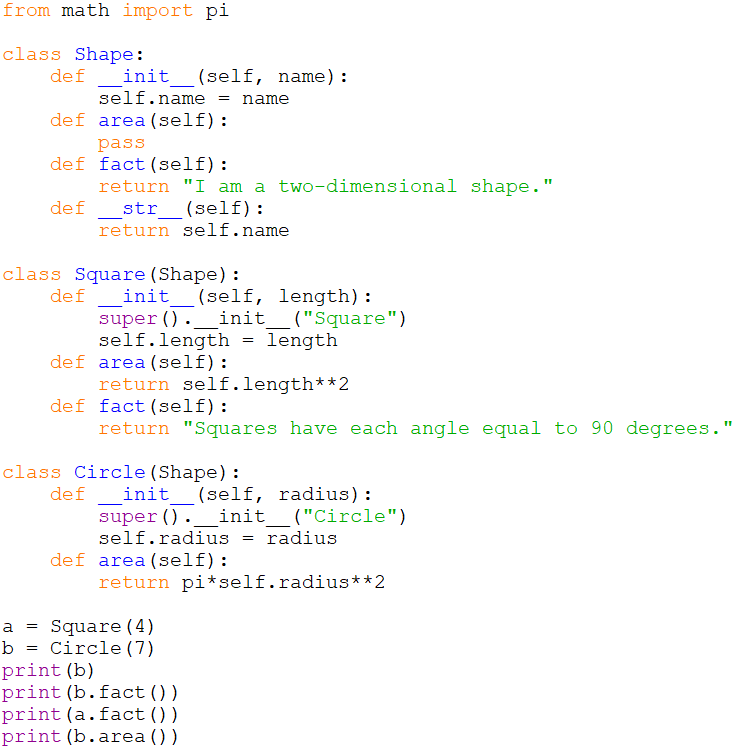
The concept of polymorphism can be used while creating class methods because Python allows different classes to have methods with the same name. These methods can be generalised by disregarding the object we are working with.



Two classes, Cat and Dog, have been created. Their structures are similar and both have the same method names info() and make\_sound(). There is no common superclass or link between the classes. However, due to polymorphism, these two different objects can be put into a tuple and iterated through using a common animal variable.

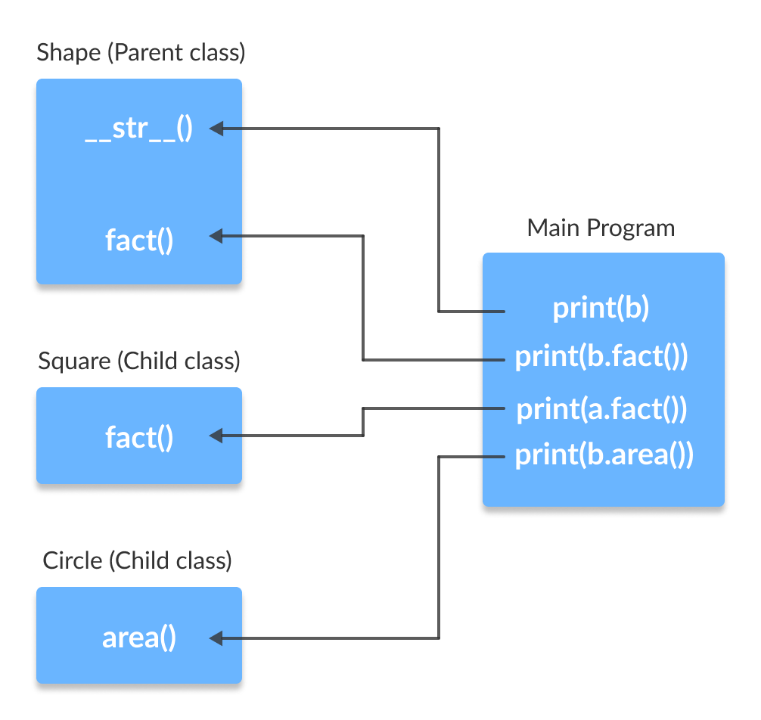
**Polymorphism and Inheritance**

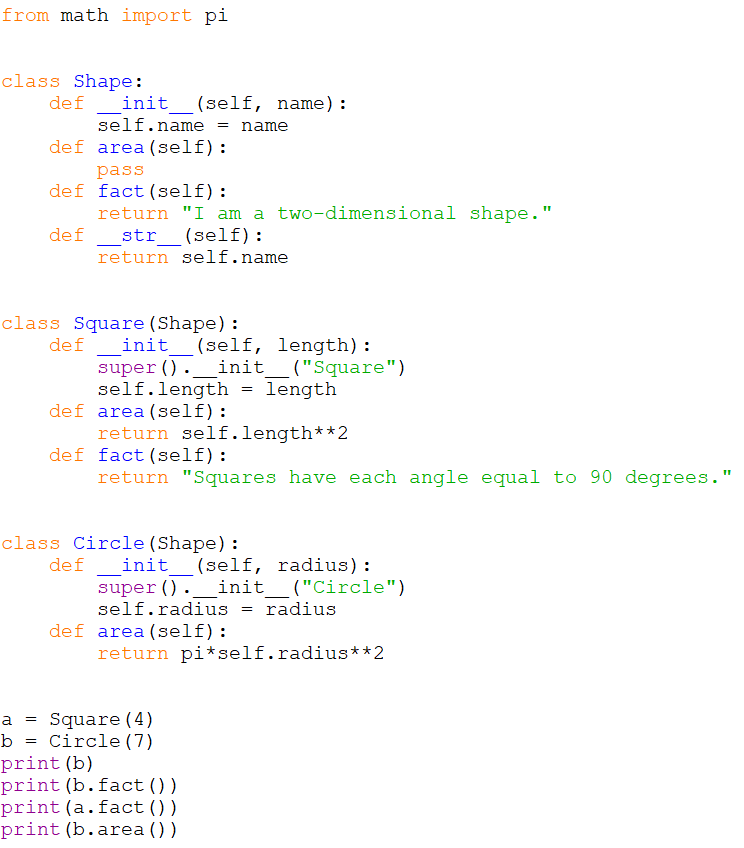
The child classes inherit methods and attributes from the parent class. Certain methods and attributes can be changed to fit the child class, which is known as Method Overriding. Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.



**Method Overriding**

Methods such as \_\_str\_\_(), which have not been overridden in the child classes, are used from the parent class. Due to polymorphism, the Python interpreter automatically recognises that the fact() method for object a(Square class) is overridden and uses the one defined in the child class. The fact() method for object b isn't overridden, so it is used from the Parent Shape class.





1. **What's inheritance in OOP?**

Inheritance is a very useful concept in OOP. We use inheritance to inherit properties of one class to another class. It means we can use the existing features on one class in another class. Simply it provides the facility of code re-usability. We don’t need to write same code in application again and again.

Example of hierarchical inheritance:

*Person class (Super class/Base class)*

*Programmer, Dancer, Singer (Child classes)*

* Class that provides attributes/properties to another class is known as parent class. Parent class is also called a Base class or Super class.
* Class that receives attributes/properties from another class is known as child class. Child class is also called a Sub class or Derived class.
* Important: In inheritance, a parent class never obtains features from its child class.

Diagram

Description automatically generated

**Inheritance Types:**

Diagram, timeline

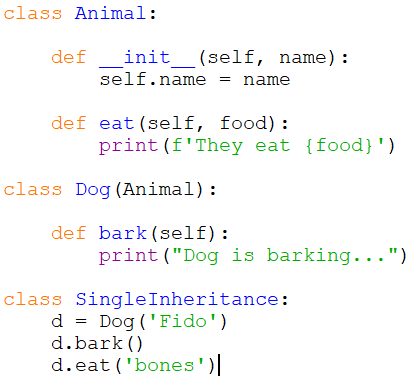
Description automatically generated

**Single Inheritance**

Diagram

Description automatically generated

The child class inherits properties and behaviour from single super class (parent class).



Super class is Animal and Sub class is Dog. We create a class called Animal and create a method eat(). If you add changes to Animal class it should affect whole code. We create a Dog class which extends from Animal class. Dog class contains all the properties of Animal class. So we can call features of both class.

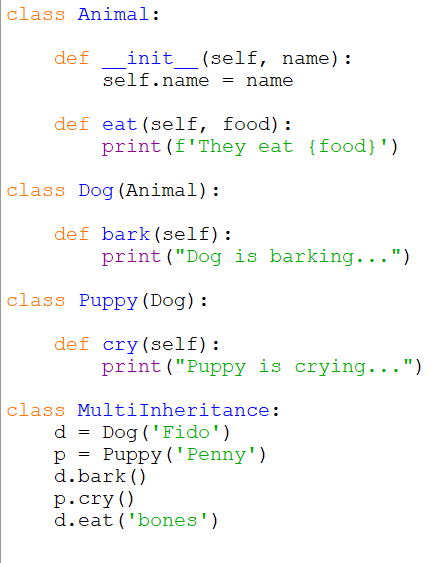
**Multilevel Inheritance**

A picture containing text, sign

Description automatically generated

Classes having more than one Super class at different levels are known as Multilevel Inheritance.

Here, Child1 class inherits the properties of the parent class. Child11 class inherits the properties of Child1 class. For Child11 class, Child1 class is a Super class. Child11 class implicitly inherits features of both Child1 class and Parent class.



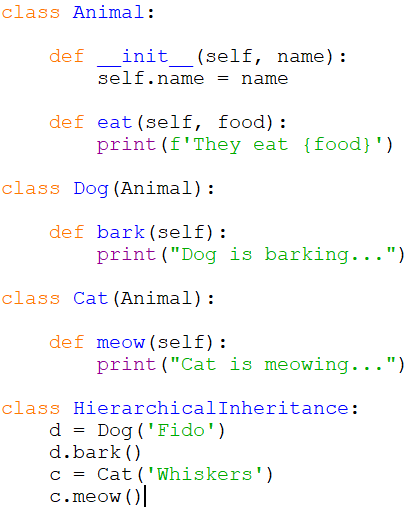
Puppy class can get all the properties in both Dog class and Animal class.

**Hierarchical Inheritance**

Diagram

Description automatically generated

If super class (parent class) has more than one sub class (child class) this is known as Hierarchical Inheritance. Both Child1 class and Child2 class are inheriting from Parent class.



Dog class and Cat class are Sub classes which are inheriting from Animal class (parent class).

**Hybrid Inheritance**

Diagram

Description automatically generated

Combination of Multilevel inheritance and Multiple inheritance.

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?**

• List of key requirements

* Frontend – access through website / app
* Search list of employees
* Voting buttons for 1st, 2nd, 3rd choices
* Visualisation of top 3 funniest people in the office
* Confirmation of voting
* Database – store employee and voter information
* Secure storage of data
* Maintain accurate information
* Backend – Python, API, server to get seat availability
* Secure access
* Fast and efficient retrieval of data
* Fault tolerant
* Locking of submitted selections

• Main considerations

* Scalability – how many users can use and access the voting system
* Accessibility – is the UI accessible to users with visual / cognitive / mobility needs
* Usability – user-friendly design, easy to access and navigate, quick and accurate
* Compatibility – system can be used on various devices e.g., PC / tablet / mobile
* Security – protecting user/subject data, passwords, backups
* Management / Maintenance – ease of use for administrators of the system

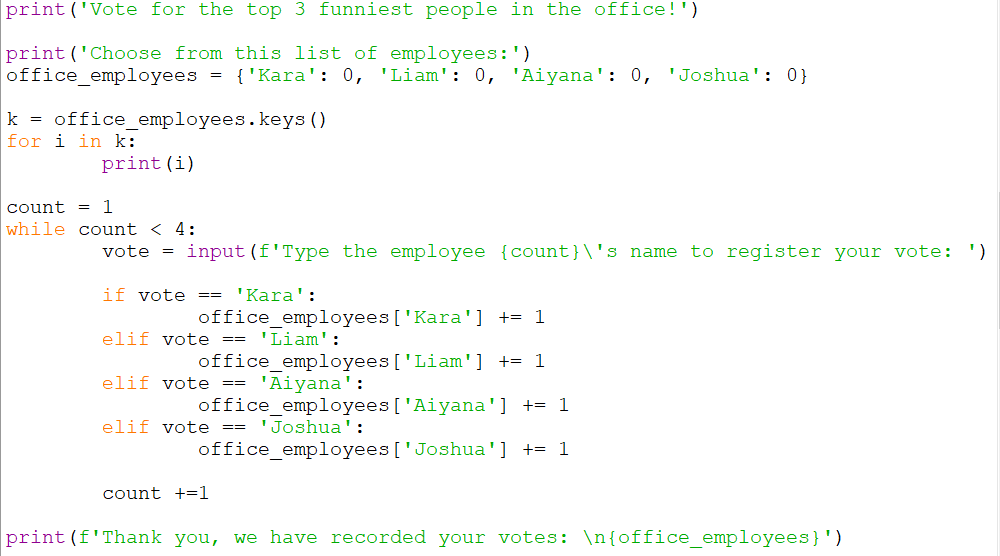
• What would be your common or biggest problems?

* Data protection / security / anonymity
* Consent from employees to feature on voting system
* Backup storage systems
* Manging traffic / scalability
* Maintaining up-to-date information e.g., consenting subjects
* Training administrators to use the system

• What components or tools would you potentially use?

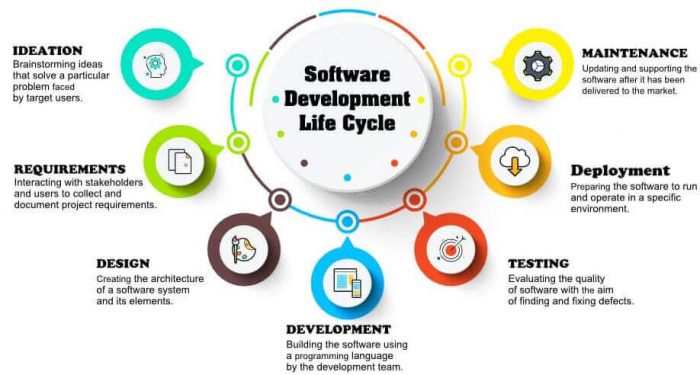
* Python celery for asynchronous code to make the user experience better (faster)
* DBMS – to manage locks for voting and to keep data accurate

**Example:**



1. **What is the software development cycle?**

The Software Development Cycle relates to the organisation and management of software development teams and models how software is taken from ideation to delivery in a series of steps. As software is a complex product, it is necessary to have clearly defined stages, deadlines, and responsibilities. Project managers and system analysts utilise software development life cycles to outline, design, develop, test, and eventually deploy information systems or software products as cheaply, efficiently, and effectively as possible, while maintaining overall quality.



**7 Stages of the System Development Life Cycle**

* Planning Stage
* Feasibility or Requirements of Analysis Stage
* Design and Prototyping Stage
* Software Development Stage
* Software Testing Stage
* Implementation and Integration
* Operations and Maintenance Stage

The software development process is a continuous cycle as there are almost always additional features and bug fixes to be designed, developed, and deployed. Depending on the methodology, these steps may be carried out in order (as with Waterfall), in parallel, or in repeating cycles (as with Agile).

**Ideation / Planning Stage**

The ideation or planning stage (also called the feasibility stage) is when developers plan for the upcoming project.

The planning phase involves Project Managers and Developers collaborating with Operations and Security teams to discuss requirements:

* Resource allocation (both human and materials)
* Capacity planning
* Project plans
* Project scheduling
* Cost estimations
* Provisioning
* Procurement requirements

Having an effective plan outlined for the SDC allows the teams to assess the scope of existing systems, foresee potential issues, secure funding / resources, outline clear objectives, and most importantly determine a schedule for the project.

**Requirements / Analysis Stage**

The analysis stage includes gathering all the specific details required for new system development and enhancement. These requirements are usually provided by business stakeholders and Subject Matter Experts (SMEs.). Architects, Development teams, and Product Managers work with them, and developers will often create a software requirement specification or SRS document, which outlines the business processes that need to be automated. This includes all the specifications for software, hardware, and network requirements for the system they plan to build and allows them to create prototypes.

Developers may:

* Define any prototype system requirements
* Evaluate alternatives to existing prototypes
* Perform research and analysis to determine the needs of end-users

**Design / Prototyping Stage**

The design stage is a necessary precursor to the main developer stage. Once the requirements are understood, software architects and developers can begin to design the software.

Developers will first outline the details for the overall application, along with more specific aspects:

* User interfaces
* System interfaces
* Network and network requirements
* Databases

To maintain consistency, developers use established [Design Patterns](https://en.wikipedia.org/wiki/Design_Patterns) to solve algorithmic problems. Architects may use an architecture framework such as TOGAF to construct an application from existing components, promoting reuse and standardisation.

Rapid prototyping allows for comparison of solutions to find the best fit:

* Design documents list the patterns and components selected for the project
* Code produced used as a starting point for main development

The SRS document is often transformed into a logical structure that can later be implemented in a programming language. Operation, training, and maintenance plans will all be produced so that developers have clear tasks throughout every stage of the cycle. Development managers will then prepare a design document to be referenced throughout the next phases of the SDLC.

**Development Stage**

Developers write the code and build the application according to design documents and specifications. This phase may be conducted in sprints (Agile) or as a single block (Waterfall) but regardless of methodology, development teams should produce testable, functional software as quickly as possible.

Developers choose the programming language to use based on the project specifications and requirements, follow the coding guidelines defined by the organisation and utilise different tools such as compilers, debuggers, and interpreters. To ensure that their expectations are being met, business stakeholders are engaged with and updated regularly.

**Testing Stage**

One of the most important phases of the SDLC is testing as it is impossible to deliver quality software without it. The output of this phase is functional software, ready for deployment to a production environment.

Testing necessary to measure quality:

* Code quality
* Unit testing (functional tests)
* Integration testing
* Performance testing
* Security testing

Developers test software rigorously, documenting any bugs or defects that need to be tracked, fixed, and later retested so that the end-user experience will not negatively be affected at any point.

Tests can be automated using Continuous Integration tools, like Codeship to ensure nothing is overlooked. It is very important that the software overall meets the quality standards previously defined in the SRS document.

**Deployment Stage**

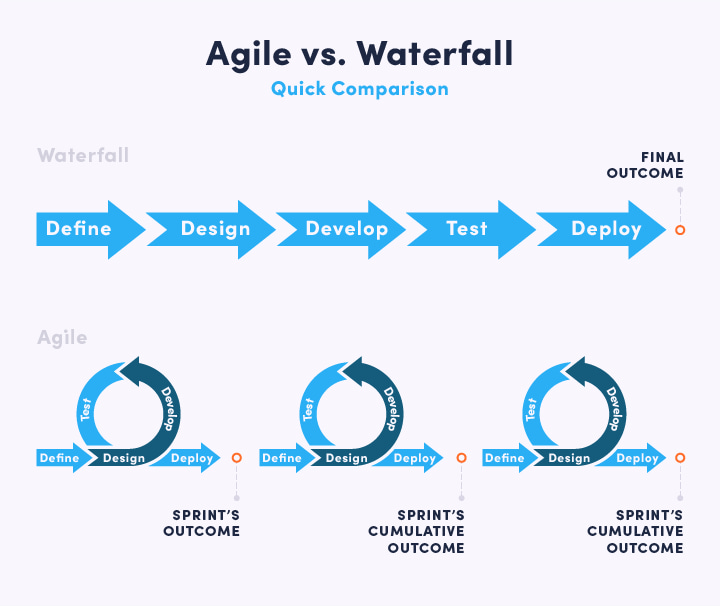
After testing, the different modules or designs will be integrated into the primary source code. Developers often use training environments to detect further errors or defects and may require approvals. To automate the deployment of applications to Production environments, Application Release Automation (ARA) tools are used. The output of this phase is the release to production of working software ready for market and may be provided to any end-users.

**Maintenance Stage**

Software must be monitored constantly to ensure proper operation, so the maintenance phase is not the end of the SDLC, it is more like the ‘end of the beginning’. Bugs and defects discovered in Production are reported by end-users and must be responded to by developers. Developers are responsible for implementing any changes that the software might need after deployment and it is necessary that they ensure any fixes do not introduce other problems (known as a [regression](https://en.wikipedia.org/wiki/Regression_testing)).

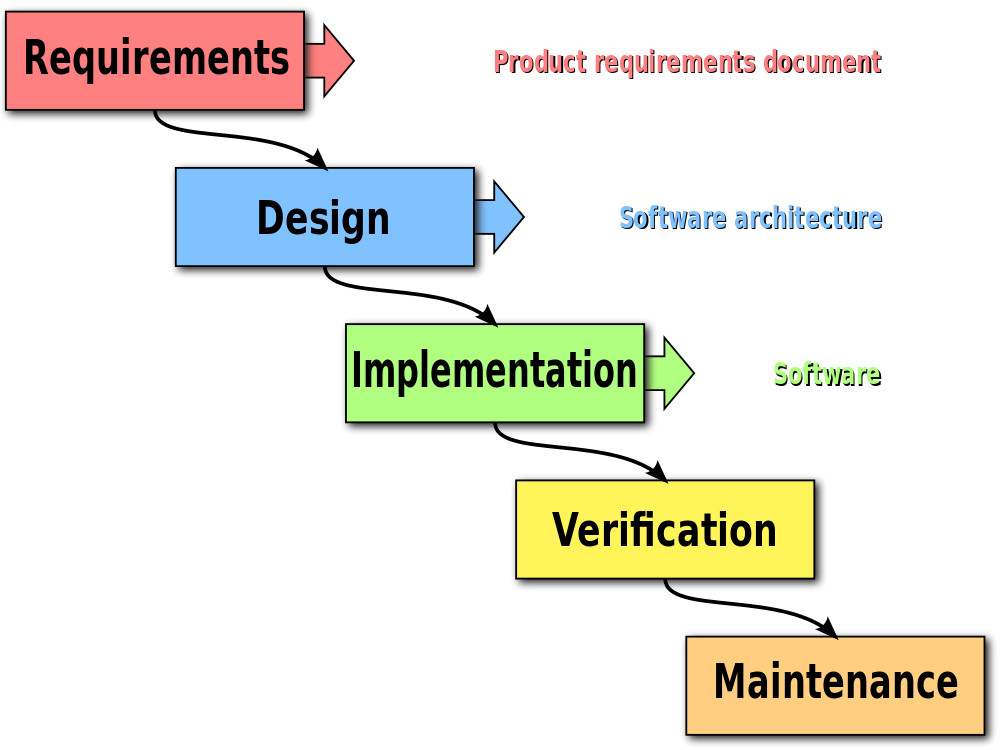
1. **What's the difference between agile and waterfall?**

Agile and Waterfall are two popular SDLC methodologies. They follow the same principles but do so in two different ways.



**Waterfall**

The waterfall model is the oldest of all SDLC methodologies and was adapted from traditional engineering. It is rigid, linear and straightforward, requiring development teams to finish each phase of the project completely before moving on to the next.

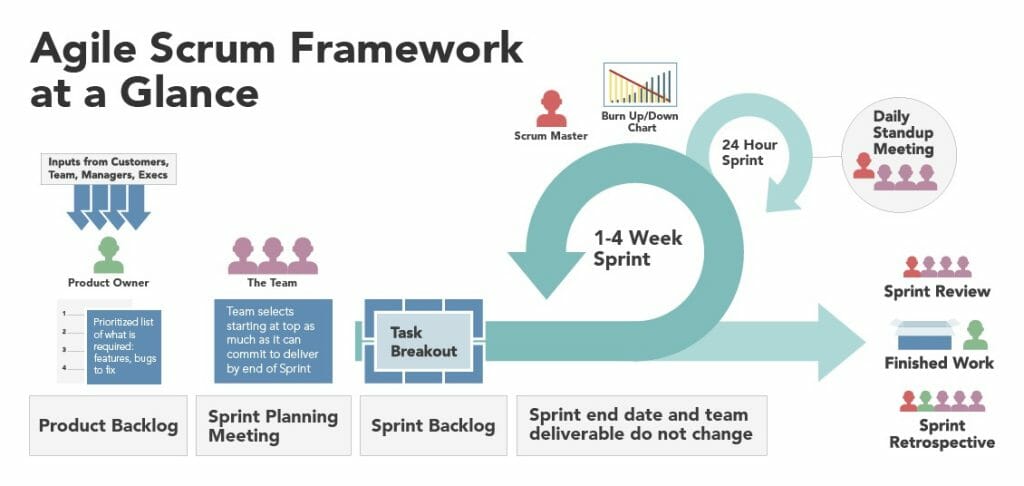


Waterfall methodology begins with long planning and design phases. The software is developed and goes through phases of testing, then it is finally deployed for use. Each stage has a separate project plan and takes information from the previous stage to avoid similar issues. However, as each stage relies on the completion of the previous, it is vulnerable to delays and bigger problems can arise for development teams further down the line.

Many consider Waterfall to be too rigid to adapt to changing requirements. It does not support feedback throughout the process, leading to the implementation of requirements that may have changed during the development effort. This weakness in Waterfall led to the development of more flexible methodologies, such as Agile.

**Agile**

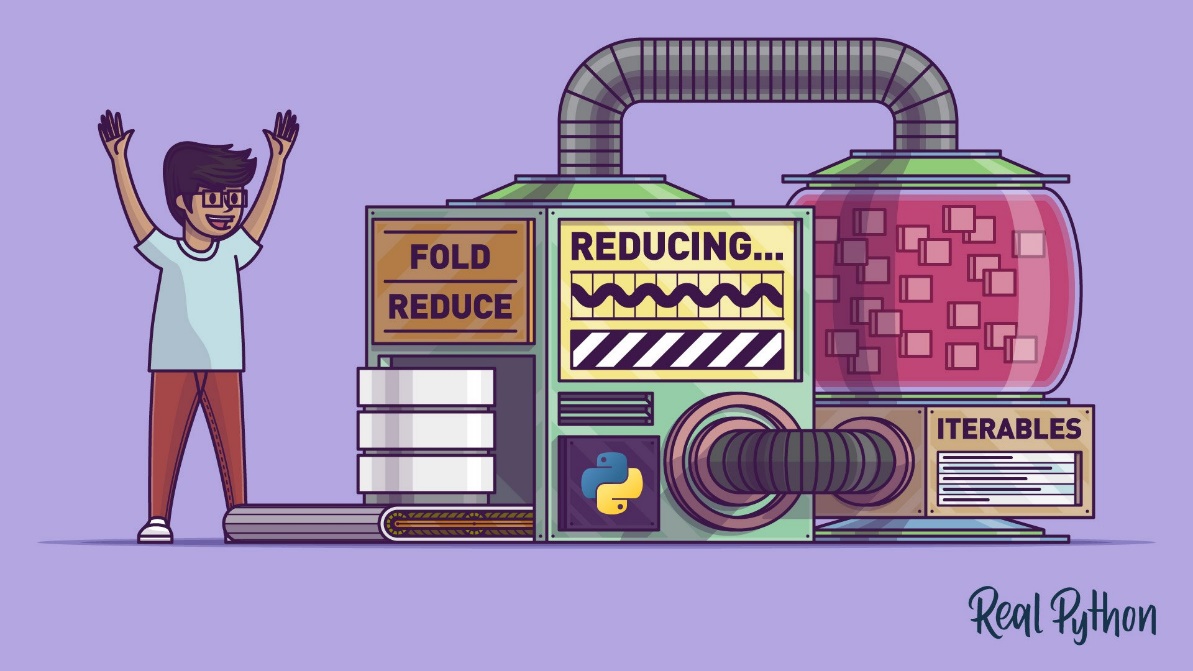
The agile methodology prioritises fast and ongoing release cycles, utilising small but incremental changes between releases and is well known in the industry. Compared to other methodologies, many more iterations and tests are completed. The advantage of this is that teams can address small issues as they arise rather than having to fix them at more complex stages of the project. Unlike Waterfall, Agile is a much more flexible framework that allows teams to better adapt to changes in requirements, and emphasises teamwork, prototyping, and regular feedback.



Several variants of Agile have emerged such as Scrum and Kanban. Scrum defines specific roles and events, known as ceremonies, whereas Kanban is simpler and more flexible. Agile teams often combine these and adapt a bespoke process that fits them best.

While Waterfall methodology is still commonly used, Agile is rapidly becoming more popular. Software is difficult to delivery reliably without a good methodology and as there is no perfect methodology, it is about choosing the approach that is the best fit for the project.

1. **What is a reduce() function used for?**



The function reduce() implements a mathematical technique called folding or reduction to reduce any iterable to a single cumulative value.

It performs the following steps:

* **Apply** a function (or callable) to the first two items in an iterable and generate a partial result.
* **Use** that partial result, together with the third item in the iterable, to generate another partial result.
* **Repeat** the process until the iterable is exhausted and then return a single cumulative value.

Originally, reduce() was a built-in function but was moved to functools.reduce() in Python 3.0 due to performance and readability concerns. The introduction of built-in functions like sum(), any(), all(), max(),

min(), and len(), which handle common use cases for reduce() in more efficient, readable, and Pythonic ways. It is useful for processing iterables without explicit for loops and as it is written in C, it tends to be faster.

In Python 3.x, if you need to use reduce(), then you first have to import the function in one of the following ways:

* import functools and then use fully-qualified names like functools.reduce().
* from functools import reduce and then call reduce() directly.

The reduce() function has the following signature:

functools.reduce(function, iterable[, initializer])

It is roughly equivalent to the following Python function:

def reduce(function, iterable, initializer=None):

it = iter(iterable)

if initializer is None:

value = next(it)

else:

value = initializer

for element in it:

value = function(value, element)

return value

As with the function above, reduce () applies a two-argument function to the items of iterable in a loop from left to right, reducing it to a single cumulative value. A third and optional argument called initializer can be used with reduce() to provide a seed value.

**The Required Arguments:**function and iterable

**Function:** a two-argument function applied to the items in an iterable to cumulatively compute a final value. You can pass any Python callable to reduce() as long as the callable accepts two arguments. Callable objects include classes, instances that implement a special method called \_\_call\_\_(), instance methods, class methods, static methods, and functions.

**Iterable:** accepts any Python iterable including lists, tuples, range objects, generators, iterators, sets, dictionary keys and values etc. The function will need to exhaust the iterator before a final value is given.

>>> def my\_add(a, b):

... result = a + b

... print(f"{a} + {b} = {result}")

... return result

This function my\_add() calculates the sum of a and b, prints a message with the operation using an [f-string](https://realpython.com/courses/python-3-f-strings-improved-string-formatting-syntax/), and returns the result of the computation.

>>> my\_add(5, 5)

5 + 5 = 10

10

As my\_add() is a two-argument function, you can pass it to Python’s reduce() with an iterable to compute the cumulated sum of the items in the iterable.

>>> from functools import reduce

>>> numbers = [0, 1, 2, 3, 4]

>>> reduce(my\_add, numbers)

0 + 1 = 1

1 + 2 = 3

3 + 3 = 6

6 + 4 = 10

10

When reduce() is called with my\_add() and numbers passed as arguments, reduce() performs several operations to calculate a final result of 10.

**The Optional Argument:**initializer

If you supply a value to initializer, then reduce() will pass it to the first call of function as its first argument to perform its first partial computation. reduce() then continues calculations with the subsequent items of iterable.

Example of my\_add() with initializer set to 100:

>>> from functools import reduce

>>> numbers = [0, 1, 2, 3, 4]

>>> reduce(my\_add, numbers, 100)

100 + 0 = 100

100 + 1 = 101

101 + 2 = 103

103 + 3 = 106

106 + 4 = 110

110

Python’s reduce() uses the initializer value (100) in the first call as the first argument to my\_add(). In the first iteration, my\_add() uses 100 and the first item of numbers (0) to perform the calculation 100 + 0 = 100. Therefore, if you supply a value to initializer, reduce() will perform one more iteration than it would without an initializer.

Python’s reduce() will use the initializer as its default return value when iterable is empty. If you don’t provide an initializer value, then reduce() will raise a TypeError.

>>> from functools import reduce

>>> # Using an initializer value

>>> reduce(my\_add, [], 0) # Use 0 as return value

0

>>> # Using no initializer value

>>> reduce(my\_add, []) # Raise a TypeError with an empty iterable

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: reduce() of empty sequence with no initial value

**Conclusion**

Reduction operations can be performed on iterables using Python callables and lambda functions with reduce(). A function is applied to the items in an iterable that reduces them to a single cumulative value. Although Pythonic tools like sum(), min(), max() all(), any(), have replaced reduce() over time, it is still used by many functional programmers.

1. **How does merge sort work?**

The merge sort algorithm takes a divide and conquer approach to problem solving that is made up of two subroutines – the split subroutine and the merge subroutine. It begins by splitting an unordered list of items into two halves called sublists. This process is then repeated, splitting the sublists until it reaches sublists of single elements. These single element sublists are then merged in pairs to form sublists of two items. During the merging the items are ordered. This process is repeated until finally the two ordered halves are merged to form a single ordered list.

Example: sort a group of seven numbered cards into ascending order using merge sort.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **9** | **3** | **10** | **2** | **8** | **5** | **1** |

**Splitting**

The group of cards is repeatedly split into half. The process is repeated until you have 7 separate groups with a single card each.

**Step 1**

First, split the cards in half by finding the card that is in the middle of the group. As there are seven cards, the middle card is in the fourth position. Then 'split' the list at the middle card and create two sublists – one half with every item up to and including the middle card, and the other group with everything after the middle card.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 1** | **group 1** | | | | **group 2** | | |
| 9 | 3 | 10 | 2 | 8 | 5 | 1 |

**Step 2**

The splitting process is repeated with the two new groups: you find the middle point of each and split each group in half.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 2** | **group 1** | | **group 2** | | **group 3** | | **group 4** |
| 9 | 3 | 10 | 2 | 8 | 5 | 1 |

**Step 3**

Split each group of cards in half again. There are now seven separate groups, each containing a single card.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 3** | **group 1** | **group 2** | **group 3** | **group 4** | **group 5** | **group 6** | **group 7** |
| 9 | 3 | 10 | 2 | 8 | 5 | 1 |

**Merging**

The sublists of single cards now need to be reassembled by merging the cards in order. Each half of the group is split and sorted separately, and then the two sorted halves are merged at the end.

**Step 4**

Merge the first two groups of single cards to create a new combined ordered group. Compare the first group’s value to the value of the second group, the lowest value card becomes the first card in the new combined group and the other card takes the second place. The process is repeated for the other groups. The last group has nothing to merge with so it is a sorted group.

* compare 9,9 and 3,3, place 3,3 and then 9,9 in the new merged group (group 1)
* compare 10,10 and 2,2, place 2,2 and then 10,10 in the new merged group (group 2)
* compare 8,8 and 5,5, place 5,5 and then 8,8 in the new merged group (group 3)
* place 1,1 in group 4

**Previous step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 3** | **group 1** | **group 2** | **group 3** | **group 4** | **group 5** | **group 6** | **group 7** |
| 9 | 3 | 10 | 2 | 8 | 5 | 1 |

**Current step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 4** | **group 1** | | **group 2** | | **group 3** | | **group 4** |
| 3 | 9 | 2 | 10 | 5 | 8 | 1 |

**Step 5**

The process is repeated to merge group 1 and group 2.

* consider the first card in each group; compare the two values (3,3 and 2,2); take the lowest value card (2,2) to become the first card of the new merged group; you now are left with two cards in group 1 and one card in group 2
* consider the first of the remaining cards in each group; compare the two values (3,3 and 10,10); take the lowest value card (3,3) to become the next card of the new merged group; you now have one card in group 1 and one card in group 2
* consider the first of the remaining cards in each group; compare the two values (9,9 and 10,10); take the lowest value card (9,9) to become the next card of the new merged group; you now have one card left in group 2
* there is only one card left (10,10); place it into the new merged group
* both groups are now empty

Repeat the process to merge group 3 and 4:

* compare 5,5 and 1,1, place 1,1 in the new merged group
* place 5,5 in the new merged group
* place 8,8 in the new merged group

The algorithm is very efficient as when you have only one remaining group with cards to place, you can just place them right away in the merged group as they were already ordered.

**Previous step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 4** | **group 1** | | **group 2** | | **group 3** | | **group 4** |
| 3 | 9 | 2 | 10 | 5 | 8 | 1 |

**Current step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 5** | **group 1** | | | | **group 2** | | |
| 2 | 3 | 9 | 10 | 1 | 5 | 8 |

**Step 6**

This leaves you with just the final merge to do.

* compare 2,2 and 1,1, place 1,1
* compare 2,2 and 5,5, place 2,2
* compare 3,3 and 5,5, place 3,3
* compare 9,9 and 5,5, place 5,5
* compare 9,9 and 8,8, place 8,8
* place 9,9
* place 10,10

**Previous step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 5** | **group 1** | | | | **group 2** | | |
| 2 | 3 | 9 | 10 | 1 | 5 | 8 |

**Current step:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Step 6** | **sorted group** | | | | | | |
| 1 | 2 | 3 | 5 | 8 | 9 | 10 |

**Merging two lists**

The algorithm combines two ordered lists (or single item lists) by merging the items in ascending order into a single ordered list. The subroutine takes three arguments:

* merged – the list that will hold the ordered items after merging the two lists
* list\_1 – the first list to be merged
* list\_2 – the second list to be merge

SUBROUTINE merge (merged, list1, list2)

index\_1 ← 0 # list\_1 current position

index\_2 ← 0 # list\_2 current position

index\_merged ← 0 # merged current position

# While there are still items to merge

WHILE index\_1 < LEN(list\_1) AND index\_2 < LEN(list\_2)

# Find the lowest of the two items being compared

# and add it to the new list

IF list\_1[index\_1] < list\_2[index\_2] THEN

merged[index\_merged] ← list\_1[index\_1]

index\_1 ← index\_1 + 1

ELSE

merged[index\_merged] ← list\_2[index\_2]

index\_2 ← index\_2 + 1

ENDIF

index\_merged ← index\_merged + 1

ENDWHILE

# Add to the merged list any remaining data from list\_1

WHILE index\_1 < len(list\_1)

merged[index\_merged] ← list\_1[index\_1]

index\_1 ← index\_1 + 1

index\_merged ← index\_merged + 1

ENDWHILE

# Add to the merged list any remaining data from list\_2

WHILE index\_2 < len(list\_2)

merged[index\_merged] ← list\_2[index\_2]

index\_2 ← index\_2 + 1

index\_merged ← index\_merged + 1

ENDWHILE

ENDSUBROUTINE

The subroutine is merging two ordered lists (list\_1 and list\_2), into a single list (merged). It needs a set of variables to keep track of where it is in each list:

* The variable index\_1 keeps track of the current position in list\_1
* The variable index\_2 keeps track of the current position in list\_2
* The variable index\_merged keeps track of the current position in merged

The first while loop

* The values from list\_1[index\_1] and list\_2[index\_2] are compared
* The lower of the two values is written to the merged list merged at the next empty position indicated by index\_merged
* The pointer for the list that the item came from is incremented by 1;
  + if the item came from list1, index\_1 is incremented by 1;
  + if the item came from list\_2, index\_2 is incremented by 1
* The comparison and merging continues until one list is completely empty

The second while loop:

Any remaining items from list\_1 are added to merged as they are already sorted

The third while loop:

Any remaining items from list\_2 are added to merged as they are already sorted

**Splitting and merging using recursion**

The algorithm for the merge\_sort subroutine is [recursive](https://isaaccomputerscience.org/topics/recursion). It uses the merge subroutine above to merge the sublists that are created as the algorithm progresses.

SUBROUTINE merge\_sort(items)

# Base case for recursion:

# The recursion will stop when the list has been divided into single items

IF LEN(items) < 2 THEN

return

ELSE

midpoint ← (LEN(items) - 1) DIV 2 # Calculate midpoint

left\_half ← items[0:midpoint] # Create left half list

right\_half ← items[midpoint+1:LEN(items)-1] # Create right half list

merge\_sort(left\_half) # Recursive call on left half

merge\_sort(right\_half) # Recursive call on right half

merge (items, left\_half, right\_half) # Call procedure to merge both halves

ENDIF

ENDSUBROUTINE

**Finding the midpoint and splitting the list**

Each time the recursive subroutine is called, as long as there is more than one item in the list, the algorithm finds the midpoint and splits the list using integer division.

In the pseudocode, the left and right sublists are defined as follows:

left\_half = items[0:midpoint] // Create left half list  
right\_half = items[midpoint+1:LEN(items)-1] // Create right half list

* The left\_half sublist includes all of the items from position 0 up to and including the item with position midpoint.
* The right\_half sublist includes all of the items from position midpoint + 1 up to and including the last item in the list with position LEN(items) - 1.

The expression midpoint ← (LEN(items) - 1) DIV 2 calculates the index (position) of the midpoint, regardless of whether the list has an even or an odd number of items.

* For example, if the original list has seven items, the midpoint (the first time the subroutine is called) is midpoint ← (7 - 1) DIV 2 = 3, which means that the left sublist will include the items in positions 0 to 3, and the right sublist will include the items in positions 4 to 6.
* If the original list has eight items, the midpoint (the first time the subroutine is called) is midpoint = (8 - 1) DIV 2 = 3, which means that the left sublist will include the items in positions 0to 3, and the right sublist will include the items in positions 4 to 7.

#### Splitting and merging recursively

* Once the midpoint of the original items list has been calculated, the left sublist (the left half of the items list) is then passed to another instance of the merge sort subroutine, to be further split until there are lists of just one item (which is the recursion **base case**). The algorithm then orders the items of the left sublist. It does that by calling the merge subroutine and merging the single items back into the sublist, placing each item in the correct order.
* Once this process has finished, the algorithm will split the right sublist (the right half of the items list). Once the right sublist has been split into single items (which is the recursion **base case**), the subroutine orders the items of the right sublist. It does that by calling the merge subroutine and merging the single items back into the sublist, placing each item in the correct order.
* Once the two halves (the left and right halves of the original items list) are sorted, the merge subroutine is called one last time to merge them into the final sorted list.

With recursion, each sublist is considered three times:

* first the subroutine processes the left sublist.
* second it processes the right sublist.
* finally, it merges the sublists.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **items** | **left** | **right** | **merged** | **explanation** |
| 9, 3, 10, 2, 8, 5, 1 | 9, 3, 10, 2 |  |  | Take the left half of the original list to create a sublist. |
| 9, 3, 10, 2 | 9, 3 |  |  | Take the left half of the sublist. |
| 9, 3 | 9 |  |  | Take the left half of the sublist. |
| 9 |  |  |  | Reached the base case of a single item. |
| 9, 3 |  | 3 |  | Take the right half of the sublist. |
| 3 |  |  |  | Reached the base case of a single item. |
| 9, 3 | **9** | **3** | **3, 9** | **Merge the left and right single items so that they are ordered.** |
| 9, 3, 10, 2 |  | 10, 2 |  | Move to the right half of the original left sublist. |
| 10, 2 | 10 |  |  | Take the left half of the sublist. |
| 10 |  |  |  | Reached the base case of a single item. |
| 10, 2 |  | 2 |  | Take the right half of the sublist. |
| 2 |  |  |  | Reached the base case of a single item. |
| 10, 2 | **10** | **2** | **2, 10** | **Merge the left and right single items so that they are ordered.** |
| 9, 3, 10, 2 | **3, 9** | **2, 10** | **2, 3, 9, 10** | **Merge the two sublists. The left half of the original list is now ordered.** |
| 9, 3, 10, 2, 8, 5, 1 |  | 8, 5, 1 |  | Take the right half of the original list and repeat the steps. |
| 8, 5, 1 | 8, 5 |  |  | Take the left half of the sublist. |
| 8, 5 | 8 |  |  | Take the left half of the sublist. |
| 8 |  |  |  | Reached the base case of a single item. |
| 8, 5 |  | 5 |  | Take the right half of the sublist. |
| 5 |  |  |  | Reached the base case of a single item. |
| 8, 5 | **8** | **5** | **5, 8** | **Merge the left and right single items so that they are ordered.** |
| 8, 5, 1 |  | 1 |  | Take the right half of the sublist. |
| 1 |  |  |  | Reached the base case of a single item. |
| 8, 5, 1 | **5, 8** | **1** | **1, 5, 8** | **Merge the two sublists. The right half of the original list is now ordered.** |
| 9, 3, 10, 2, 8, 5, 1 | **2, 3, 9, 10** | **1, 5, 8** | **1, 2, 3, 5, 8, 9, 10** | **Merge the left and right halves of the original list. The original list is now sorted in ascending order.** |

### **Time complexity**

The time complexity of a merge sort is **O(*nlogn*)**. To calculate time complexity, consider:

1. Splitting an original list with *n* items until we have *n* sublists of single items
2. Merging *n* sublists until we have one ordered list of *n* items

### **Splitting**

### The splitting operation repeatedly divides the list in half. If we take an initial list of n elements and keep dividing by 2, after log2​n steps we will have n sublists of 1 item. Calculating the logarithm of a number in base 2 tells you how many times a number can be divided by 2 before reaching 1:

* After 1 split there will be 2 sublists with n/2 items each
* After 2 splits there will be 4 sublists with n/4 items each
* After 3 splits there will be 8 sublists with n/8 items each
* After log2​n splits there will be n sublists with 1 item each

### Therefore, the time complexity of spitting is O(logn), as the list is divided log(n) times (where n is the number of items in the list).

### **Merging**

### During the merge process, the sublists are combined by examining every item in turn and placing it into order in a merged list (i.e. merged in the pseudocode). In the worst case scenario, there will be n comparisons to order all n items for each merge.

### Depending on the value of the items in the sublists, sometimes not all n items will need to be compared. Once all the items from one sublist have been placed, the remaining items in the other sublist can just be added to the end of the merged list, because they are already in order. However, this does not affect the overall time complexity as, in the worst case, all n items must be compared.

### **Splitting and merging combined**

### There are log2​n splits and, for every instance of splitting, n items are compared and merged, i.e. the merging of the n items repeats log2​n times. Therefore, splitting and merging involve nlog2​n operations, which means that the time complexity of merge sort is O(nlogn). The complexity is the same in all three cases (worst, best and average) as the full split and merge process is always executed.

### **Space complexity**

### The merge sort needs extra space for the left and right sublists that are created whenever items are split (i.e. on each recursive call). If the original list has n items, then the sublists will also need to hold n items in total. This means that the algorithm needs an additional amount of space equal to that used by the original list of items. Therefore, the space complexity of the algorithm is O(n).

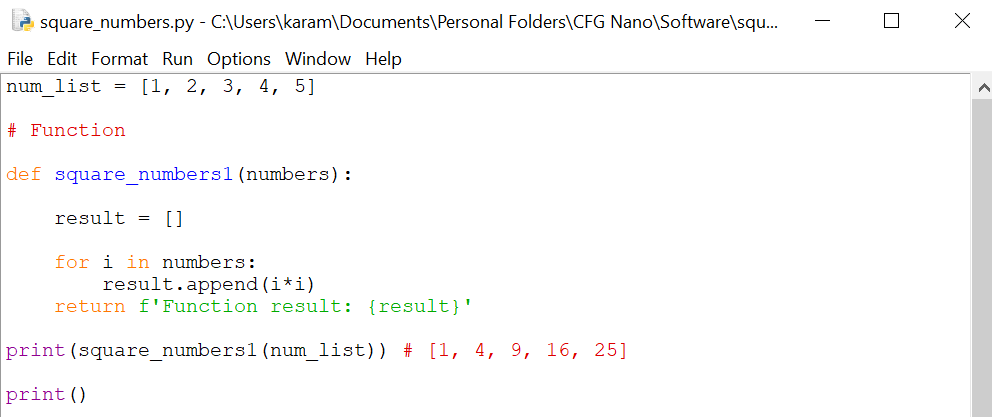
1. **Generators – Generator functions allow you to declare a function that behaves like an iterator, i.e., it can be used in a for loop. What is the use case?**

Functions can be transformed into generator functions with a simple change – instead of using the ‘return’ keyword, we instead use ‘yield.’ Generators return objects (iterators) by implementing \_\_iter\_\_() and \_\_next\_\_() methods automatically, but do not start execution until there is a call.

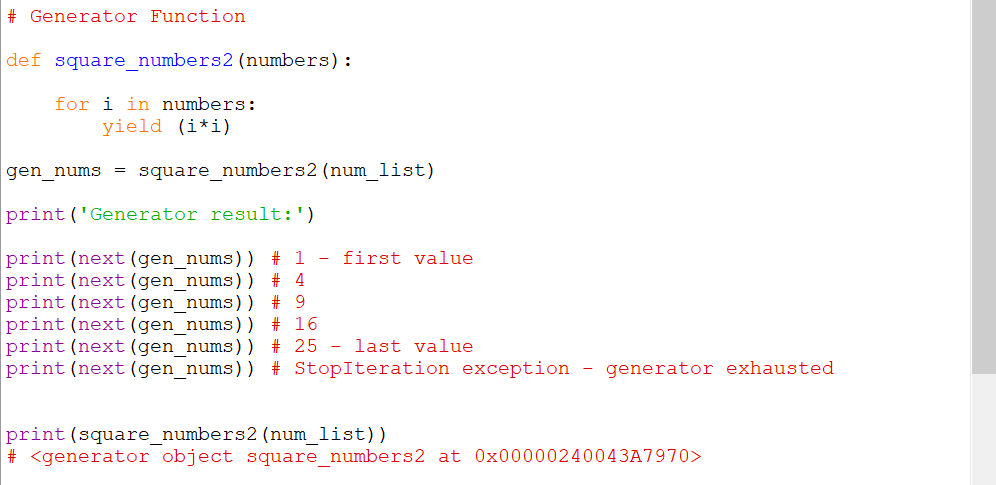
Generators don’t require as much space in memory as they yield one result at a time and keep state (track) of the current position in the object between calls. While ordinary functions terminate completely after ‘return’, the ‘yield’ keyword causes the generator function to pause on the current value, then the next() method causes the generator function to return the next value in the object. This efficient use of memory is particularly important for large lists and when memory space is limited.

Comparison of a function and a generator function:

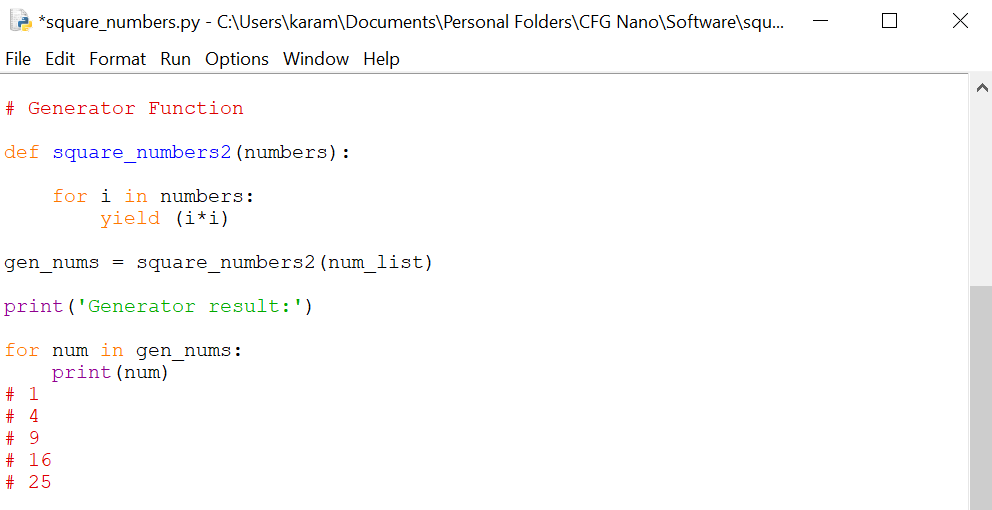
*The function square\_numbers1 takes a list of numbers and iterates through the list using a for loop. Each number is squared then the result is appended to the initially empty result list. This entire list is stored in memory and returned by the function.*

**

*The generator function square\_numbers2 does not require an empty list as it yields each value while keeping state. As 25 is the last value to be yielded, a Stop Iteration exception is thrown on the 6th time next() is used – the generator is exhausted and out of values.*



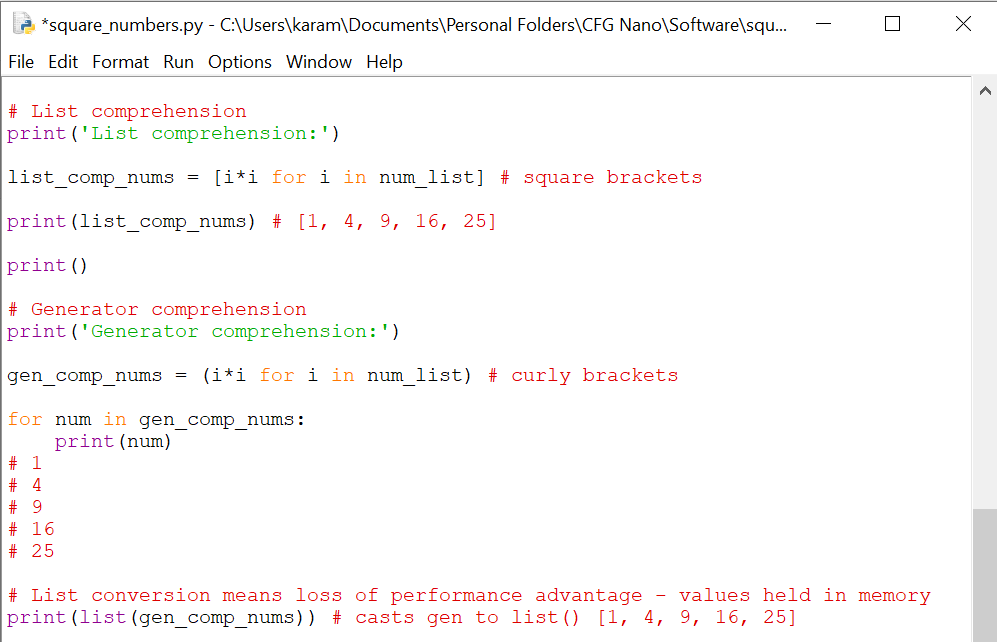
*For loops can also be used with generators to output each value in the object:*



List vs generator comprehension:

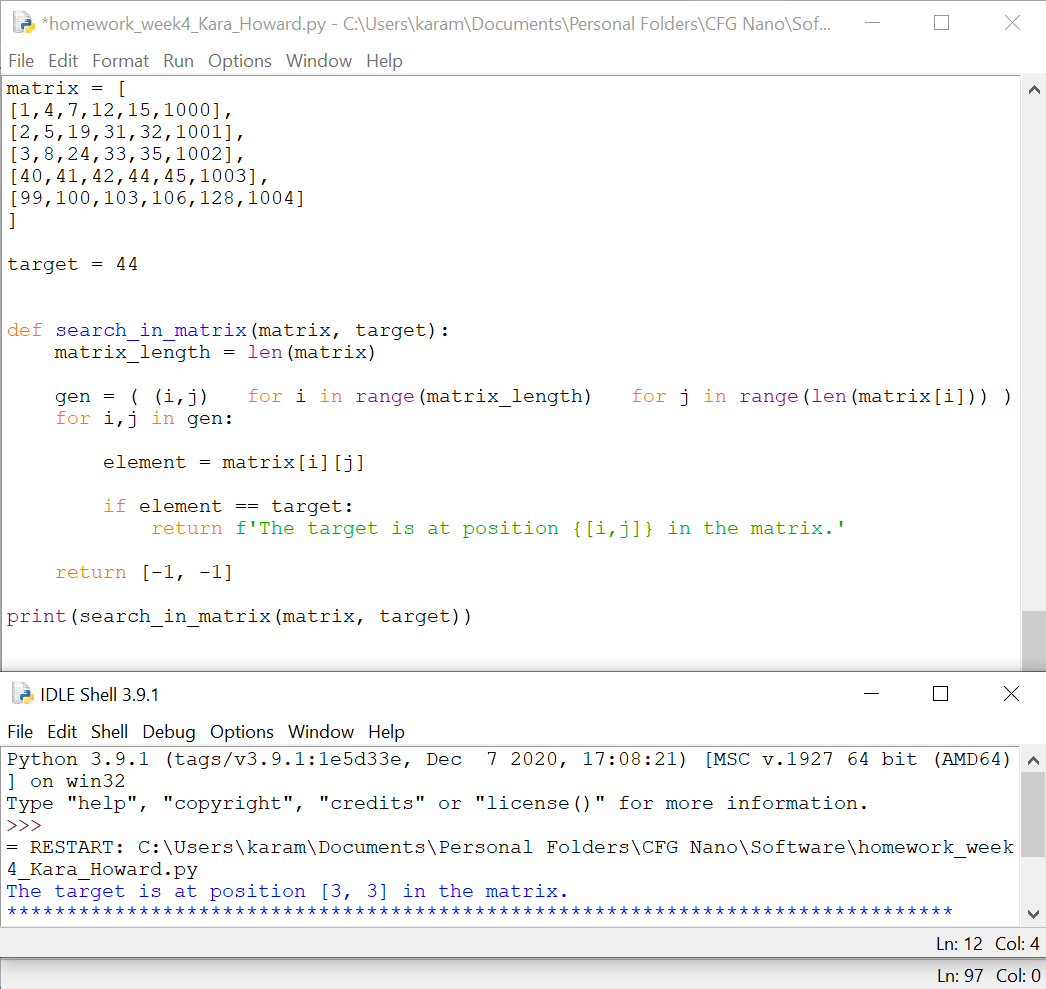
*Python generator expressions are similar to anonymous lambda functions. Generator comprehensions are often used as they are less memory intensive compared to list comprehension; however, they tend to take more time to run.*

*Instead of the square brackets used in list comprehension, generator comprehension uses curly brackets. They output one item at a time unlike list comprehension, which outputs the entire list. The output from the generator expression can be converted to a list, but the performance advantage is lost as the list has to be stored in memory.*



Example of a generator expression within a function:

*Generator comprehension ‘gen’ returns a generator object that can be iterated over to compare each value in the matrix to the target value. The search\_in\_matrix() function returns the position of the target if found, otherwise [-1, -1] is returned.*



**Conclusion:**

* Generators are much easier to implement than iterator classes
* They are more efficient in terms of memory (particularly useful for large data sets)
* They can be used to represent an infinite stream of data as only one item is produced at a time
* Multiple generators can be used to pipeline output of functions clearly and efficiently

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

Python decorators help improve code readability and reusability by abstraction. Decorators are a powerful concept that allow you to ‘wrap’ a function with another function and allow you to alter how a function behaves without changes being made to the code.

@my\_decorator

def hello():

print('hello')

The function hello() is a function object. @my\_decorator is a function that can use the hello object and return a different object to the interpreter. The decorator can swallow the function or return an object that is not a function. A decorator is simply a function that is passed another function and returns an object. Any function can be used as a decorator.

To start writing a decorator you define a function:

def my\_decorator(f):

return 5

The decorator is passed a function and returns a different object. Below, it swallows the function it is passed and always just returns 5.

@my\_decorator

def hello():

print('hello')

>>> hello()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'int' object is not callable

'int' object is not callable

Because the decorator's return value *replaces* hello, the decorator is returning an int, and not a callable, so it cannot be called as a function.

>>> hello

5

Usually, you want the decorator to return an object that mimics the decorated function, so the returned object needs to be a function itself.

To print every time the function is called, you write a function that prints that information, then calls the function. That function would need to be returned by the decorator through function nesting:

def mydecorator(f): # f is the function passed to us from python

def log\_f\_as\_called():

print(f'{f} was called.')

f()

return log\_f\_as\_called

The function hello() can now be called like a standard function:

@mydecorator

def hello():

print('hello')

We get the following output:

>>> hello()

<function hello at 0x7f27738d7510> was called.

hello

**Wrapping Functions Correctly**

A function can be decorated many times and the decorators then cause a chain effect. The top decorator passes the object to the next, and so on.

@a

@b

@c

def hello():

print('hello')

The interpreter is essentially doing hello = a(b(c(hello)))

@mydecorator

@mydecorator

def hello():

print('hello')

>>> hello()

<function mydec.<locals>.a at 0x7f277383d378> was called.

<function hello at 0x7f2772f78ae8> was called.

hello

In the above example, the first decorator wrapped the second and printed it separately. The first line printed <function mydec.<locals>.a at 0x7f277383d378> instead of what the second line printed. The is because the object returned by the decorator is a new function, not called hello. This can often break tests and things that might be trying to introspect the function attributes.

If the decorator is to act like the function it decorates, it needs to also mimic that function. In the functools module in the python standard library, there is a decorator called wraps that facilitates this:

import functools

def mydecorator(f):

@functools.wraps(f) # we tell wraps that the function we are wrapping is f

def log\_f\_as\_called():

print(f'{f} was called.')

f()

return log\_f\_as\_called

@mydecorator

@mydecorator

def hello():

print('hello')

>>> hello()

<function hello at 0x7f27737c7950> was called.

<function hello at 0x7f27737c7f28> was called.

hello

The new function appears like the one it is decorating but still relies on the fact that it accepts no input returns nothing. You can modify the function to pass in arguments and return the same value:

import functools

def mydecorator(f):

@functools.wraps(f) # wraps is a decorator that tells our function to act like f

def log\_f\_as\_called(\*args, \*\*kwargs):

print(f'{f} was called with arguments={args} and kwargs={kwargs}')

value = f(\*args, \*\*kwargs)

print(f'{f} return value {value}')

return value

return log\_f\_as\_called

All the inputs the function receives, and what it returns are printed every time the function is called. Any existing function can now be decorated and debug logging on all inputs and outputs can be obtained without needing to manually write the logging code.

**Strengths:**

* you can make an extension for class-methods without inheritance
* you could mix inheritance with decorators
* you still have the same class-object

**Limitations:**

* all methods need to be initialised and re-assigned at init
* for each method you need to create another method in decorator (not flexible enough)
* decorator is limited to this class methods – cannot be reused at different class with different methods