Table of Contents

[References 13](#_Toc176091357)

[09th March 2024 14](#_Toc176091358)

[Installation 14](#_Toc176091359)

[Variables 14](#_Toc176091360)

[Type method 14](#_Toc176091361)

[del 14](#_Toc176091362)

[Unpacking or multiple declarations 14](#_Toc176091363)

[F-Strings 15](#_Toc176091364)

[Operators 15](#_Toc176091365)

[Input 16](#_Toc176091366)

[Format input 16](#_Toc176091367)

[10th March 17](#_Toc176091368)

[Python Data Structure 17](#_Toc176091369)

[Strings 17](#_Toc176091370)

[triple quotes 17](#_Toc176091371)

[methods 18](#_Toc176091372)

[Convert List to string 19](#_Toc176091373)

[Convert String to List 19](#_Toc176091374)

[Lists 19](#_Toc176091375)

[len 19](#_Toc176091376)

[append 20](#_Toc176091377)

[remove 20](#_Toc176091378)

[Indexing 20](#_Toc176091379)

[index 20](#_Toc176091380)

[extend 20](#_Toc176091381)

[pop 21](#_Toc176091382)

[count 21](#_Toc176091383)

[sort 21](#_Toc176091384)

[Mutability 21](#_Toc176091385)

[Tuple 22](#_Toc176091386)

[Dictionaries 22](#_Toc176091387)

[Print Specific key value 22](#_Toc176091388)

[Print Key and Values 22](#_Toc176091389)

[Keys 23](#_Toc176091390)

[Values 23](#_Toc176091391)

[Sets 23](#_Toc176091392)

[Remove vs Discard 23](#_Toc176091393)

[Tuple vs List vs Set vs Dictionary 23](#_Toc176091394)

[Iterators 24](#_Toc176091395)

[for Loop 24](#_Toc176091396)

[Iterating over Sequences 24](#_Toc176091397)

[Known Number of Iterations 24](#_Toc176091398)

[Enumeration 24](#_Toc176091399)

[Iteration with Iterators 25](#_Toc176091400)

[while Loop 25](#_Toc176091401)

[Unknown Number of Iterations 25](#_Toc176091402)

[Condition-based Iteration 26](#_Toc176091403)

[Looping with State 26](#_Toc176091404)

[Looping with Dynamic Conditions 26](#_Toc176091405)

[For vs While 27](#_Toc176091406)

[Common Applications 27](#_Toc176091407)

[Control Flow Statements 27](#_Toc176091408)

[break 27](#_Toc176091409)

[continue 27](#_Toc176091410)

[else 27](#_Toc176091411)

[16th March, 2024 28](#_Toc176091412)

[Slicing 28](#_Toc176091413)

[Slicing a string 28](#_Toc176091414)

[Slicing a list 28](#_Toc176091415)

[Slicing a tuple 28](#_Toc176091416)

[Escape Sequences 28](#_Toc176091417)

[Conditional statements 29](#_Toc176091418)

[If, elif else 29](#_Toc176091419)

[Ternary Operator 30](#_Toc176091420)

[17th March 31](#_Toc176091421)

[Functions 31](#_Toc176091422)

[Arguments 31](#_Toc176091423)

[Return Values 31](#_Toc176091424)

[Function Call 31](#_Toc176091425)

[Scope 32](#_Toc176091426)

[OOPs Concept 32](#_Toc176091427)

[Encapsulation 32](#_Toc176091428)

[Abstraction 33](#_Toc176091429)

[Inheritance 33](#_Toc176091430)

[Global Variables 34](#_Toc176091431)

[23rd March 35](#_Toc176091432)

[Functions as Generators 35](#_Toc176091433)

[Return vs Yield 36](#_Toc176091434)

[Real Time Applications 36](#_Toc176091435)

[Using Modules 37](#_Toc176091436)

[Creating a Module 37](#_Toc176091437)

[Importing a Module 37](#_Toc176091438)

[Importing Specific Functions or Variables 38](#_Toc176091439)

[Renaming Imported Modules or Symbols 38](#_Toc176091440)

[Accessing Module Documentation: 38](#_Toc176091441)

[Classes 39](#_Toc176091442)

[Defining a Class 39](#_Toc176091443)

[Creating Objects (Instances) 39](#_Toc176091444)

[Accessing Attributes and Calling Methods: 39](#_Toc176091445)

[Self key word 40](#_Toc176091446)

[Within Instance Methods 40](#_Toc176091447)

[Accessing Instance Attributes 40](#_Toc176091448)

[Referencing the Current Instance 40](#_Toc176091449)

[Differentiating Between Instance and Class Variables 40](#_Toc176091450)

[Constructors 41](#_Toc176091451)

[Class Inheritance 41](#_Toc176091452)

[Class Documentation 42](#_Toc176091453)

[Private Variables 42](#_Toc176091454)

[24th March 44](#_Toc176091455)

[Iterators in Classes 44](#_Toc176091456)

[Polymorphism 44](#_Toc176091457)

[Method Overriding 45](#_Toc176091458)

[Method Overloading 45](#_Toc176091459)

[Variable Length Argument Lists 45](#_Toc176091460)

[Default Parameter Values 46](#_Toc176091461)

[KWARGS 46](#_Toc176091462)

[The Pass Keyword 46](#_Toc176091463)

[Placeholder for Future Code 46](#_Toc176091464)

[Empty Code Blocks 47](#_Toc176091465)

[Infinite Loops 47](#_Toc176091466)

[Placeholders in Classes and Functions 47](#_Toc176091467)

[File Operation 47](#_Toc176091468)

[Opening a File 47](#_Toc176091469)

[Reading from a File 47](#_Toc176091470)

[End of File 48](#_Toc176091471)

[Writing to a File 48](#_Toc176091472)

[Appending to a File 48](#_Toc176091473)

[Closing a File 49](#_Toc176091474)

[Using "with" Statement (Context Manager): 49](#_Toc176091475)

[File Seek and Tell 49](#_Toc176091476)

[File Modes 49](#_Toc176091477)

[Exception Handling 51](#_Toc176091478)

[Raise 51](#_Toc176091479)

[Decorators 52](#_Toc176091480)

[30th March 55](#_Toc176091481)

[Static Method 55](#_Toc176091482)

[Custom Environment 55](#_Toc176091483)

[31st March 57](#_Toc176091484)

[NumPy 57](#_Toc176091485)

[Advanced Indexing 58](#_Toc176091486)

[Integer Array Indexing 58](#_Toc176091487)

[Boolean Array Indexing 58](#_Toc176091488)

[Complex Indexing Examples 59](#_Toc176091489)

[Slicing Operator 59](#_Toc176091490)

[Where Function 60](#_Toc176091491)

[Select Function 61](#_Toc176091492)

[arrange function 62](#_Toc176091493)

[Reshape Function 62](#_Toc176091494)

[Flattening Array 63](#_Toc176091495)

[3D- reshape 63](#_Toc176091496)

[Have to go recording from 12:30 PM 63](#_Toc176091497)

[6th April 64](#_Toc176091498)

[Pandas 64](#_Toc176091499)

[Category vs Object Data Types 65](#_Toc176091500)

[Series 65](#_Toc176091501)

[Creating a Series 65](#_Toc176091502)

[Custom Indexing 66](#_Toc176091503)

[Accessing Elements 66](#_Toc176091504)

[Operations on Series 66](#_Toc176091505)

[Series Attributes and Methods 67](#_Toc176091506)

[Data Frame 67](#_Toc176091507)

[Descriptive statistics 67](#_Toc176091508)

[For a DataFrame 68](#_Toc176091509)

[For a Series 68](#_Toc176091510)

[Exaplanation on the output 68](#_Toc176091511)

[Renaming the Columns 69](#_Toc176091512)

[Drop Null Values using dropna 70](#_Toc176091513)

[DataFrame: 70](#_Toc176091514)

[Series: 71](#_Toc176091515)

[Parameters: 71](#_Toc176091516)

[Drop Method 71](#_Toc176091517)

[Delete Columns 71](#_Toc176091518)

[Delete Rows 71](#_Toc176091519)

[Filling the Null Values using fillna method 72](#_Toc176091520)

[Change the column Type 72](#_Toc176091521)

[Indexing using iloc and loc 72](#_Toc176091522)

[Loc vs iloc 72](#_Toc176091523)

[Sort\_values method 73](#_Toc176091524)

[Boolean Indexing 73](#_Toc176091525)

[GroupBy 73](#_Toc176091526)

[Lambda Functions 74](#_Toc176091527)

[Example: 75](#_Toc176091528)

[Use Cases 75](#_Toc176091529)

[Limitations 75](#_Toc176091530)

[Using map() with Lambda Function 75](#_Toc176091531)

[Using filter() with Lambda Function 76](#_Toc176091532)

[Sorting with Lambda Function 76](#_Toc176091533)

[Read data from the excel 77](#_Toc176091534)

[Read data from the CSV 77](#_Toc176091535)

[07 April 78](#_Toc176091536)

[Have to go with recording till 10.40 AM 78](#_Toc176091537)

[CSV joins 78](#_Toc176091538)

[Joins 78](#_Toc176091539)

[Inner Join (how='inner') 79](#_Toc176091540)

[Outer Join (how='outer') 79](#_Toc176091541)

[Left Join (how='left') 79](#_Toc176091542)

[Right Join (how='right') 79](#_Toc176091543)

[Cross Join (Cartesian Join) 79](#_Toc176091544)

[MatPlotLib 79](#_Toc176091545)

[Description 79](#_Toc176091546)

[Line Plot Graph 81](#_Toc176091547)

[Bar Graph 82](#_Toc176091548)

[Histogram 82](#_Toc176091549)

[PieChart 82](#_Toc176091550)

[Scatter Plot 82](#_Toc176091551)

[13th April 83](#_Toc176091552)

[14th April 84](#_Toc176091553)

[20th April 84](#_Toc176091554)

[21st April 85](#_Toc176091555)

[4 May 86](#_Toc176091556)

[Statastics / stats 86](#_Toc176091557)

[Descriptive statistics 86](#_Toc176091558)

[Mean / Average 86](#_Toc176091559)

[Meadian 87](#_Toc176091560)

[Mode 87](#_Toc176091561)

[Range 88](#_Toc176091562)

[Variance 88](#_Toc176091563)

[Standard Deviation 90](#_Toc176091564)

[Percentile 91](#_Toc176091565)

[Quartile / quantile 93](#_Toc176091566)

[5th May 94](#_Toc176091567)

[Outliers 94](#_Toc176091568)

[Correlation 96](#_Toc176091569)

[Inferential statistics 96](#_Toc176091570)

[11th May 97](#_Toc176091571)

[12th May 97](#_Toc176091572)

[18th May 97](#_Toc176091573)

[19th May 97](#_Toc176091574)

[25th May 98](#_Toc176091575)

[CHI square test 98](#_Toc176091576)

[Introduction to ML 98](#_Toc176091577)

[Types of ML algorithms 98](#_Toc176091578)

[Supervised Learning 98](#_Toc176091579)

[Unsupervised Learning 99](#_Toc176091580)

[Reinforcement Learning 101](#_Toc176091581)

[Key Concepts in Reinforcement Learning 101](#_Toc176091582)

[Types of Reinforcement Learning 101](#_Toc176091583)

[Key Algorithms in Reinforcement Learning 102](#_Toc176091584)

[Workflow of a Reinforcement Learning Task 102](#_Toc176091585)

[Examples of Applications of Reinforcement Learning 102](#_Toc176091586)

[26th May 103](#_Toc176091587)

[Linear Regression 103](#_Toc176091588)

[Linear Regression in Machine Learning 103](#_Toc176091589)

[Types of Linear Regression 103](#_Toc176091590)

[Key Components 103](#_Toc176091591)

[Objective 103](#_Toc176091592)

[Mathematical Representation 103](#_Toc176091593)

[Assumptions 104](#_Toc176091594)

[Sum of Square Errors (SSEs) 104](#_Toc176091595)

[Mean Square Error (MSE) 105](#_Toc176091596)

[Why Use MSE? 105](#_Toc176091597)

[Interpretation of MSE 105](#_Toc176091598)

[1st June 106](#_Toc176091599)

[2nd June 106](#_Toc176091600)

[8th June 106](#_Toc176091601)

[Linear Regression 106](#_Toc176091602)

[**Homoscedasticity** and **Heteroscedasticity** 109](#_Toc176091603)

[Homoscedasticity 110](#_Toc176091604)

[Heteroscedasticity 110](#_Toc176091605)

[Example in Simple Linear Regression 111](#_Toc176091606)

[Multicollinearity in Regression Analysis 111](#_Toc176091607)

[Effects of Multicollinearity 112](#_Toc176091608)

[Diagnosing Multicollinearity 112](#_Toc176091609)

[9th June 114](#_Toc176091610)

[Naive Model 114](#_Toc176091611)

[Adjusted R-squared 114](#_Toc176091612)

[Logistic Regression 115](#_Toc176091613)

[Measure of Spread 117](#_Toc176091614)

[15th June 119](#_Toc176091615)

[Startify attribute 119](#_Toc176091616)

[Importance of Stratification 119](#_Toc176091617)

[How Stratification Works 119](#_Toc176091618)

[Usage Example in Python with train\_test\_split 119](#_Toc176091619)

[Output Explanation 120](#_Toc176091620)

[When to Use Stratification 120](#_Toc176091621)

[Summary 120](#_Toc176091622)

[random\_state attribute 120](#_Toc176091623)

[Why Use random\_state? 120](#_Toc176091624)

[How random\_state Works 120](#_Toc176091625)

[fit\_transform vs transform 121](#_Toc176091626)

[fit\_transform 121](#_Toc176091627)

[When to Use fit\_transform 121](#_Toc176091628)

[transform 121](#_Toc176091629)

[When to Use transform 121](#_Toc176091630)

[Summary 121](#_Toc176091631)

[Decision Trees 122](#_Toc176091632)

[1. Concept and Structure 122](#_Toc176091633)

[2. How Decision Trees Work 122](#_Toc176091634)

[3. Advantages and Disadvantages 122](#_Toc176091635)

[4. Pruning 123](#_Toc176091636)

[5. Important Parameters 123](#_Toc176091637)

[6. Ensemble Methods 123](#_Toc176091638)

[Occam's Razor Principle 123](#_Toc176091639)

[Origin and Explanation 123](#_Toc176091640)

[Application in Science and Machine Learning 124](#_Toc176091641)

[Examples 124](#_Toc176091642)

[Mathematical Formulation 124](#_Toc176091643)

[Practical Considerations 124](#_Toc176091644)

[Summary 124](#_Toc176091645)

[16th June 125](#_Toc176091646)

[Overfitting vs Underfitting 125](#_Toc176091647)

[Overfitting 125](#_Toc176091648)

[Underfitting 125](#_Toc176091649)

[Balancing Overfitting and Underfitting 126](#_Toc176091650)

[Entropy 126](#_Toc176091651)

[Conclusion 128](#_Toc176091652)

[Random Forest 128](#_Toc176091653)

[Key Concepts 128](#_Toc176091654)

[How Random Forest Works 129](#_Toc176091655)

[Advantages 129](#_Toc176091656)

[Disadvantages 129](#_Toc176091657)

[Hyperparameters 129](#_Toc176091658)

[Example in Python (scikit-learn) 130](#_Toc176091659)

[Conclusion 130](#_Toc176091660)

[22nd June 133](#_Toc176091661)

[Bias vs variance Error 133](#_Toc176091662)

[Bias Error 133](#_Toc176091663)

[Variance Error 134](#_Toc176091664)

[Techniques to Address Bias and Variance 134](#_Toc176091665)

[Synthetic Data 134](#_Toc176091666)

[1. What is Synthetic Data? 134](#_Toc176091667)

[2. Uses of Synthetic Data in ML 134](#_Toc176091668)

[3. Generating Synthetic Data 135](#_Toc176091669)

[4. Advantages of Synthetic Data 135](#_Toc176091670)

[5. Challenges and Considerations 135](#_Toc176091671)

[Summary 135](#_Toc176091672)

[Oversampling and Undersampling 135](#_Toc176091673)

[Oversampling 136](#_Toc176091674)

[Methods of Oversampling 136](#_Toc176091675)

[Advantages of Oversampling 136](#_Toc176091676)

[Disadvantages of Oversampling 136](#_Toc176091677)

[Undersampling 136](#_Toc176091678)

[Methods of Undersampling 136](#_Toc176091679)

[Advantages of Undersampling 137](#_Toc176091680)

[Disadvantages of Undersampling 137](#_Toc176091681)

[Choosing Between Oversampling and Undersampling 137](#_Toc176091682)

[Practical Tips 137](#_Toc176091683)

[Confusion Matrix 138](#_Toc176091684)

[Structure of a Confusion Matrix 138](#_Toc176091685)

[Key Terms 138](#_Toc176091686)

[Metrics Derived from the Confusion Matrix 138](#_Toc176091687)

[**Accuracy** 138](#_Toc176091688)

[**Precision** (Positive Predictive Value) 138](#_Toc176091689)

[**Recall** (Sensitivity or True Positive Rate) 139](#_Toc176091690)

[F1 Score 139](#_Toc176091691)

[**Specificity** (True Negative Rate) 139](#_Toc176091692)

[False Positive Rate (FPR) 139](#_Toc176091693)

[False Negative Rate (FNR) 139](#_Toc176091694)

[Example 139](#_Toc176091695)

[Importance of the Confusion Matrix 140](#_Toc176091696)

[Accuracy Score vs ROC\_AUC\_Score 140](#_Toc176091697)

[Accuracy Score 140](#_Toc176091698)

[Advantages 140](#_Toc176091699)

[Disadvantages 140](#_Toc176091700)

[ROC AUC Score 140](#_Toc176091701)

[Components 141](#_Toc176091702)

[Advantages 141](#_Toc176091703)

[Disadvantages 141](#_Toc176091704)

[When to Use Each Metric 141](#_Toc176091705)

[Summary 141](#_Toc176091706)

[Naive Bayes algorithm 142](#_Toc176091707)

[23rd June 143](#_Toc176091708)

[CountVectorizer 143](#_Toc176091709)

[K-Means algorithm 143](#_Toc176091710)

[How K-Means Algorithm Works 143](#_Toc176091711)

[Detailed Steps 144](#_Toc176091712)

[Choosing the Number of Clusters (k) 144](#_Toc176091713)

[**Elbow Method** 144](#_Toc176091714)

[**Silhouette Score** 145](#_Toc176091715)

[Advantages and Disadvantages 146](#_Toc176091716)

[Advantages: 146](#_Toc176091717)

[Disadvantages: 146](#_Toc176091718)

[Conclusion 146](#_Toc176091719)

[K-Means++ 146](#_Toc176091720)

[How K-Means++ Works 146](#_Toc176091721)

[Benefits of K-Means++ 147](#_Toc176091722)

[Conclusion 147](#_Toc176091723)

[29th June 148](#_Toc176091724)

[K-Means Algorithm Applications 148](#_Toc176091725)

[K-Means Clustering Drawbacks 148](#_Toc176091726)

[1. Choosing the Number of Clusters (K) 148](#_Toc176091727)

[2. Sensitivity to Initialization 148](#_Toc176091728)

[3. Assumption of Spherical Clusters 149](#_Toc176091729)

[4. Sensitivity to Outliers 149](#_Toc176091730)

[5. Scalability 149](#_Toc176091731)

[6. Cluster Assumption 149](#_Toc176091732)

[7. Interpretability 149](#_Toc176091733)

[8. Data Preprocessing Requirements 149](#_Toc176091734)

[9. Cluster Size Imbalance 149](#_Toc176091735)

[10. Non-deterministic Results 150](#_Toc176091736)

[11. Computational Complexity 150](#_Toc176091737)

[Hierarchical Clustering 150](#_Toc176091738)

[Agglomerative Hierarchical Clustering 150](#_Toc176091739)

[Steps: 150](#_Toc176091740)

[Divisive Hierarchical Clustering 150](#_Toc176091741)

[Steps: 150](#_Toc176091742)

[Proximity Matrix 151](#_Toc176091743)

[Construction of a Proximity Matrix 151](#_Toc176091744)

[Dendrogram 152](#_Toc176091745)

[Advantages 152](#_Toc176091746)

[Disadvantages 152](#_Toc176091747)

[Applications 152](#_Toc176091748)

[30th June 153](#_Toc176091749)

[Linkage 153](#_Toc176091750)

[Metric 153](#_Toc176091751)

[Association Rule Mining 153](#_Toc176091752)

[Key Concepts 153](#_Toc176091753)

[Example 154](#_Toc176091754)

[Process of Association Rule Mining 154](#_Toc176091755)

[Algorithms 155](#_Toc176091756)

[6th July 156](#_Toc176091757)

[Dimensionality Reduction 156](#_Toc176091758)

[Principal Component Analysis 156](#_Toc176091759)

[Key Concepts 156](#_Toc176091760)

[Steps in PCA 156](#_Toc176091761)

[Applications of PCA 157](#_Toc176091762)

[Advantages and Disadvantages 157](#_Toc176091763)

[Projection of a Data Point 157](#_Toc176091764)

[Steps to Project a Data Point 158](#_Toc176091765)

[7th July 159](#_Toc176091766)

[Linear Discriminant Analysis 159](#_Toc176091767)

[Overview of LDA 161](#_Toc176091768)

[How LDA Works 161](#_Toc176091769)

[PCA vs LDA 161](#_Toc176091770)

[13th July 162](#_Toc176091771)

[Covariance 162](#_Toc176091772)

[Eigenvalues and Eigenvectors in PCA 162](#_Toc176091773)

[Step-by-Step Explanation 162](#_Toc176091774)

[Example 163](#_Toc176091775)

[Why This Matters 164](#_Toc176091776)

[14th July 165](#_Toc176091777)

[Variance 165](#_Toc176091778)

[Classification Report 166](#_Toc176091779)

[Generating a Classification Report 166](#_Toc176091780)

[Example Output 166](#_Toc176091781)

[Interpretation 166](#_Toc176091782)

[Full Code Example 167](#_Toc176091783)

[Confusion Matrix 168](#_Toc176091784)

[Structure of a Confusion Matrix 168](#_Toc176091785)

[Precision 168](#_Toc176091786)

[Formula 168](#_Toc176091787)

[Interpretation 169](#_Toc176091788)

[Summary 169](#_Toc176091789)

[Recall 169](#_Toc176091790)

[Formula 169](#_Toc176091791)

[Interpretation 169](#_Toc176091792)

[Summary 169](#_Toc176091793)

[Accuracy 170](#_Toc176091794)

[Formula 170](#_Toc176091795)

[Interpretation 170](#_Toc176091796)

[Summary 170](#_Toc176091797)

[20th July 170](#_Toc176091798)

[21st July 170](#_Toc176091799)

[27th July 171](#_Toc176091800)

[Time Series Data 171](#_Toc176091801)

[Data Patterns 172](#_Toc176091802)

[Rolling Mean 172](#_Toc176091803)

[Types of Rolling Mean 172](#_Toc176091804)

[Purpose and Applications 173](#_Toc176091805)

[Rolling Standard Deviation 173](#_Toc176091806)

[Purpose and Applications 173](#_Toc176091807)

[Calculation 173](#_Toc176091808)

[ARIMA model 174](#_Toc176091809)

[Components of ARIMA 175](#_Toc176091810)

[ARIMA Model Notation 175](#_Toc176091811)

[Steps to Build an ARIMA Model 175](#_Toc176091812)

[Example 175](#_Toc176091813)

[Benefits of ARIMA 176](#_Toc176091814)

[Adfuller Test 176](#_Toc176091815)

[Key Concepts 176](#_Toc176091816)

[Test Statistics 176](#_Toc176091817)

[Steps to Perform the ADF Test 176](#_Toc176091818)

[Example 177](#_Toc176091819)

[Summary 177](#_Toc176091820)

[28th July 177](#_Toc176091821)

[3rd August 177](#_Toc176091822)

[ACF(q) vs PACF(p) 177](#_Toc176091823)

[1. Autocorrelation Function (ACF) 177](#_Toc176091824)

[2. Partial Autocorrelation Function (PACF) 178](#_Toc176091825)

[Usage in Machine Learning 178](#_Toc176091826)

[4th August 178](#_Toc176091827)

[**Decomposition** 178](#_Toc176091828)

[Key Components of Time Series Decomposition 178](#_Toc176091829)

[Types of Decomposition 179](#_Toc176091830)

[Use Cases of Decomposition 179](#_Toc176091831)

[10th Aug 180](#_Toc176091832)

[11th Aug 180](#_Toc176091833)

[17th Aug 180](#_Toc176091834)

[18th Aug 180](#_Toc176091835)

[nlargest method 180](#_Toc176091836)

[24th August 181](#_Toc176091837)

[31st August 181](#_Toc176091838)

[1st Sep 181](#_Toc176091839)

# References

GitHub Link

* [shikharkumar13 (Data Science With Shikhar) (github.com)](https://github.com/shikharkumar13)

Drive:

* [Data Science Files - Google Drive](https://drive.google.com/drive/folders/1jiKxJCI3Cnd_pgznxQ-690O8RyfmgiPV)

# 09th March 2024

This is beginning of the Python course.

## Installation

* The anaconda installation shown here.
* Shown the Jupiter Notebook usage.
* Colab.google.com is alternative for Jupiter Notebook.

## Variables

Variables can’t have

1. They can only contain letters (a-z, A-Z), digits (0-9), and underscores ('\_').
2. They cannot start with a digit.
3. They cannot be a keyword or reserved word in Python.
4. Can’t use ‘-‘ (Hyphen) or ‘@’

Note:

* The '@' symbol is not allowed because it's reserved for decorators in Python. Decorators are used to modify the behaviour of functions or methods.
* The '-' symbol is not allowed because it's an operator in Python used for subtraction.

### Type method

Type is a method which explains the variable type

* Print(type(variable\_name))

A screenshot of a phone

Description automatically generated

### del

We can use del to delete variables from the current namespace.

Ex:

x = 5

del x # Delete the variable x

### Unpacking or multiple declarations

We can declare and assign values to multiple variables in a single line using multiple assignment, also known as unpacking. This is commonly done using tuples, lists, or even strings.

Here are a few examples:

Ex1:

x, y, z = 1, 2, 3

Ex2:

values = [1, 2, 3]

x, y, z = values

Ex3:

x, y, z = "abc"

This assigns the characters 'a', 'b', and 'c' to variables x, y, and z respectively.

Ex4:

def get\_values():

return 1, 2, 3

x, y, z = get\_values()

### F-Strings

F-strings, also known as formatted string literals, are a feature introduced in Python 3.6 that allow for easier string formatting. They provide a concise and readable way to embed expressions inside string literals.

Note:

* Its like printf in CPP

To create an f-string, prefix the string with 'f' or 'F' and then place the expressions you want to embed inside curly braces {}. When the string is evaluated, the expressions inside the curly braces are replaced with their values.

Here's a simple example:

name = "Jagadeesh"

age = 40

f\_string = f"My name is {name} and I am {age} years old."

print(f\_string)

The Output will be:

My name is Jagadeesh and I am 40 years old.

Or we can use directly print like below which also give same output:

print(f"My name is {name} and I am {age} years old.")

## Operators

* \*\* is used for power values
  + 5\*\*2 = 25
* // is for floor division
  + 63//5 = 12
  + 63/5 = 12.6

## Input

The input() function is used to accept user input from the keyboard. It allows the program to pause and wait for the user to enter some text, which is then returned as a string.

Ex:

name = input("Enter your name: ")

print("Hello,", name)

It's important to note that the value returned by input() is always a string. If you want to convert it to another data type (e.g., integer, float), you'll need to use appropriate type conversion functions like int() or float().

Ex:

age = int(input("Enter your age: "))

print(F"You are {age} years old.")

### Format input

names = ["Sachin", "Rohit", "Sourav"]

attendance = []

for name in names:

status = input(f"is {name} present or absent ")

attendance.append(status)

print(attendance)

The output will be:

Sachin is present or absent P

Rohit is present or absent A

Sourav is present or absent P

['P', 'A', 'P']

# 10th March

## Python Data Structure

* Strings
* Lists
* Tuples
* Dictionaries
* Sets

Strings  
Strings are sequences of characters enclosed within single, double, or triple quotes. Although not explicitly a data structure, strings can be manipulated in various ways and treated as sequences in Python.

Ex:

my\_string = "Hello, World!"

#### triple quotes

Triple quotes (''' or """) in Python are used to define multi-line strings or to include special characters such as newline characters (\n) without using escape sequences. Here are some situations where you might want to use triple quotes:

* **Multi-line Strings:**

Triple quotes are commonly used to define multi-line strings, especially when the string spans across multiple lines in the code.

message = '''This is a multi-line

string. It spans across

multiple lines.'''

* **Docstrings:**

Triple quotes are used to write docstrings, which are documentation strings enclosed in triple quotes and used to document functions, classes, modules, or packages.

def my\_function():

"""This is a docstring.

It provides information about the function."""

pass

**Including Special Characters:**

Triple quotes allow you to include special characters such as newline characters (\n) without using escape sequences.

poem = """Roses are red,

Violets are blue,

Sugar is sweet,

And so are you."""

 **Embedding Quotes:**

Triple quotes can be useful when you need to include single or double quotes within the string without escaping them.

message = """He said, "Hello!" """

#### methods

Following are the various methods which are using in Strings

1. capitalize()
   1. This method returns a copy of the string with the first character capitalized and the rest of the characters converted to lowercase.
   2. Ex:

string = "hello world"

capitalized\_string = string.capitalize()

print(capitalized\_string) # Output: Hello world

1. lower()
2. upper()
3. title()
   1. The title() method in Python is used to convert the first character of each word in a string to uppercase and convert all other characters to lowercase.
   2. Ex:

string = "hello world"

title\_case\_string = string.title()

print(title\_case\_string) # Output: Hello World

1. find()
2. index()
3. replace()
4. split()
5. join()
   1. This method joins the elements of an iterable (such as a list) into a single string using the string as a separator.
   2. Ex:

words = ['hello', 'world']

string = " ".join(words)

print(string) # Output: hello world, it joined two string

using space a separator

1. strip()
   1. This method returns a copy of the string with leading and trailing whitespace removed.
   2. Ex:

string = " hello world "

stripped\_string = string.strip()

print(stripped\_string) # Output: hello world

1. startswith()
2. endswith()
3. isalpha()
4. isdigit()
5. len()
6. count()
7. format()
   1. The format() method in Python is used to format strings by replacing placeholders {} with the values provided as arguments. It allows you to create dynamic strings where certain parts are replaced with variables or other dynamic content.
   2. Ex:

n = "Alice"

a = 30

string = "Name is {}, {} years old.".format(n, a)

print(formatted\_string) # Output: Name is Alice, 30 years old.

#### Convert List to string

We have two methods here:

* Using iterative method
* Using join method

##### Iterative method

my\_list = ['a', 'b', 'c']

result\_string = ''

for char in my\_list:

result\_string += char

print(result\_string) # Output: abc

##### Join method

my\_list = ['a', 'b', 'c']

result\_string = ''.join(my\_list)

print(result\_string) # Output: abc

#### Convert String to List

my\_string = "hello"

my\_list = list(my\_string)

print(my\_list) # Output: ['h', 'e', 'l', 'l', 'o']

### Lists

Lists are ordered collections of items, which can be of different data types. They are mutable, meaning you can modify the elements after creation.

Ex:

my\_list = [1, “ABC”, 30.5, 4, 5]

#### len

Ex:

Len(my\_list)

#### append

the append() method is used to add an element to the end of a list. It modifies the original list in place.

Ex:

my\_list = [1, 2, 3]

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

#### remove

the remove() method is used to remove the first occurrence of a specified value from a list. It modifies the original list in place. If the specified value is not found in the list, it raises a ValueError

Ex:

my\_list = [1, 2, 3, 4, 5, 3]

my\_list.remove(3)

print(my\_list) # Output: [1, 2, 4, 5, 3]

#### Indexing

Ex:

my\_list[0]

#### index

the index() method is used to find the index of the first occurrence of a specified value within a list. It returns the index of the first occurrence of the value if it exists in the list, and raises a ValueError if the value is not found.

Ex:

my\_list = [10, 20, 30, 40, 50]

index = my\_list.index(30)

print(index) # Output: 2

#### extend

the extend() method is used to add elements from an iterable (such as a list, tuple, or set) to the end of a list. This method modifies the original list in place and does not return a new list.

Ex:

my\_list = [1, 2, 3]

additional\_elements = [4, 5, 6]

my\_list.extend(additional\_elements)

print(my\_list) # Output: [1, 2, 3, 4, 5, 6]

We can also use extend() with other iterables, such as tuples or sets

Ex1:

my\_list = [1, 2, 3]

additional\_elements = (4, 5, 6) # Tuple

my\_list.extend(additional\_elements)

print(my\_list) # Output: [1, 2, 3, 4, 5, 6]

Ex2:

my\_list = [1, 2, 3]

additional\_elements = {4, 5, 6} # Set

my\_list.extend(additional\_elements)

print(my\_list) # Output: [1, 2, 3, 4, 5, 6]

#### pop

the pop() method is used to remove and return the element at a specified index from a list. It modifies the original list in place. If no index is specified, pop() removes and returns the last element of the list.

Ex1:

my\_list = [1, 2, 3, 4, 5]

element = my\_list.pop(2)

print(element) # Output: 3

print(my\_list) # Output: [1, 2, 4, 5]

Ex2:

my\_list = [1, 2, 3, 4, 5]

element = my\_list.pop()

print(element) # Output: 5

#### count

The count() method is used to count the number of occurrences of a specified element in a list. It returns the number of times the specified element appears in the list.

Ex:

my\_list = [1, 2, 3, 4, 1, 2, 1, 3, 1]

count\_of\_ones = my\_list.count(1)

print(count\_of\_ones) # Output: 4

#### sort

the sort() method is used to sort the elements of a list in place. It modifies the original list and does not return a new list. The sort() method arranges the elements of the list in ascending order by default.

Ex:

my\_list = [3, 1, 4, 1, 5, 9, 2, 6]

my\_list.sort()

print(my\_list) # Output: [1, 1, 2, 3, 4, 5, 6, 9]

We can also use the **reverse** parameter to sort the list in descending order

my\_list.sort(reverse=True)

print(my\_list) # Output: [9, 6, 5, 4, 3, 2, 1, 1]

#### Mutability

In programming, when an object is mutable, it means that its state or contents can be changed, added to, or removed from after it has been created.

Note:

* lists, dictionaries, and sets in Python are **mutable**.
  + my\_list = [1, 2, 3]
  + my\_list[0] = 10 # Modifying element
  + my\_list.append(4) # Adding element
* tuples, strings, and integers in Python are **immutable**.
  + my\_tuple = (1, 2, 3)
  + my\_tuple[0] = 10 # Raises TypeError: 'tuple' object does not support item assignment

### Tuple

A tuple is an ordered, **immutable** collection of elements.

* Tuples are similar to lists, but with the key difference that tuples cannot be modified after creation.
* They are defined using parentheses ()
* Can contain elements of different data types, including other tuples.

Ex:

my\_tuple = (1, 2, 3, "hello", True)

Tuples can also be created without parentheses, but it's a good practice to include them for clarity:

my\_tuple = 1, 2, 3, "hello", True

We can access elements of a tuple using indexing, just like as lists:

print(my\_tuple[0]) # Output: 1

print(my\_tuple[3]) # Output: "hello"

### **Dictionaries**

Dictionaries are collections of key-value pairs, where each key is associated with a value. They are unordered and mutable.

Note:

* It’s kind of hash map.

Ex:

my\_dict = {"name": "John", "age": 30, "city": "New York"}

print(my\_dict) # {'name': 'John', 'age': 30, 'city': 'New York'}

#### Print Specific key value

Ex:

Print(my\_dict(“name”)) #Output: “John”

Note

* Using indexes we can’t print dictionaries.

#### Print Key and Values

for key, value in my\_dict.items():

print(key, ":", value)

The Above statement gives the out as

name : John

age : 30

city : New York

#### Keys

We can also print keys

# Print keys only

print("Keys:", my\_dict.keys())

The output will be

Keys: dict\_keys(['name', 'age', 'city'])

#### Values

We can print all values

print("Values:", my\_dict.values())

The output will be

Values: dict\_values(['John', 30, 'New York'])

### **Sets**

* Sets are unordered collections of unique elements.
* They are useful for eliminating duplicate values from a sequence and performing mathematical set operations like
  + union,
    - a.union(b)
  + intersection, etc.
    - a.intersection(b)
* In sets, add is the method which will be used to insert any new element to set.
  + Append is the method for the same operation for other data types.
* Clear is the method to clear all the data
* Del is the method to delete the set
* We can’t use pop method using the index.

Ex:

my\_set = {1, 2, 3, 4, 5}

#### Remove vs Discard

Remove: Find the value and delete. If the value does’t exist throughs an error.

### Tuple vs List vs Set vs Dictionary

| **Feature** | **Tuple** | **List** | **Set** | **Dictionary** |
| --- | --- | --- | --- | --- |
| **Mutability** | Immutable | Mutable | Mutable | Mutable |
| **Ordered** | Yes | Yes | No | Python 3.7+: Insertion order is maintained; before 3.7, No |
| **Allows Duplicates** | Yes | Yes | No | Keys: No; Values: Yes |
| **Syntax** | Defined using () | Defined using [] | Defined using {} | Defined using {key: value, ...} |
| **Example** | (1, 2, 3) | [1, 2, 3] | {1, 2, 3} | {'a': 1, 'b': 2, 'c': 3} |

## Iterators

loops, or iterators, are used to execute a block of code repeatedly until a certain condition is met. Python supports two main types of loops:

* for
* while

### for Loop

The for loops are used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence.

#### **Iterating over Sequences**

for item in sequence:

# Do something with item

Ex:

# Iterating over a list

fruits = ["apple", "banana", "orange"]

for fruit in fruits:

print(fruit)

#### **Known Number of Iterations**

Use for loops when you know the number of iterations in advance, such as iterating over a fixed-size sequence.

Ex:

# Iterate over a range of numbers

for i in range(5):

print(i)

#### **Enumeration**

for loops are often used with the enumerate() function to iterate over the elements of a sequence along with their indices.

for index, item in enumerate(sequence):

# Do something with index and item

Ex:

# Enumerate over a list

fruits = ["apple", "banana", "orange"]

for index, fruit in enumerate(fruits):

print(f"Index: {index}, Fruit: {fruit}")

The output for above code snippet is

Index: 0, Fruit: apple

Index: 1, Fruit: banana

Index: 2, Fruit: orange

#### **Iteration with Iterators**

for loops can be used to iterate over any iterable object, including generators and custom iterator objects.

Ex:

# Iterating over a string iterator

string = "hello"

for char in string:

print(char)

Short Hand Notation of For

you can use list comprehensions and generator expressions as shorthand notations for simple for loops. They provide a concise and readable way to create lists or iterators based on existing iterables.

Syntax

[expression for item in iterable]

Ex:

# Generator expression

squares\_iterator = (x \*\* 2 for x in range(5))

# Accessing elements of the generator

for square in squares\_iterator:

print(square)

Note:

* Short hand notations of for loop is used to create new lists or iterators by applying an expression to each element of an existing iterable. They are not designed to execute statements like print().

### while Loop

The while loops are used to repeatedly execute a block of code as long as a specified condition is true.

Ex:

# Print numbers from 0 to 4 using a while loop

i = 0

while i < 5:

print(i)

i += 1

#### **Unknown Number of Iterations**

while loops are used when you don't know the number of iterations in advance and want to repeat a block of code until a certain condition is met.

while condition:

# Do something

Ex:

# Iterating until a certain condition is met

count = 0

while count < 5:

print(count)

count += 1

#### **Condition-based Iteration**

Use while loops when you want to iterate until a specific condition becomes False, such as processing user input until a specific command is entered.

Ex:

# Iterating until a certain condition is met based on user input

user\_input = ''

while user\_input != 'exit':

user\_input = input("Enter a command (type 'exit' to quit): ")

print("You entered:", user\_input)

#### **Looping with State**

while loops are often used when you need to maintain state within the loop and update it based on certain conditions.

Ex:

# Looping with state to calculate the factorial of a number

number = 5

factorial = 1

while number > 0:

factorial \*= number

number -= 1

print("Factorial:", factorial)

Looping with Dynamic Conditions

while loops are suitable for tasks where the termination condition depends on dynamic factors that may change during each iteration.

Ex:

# Looping with dynamic conditions

num = 1

while num < 100:

print(num)

num \*= 2

### For vs While

Both for and while loops are used for iteration in Python, but they are suited for different types of tasks and scenarios. Here's a comparison of their typical applications:

#### Common Applications

* **for Loop**: Used for tasks with a fixed number of iterations or when iterating over sequences.
* **while Loop**: Used for tasks with an unknown or dynamic number of iterations or when repeating a block of code until a specific condition is met.

In summary, choose between for and while loops based on the nature of the task, whether you know the number of iterations in advance, and whether the termination condition depends on dynamic factors. Both loops have their distinct use cases and are essential tools for writing effective Python code.

### Control Flow Statements

We can also use control flow statements such as break, continue, and else within loops to control their behaviour.

#### break

Terminates the loop prematurely.

#### continue

Skips the rest of the current iteration and proceeds to the next iteration.

#### else

Executes a block of code when the loop completes normally (i.e., without encountering a break statement).

Ex:

# Print odd numbers from 1 to 9, skipping 5

for i in range(1, 10):

if i == 5:

continue

if i % 2 == 0:

print("Even:", i)

else:

print("Odd:", i)

else:

print("Loop completed normally")

# 16th March, 2024

In first part of the session, we have discussed on:

* Sets
* String

These two topics explained above.

## Slicing

slices are a way to extract a portion of a sequence (such as a string, list, or tuple). Slicing allows you to create a new sequence containing a subset of the elements from the original sequence.

The syntax for slicing is [start:stop:step], where:

* start: The index representing the start of the slice (inclusive).
* stop: The index representing the end of the slice (exclusive).
* step: Optional. The step size indicating the increment between elements to include in the slice. The default value is 1.

Here are some examples demonstrating the usage of slicing:

### Slicing a string

my\_string = "hello world"

print(my\_string[0:5]) # Output: hello

print(my\_string[6:]) # Output: world

print(my\_string[::-1]) # Output: dlrow olleh (reverse the string)

### Slicing a list

my\_list = [1, 2, 3, 4, 5]

print(my\_list[1:4]) # Output: [2, 3, 4]

print(my\_list[:3]) # Output: [1, 2, 3]

**print(my\_list[::2]) # Output: [1, 3, 5] (every other element)**

#### Slicing a tuple

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple[2:]) # Output: (3, 4, 5)

## Escape Sequences

Escape sequences in Python are special characters that are preceded by a backslash (\) within a string. They are used to represent characters that are difficult or impossible to type directly into the code, such as newline characters or quotation marks.

Here are some common escape sequences in Python:

1. \n: Newline
   * Inserts a newline character.
2. \t: Tab
   * Inserts a horizontal tab.
3. \\: Backslash
   * Inserts a literal backslash.
4. \': Single Quote
   * Inserts a single quote character.
5. \": Double Quote
   * Inserts a double quote character.
6. \b: Backspace
   * Inserts a backspace character.
7. \r: Carriage Return
   * Moves the cursor to the beginning of the line.
8. \f: Formfeed
   * Inserts a formfeed character.
9. \v: Vertical Tab
   * Inserts a vertical tab character.
10. \N{name}: Unicode Character by Name
    * Inserts the Unicode character specified by name.
11. \xhh: Hexadecimal Value
    * Inserts the character with the hexadecimal value hh.
12. \uhhhh: Unicode Character by Hexadecimal Value
    * Inserts the Unicode character with the hexadecimal value hhhh.
13. \ooo: Octal Value
    * Inserts the character with the octal value ooo.

These escape sequences can be used within string literals to represent special characters or characters that are difficult to type directly.

Ex:

print("Hello\nWorld") # Output:

# Hello

# World

print("This is a\ttab") # Output: This is a tab

print("A backslash: \\") # Output: A backslash: \

print("He said, \"Hello!\"") # Output: He said, "Hello!"

print("Unicode character: \u03A9") # Output: Unicode character: Ω

## Conditional statements

### If, elif else

The main conditional statements in Python are if, elif (short for "else if"), and else.

Syntax:

if condition:

# Block of code to execute if the condition is True

elif another\_condition:

# Block of code

else:

# Block of code to execute if all previous conditions are False

### Ternary Operator

The ternary operator is a concise way to write an if-else statement in a single line. It's often referred to as the conditional expression.

Syntax

<if-statement> **if** <condition> **else** <else-statement>

Ex:

x = 10

y = 20

result = x if x > y else y

print(result)

# 17th March

In this class we have explanation for

* Shorthand notation of for loop
* While loop
* Control statement

These all topics have above in this document

## Functions

A function is a block of reusable code that performs a specific task. Functions are used to organize code into manageable pieces, improve code readability, and promote code reuse. They allow you to encapsulate a sequence of statements into a single entity that can be called from elsewhere in your program.

Here are some key features of functions in Python:

**Syntax**:

* Functions in Python are defined using the def keyword, followed by the function name, parentheses (), and a colon :. The body of the function is indented.

Ex:

def my\_function():

# Function body

### Arguments

* Functions can accept zero or more parameters, also known as arguments, which are specified within the parentheses after the function name. These parameters can be used inside the function to perform operations.

Ex:

def greet(name):

print("Hello,", name)

### Return Values

* Functions can optionally return a value using the return statement. This allows the function to produce output that can be used by the caller.

Ex:

def add(a, b):

return a + b

### Function Call

* To call a function, you simply write its name followed by parentheses, optionally passing arguments inside the parentheses.

Ex:

result = add(3, 5)

### Scope

* Functions introduce their own scope, meaning variables defined inside a function are local to that function by default. They cannot be accessed from outside the function unless explicitly returned.

Functions are essential building blocks in Python programming, allowing you to modularize your code and make it more structured and reusable.

Note:

* Functions are called as encapsulation in OOPs concept

## OOPs Concept

OOPS, or Object-Oriented Programming (OOP), is a programming paradigm based on the concept of "objects," which can contain data (attributes) and code (methods). OOP allows you to model real-world entities as software objects, making it easier to understand and manage complex systems.

The key principles of OOP include:

* Encapsulation
* Abstraction
* Inheritance
* Polymorphism

OOP promotes modular design, code reusability, and easier maintenance by organizing code into smaller, self-contained units (objects) that interact with each other. It encourages a more natural and intuitive way of thinking about software development by modeling the problem domain using concepts and entities from the real world. OOP is widely used in modern programming languages such as Python, Java, C++, and C#.

### Encapsulation

Encapsulation involves bundling data (attributes) and methods (functions) into a single unit, called a class. The internal state of an object is hidden from the outside world, and access to it is restricted to methods defined within the class.

Ex:

class Dog:

def \_\_init\_\_(self, name):

self.name = name

def bark(self):

return "Woof!"

# Creating an instance of the Dog class

my\_dog = Dog("Buddy")

# Accessing the name attribute directly

print(my\_dog.name) # Output: Buddy

# Accessing the bark method

print(my\_dog.bark()) # Output: Woof!

In this example, the name attribute is encapsulated within the Dog class, and it can only be accessed or modified through methods like bark().

### Abstraction

Abstraction involves extracting common properties and behaviors from a set of objects to create a generalized model. Abstract classes and interfaces provide blueprints for defining common behaviours.

Ex:

class Car:

def drive(self):

pass

class Maruti(Car):

def drive(self):

return "Automated driving"

class Ford(Car):

def drive(self):

return "Manual driving"

# Creating instances of subclasses

my\_maruti = Maruti()

my\_ford = Ford()

In this example, the Car class is abstract, and its subclasses Maruti and Ford provide specific implementations of the drive() method.

### Inheritance

Inheritance allows a class (subclass) to inherit properties and behaviors from another class (superclass), promoting code reuse and facilitating hierarchical relationships.

Ex:

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Woof!"

# Creating an instance of the Dog class

my\_dog = Dog()

# Using the speak method inherited from the Animal class

print(my\_dog.speak()) # Output: Woof!

In this example, the Dog class inherits the speak() method from the Animal class.

## Global Variables

* Global variables are defined outside any function and can be accessed from any part of the code, including inside functions.
* They have a global scope, meaning they are visible throughout the entire program.
* Global variables are created when they are first assigned a value and persist until the program terminates.
* If you modify a global variable inside a function, you need to explicitly declare it as global using the global keyword.

Ex:

y = 20 # Global variable

x = 200

def my\_function():

global y # Declaring y as global

print("Inside function y:", y)

#print("Inside function x:", x) # it returns error as x is not

defined locally

y = 30 # Modifying global variable

x = 300 # Modifying global variable, but its local only

print("Inside function x:", x) # Output: 300

my\_function()

print("Outside function y:", y) # Output: 30

print("Outside function x:", x) # Output: 200

# 23rd March

## Functions as Generators

By using below keywords functions will act as generators

* Yield
* Generator

Functions in Python can act as generators when they contain one or more yield statements. Generators are a special type of iterator that generates values lazily, one at a time, instead of computing and storing all values at once. This allows generators to be memory-efficient, especially when dealing with large or infinite sequences.

Ex1: This example available in “**23rd March.ipynb**”

student\_info = {

"abc" : 50, "def" : 60, "ghi" : 70, "ijk" : 80, "lmn" : 90

}

def information\_generator(student\_info) :

for name, marks in student\_info.items():

yield(name, marks);

student\_generator = information\_generator(student\_info)

next(student\_generator)

In above example, next will print the student info by pausing and resuming the method “information\_generator”

Ex2:

def countdown(n):

while n > 0:

yield n

n -= 1

# Creating a generator object

generator = countdown(5)

# Iterating over the generator to retrieve values

for value in generator:

print(value)

In this example:

* The countdown() function contains a while loop that decrements the value of n and yields the current value of n in each iteration.
* When yield is encountered, the function returns the value and temporarily suspends its execution, preserving its state.
* Each time the generator object is iterated over (e.g., using a for loop), the function resumes from the last yield statement and continues execution until the next yield.
* The for loop iterates over the generator, retrieving one value at a time until the generator is exhausted.

### Return vs Yield

| **Feature** | **return Statement** | **yield Statement** |
| --- | --- | --- |
| Purpose | Used to return a value from a function. | Used to turn a function into a generator. |
| Execution | Terminates the function immediately. | Temporarily suspends function execution. |
| Value(s) | Can return a single value or multiple values (as a tuple). | Yields a single value each time it is called. |
| State | Does not preserve the function's state. | Preserves the function's state between calls. |
| Function type | Can only be used in regular functions. | Used exclusively in generator functions. |
| Iteration | Cannot be used for lazy iteration. | Used for lazy iteration, producing values on-demand. |
| Control flow | Execution continues from the caller after the return statement. | Execution resumes from the last yield statement. |
| Example | ```python | ```python |
|  | def add(a, b): | def generate\_numbers(): |
|  | return a + b | for i in range(5): |
|  | result = add(3, 5) | yield i |
|  | print(result) # Output: 8 | numbers\_generator = generate\_numbers() |
|  | ``` | for num in numbers\_generator: |
|  |  | print(num) # Output: 0, 1, 2, 3, 4 |

### Real Time Applications

Generators are versatile constructs in Python that can be used in various scenarios to achieve different objectives. Some common applications of generators include:

1. **Lazy Evaluation**:
   * Generators are commonly used for lazy evaluation, where values are generated on-the-fly as needed, rather than pre-computed and stored in memory. This is especially useful when dealing with large or infinite sequences.
2. **Processing Large Datasets**:
   * Generators are memory-efficient, making them well-suited for processing large datasets that cannot fit into memory all at once. They allow you to read, process, and generate data in chunks, minimizing memory usage.
3. **File I/O**:
   * Generators can be used to read and process large files line by line or in chunks. Instead of reading the entire file into memory, you can use a generator to yield lines or blocks of data as needed.
4. **Data Streaming**:
   * Generators are ideal for streaming data over networks or other I/O operations. They enable you to process and transmit data in real-time, without buffering or storing large amounts of data in memory.
5. **Infinite Sequences**:
   * Generators can produce infinite sequences of values, such as numerical sequences, mathematical series, or streams of random numbers. They allow you to work with sequences of arbitrary length without consuming excessive memory.
6. **Pipeline Processing**:
   * Generators can be chained together in a pipeline to perform multi-stage data processing. Each generator in the pipeline consumes input from the previous generator, processes it, and yields output to the next generator, enabling modular and composable data processing workflows.
7. **Asynchronous Programming**:
   * Generators can be used in conjunction with asynchronous programming techniques, such as coroutines and async/await syntax, to implement non-blocking I/O operations and concurrent execution. They allow you to write asynchronous code that is more readable and maintainable.
8. **Stateful Iterators**:
   * Generators can maintain internal state between iterations, allowing them to implement complex behavior or algorithms that require persistent state. This makes them suitable for implementing stateful iterators, such as parsers, state machines, or generators with memoization.

## Using Modules

Modules in Python are files containing Python code that can define functions, classes, and variables. They allow you to organize code into reusable units and facilitate better code organization, modularity, and reusability.

Here's how you can use modules in Python:

### Creating a Module

You can create a module by saving Python code in a .py file. For example, suppose you have a file named my\_module.py with the following content:

# my\_module.py

def greet(name):

return f"Hello, {name}!"

def add(a, b):

return a + b

pi = 3.14159

### Importing a Module

To use the functions and variables defined in a module, you need to import it into your Python script or interactive session.

# Importing the entire module

import my\_module

# Using functions and variables from the module

print(my\_module.greet("Alice")) # Output: Hello, Alice!

print(my\_module.add(3, 5)) # Output: 8

print(my\_module.pi) # Output: 3.14159

### Importing Specific Functions or Variables

You can import specific functions or variables from a module using the from keyword.

# Importing specific functions and variables from the module

from my\_module import greet, pi

# Using the imported functions and variables directly

print(greet("Bob")) # Output: Hello, Bob!

print(pi) # Output: 3.14159

### Renaming Imported Modules or Symbols

You can rename imported modules or symbols using the as keyword.

# Importing a module with a different name

import my\_module as mm

# Using the module with the new name

print(mm.greet("Charlie")) # Output: Hello, Charlie!

### Accessing Module Documentation:

You can access documentation for modules, functions, and variables using the help() function or by accessing the \_\_doc\_\_ attribute.

# Accessing module documentation

print(help(my\_module))

# Accessing function documentation

print(help(my\_module.greet))

Above Code snippet will give following output

Help on module my\_module:

NAME

my\_module - # my\_module.py

FUNCTIONS

add(a, b)

greet(name)

DATA

pi = 3.14159

FILE

c:\users\windows 10\my\_module.py

None

Help on function greet in module my\_module:

greet(name)

None

Using modules in Python allows you to organize your code into logical units, manage dependencies, and promote code reuse. Modules are a fundamental concept in Python programming and are widely used in practice to build complex software systems.

## Classes

In Python, a class is a blueprint for creating objects (instances) with specific properties and behaviors. Classes are fundamental to object-oriented programming (OOP) and provide a way to model real-world entities as software objects. Each class defines a set of attributes (data) and methods (functions) that operate on the data.

Here's how you can define and use classes in Python:

### Defining a Class

You can define a class using the class keyword, followed by the class name and a colon. Inside the class body, you can define attributes and methods.

class Person:

# Constructor method (\_\_init\_\_)

def \_\_init\_\_(self, name, age):

self.name = name # Attribute

self.age = age # Attribute

# Method

def greet(self):

return f"Hello, my name is {self.name} and I'm {self.age} years old."

### Creating Objects (Instances)

Once a class is defined, you can create objects (instances) of that class using the class name followed by parentheses. The constructor method \_\_init\_\_() is called automatically to initialize the object's attributes.

# Creating objects (instances) of the Person class

person1 = Person("Alice", 30)

person2 = Person("Bob", 25)

### Accessing Attributes and Calling Methods:

You can access an object's attributes using dot notation (object.attribute) and call its methods using dot notation as well (object.method()).

# Accessing attributes and calling methods

print(person1.name) # Output: Alice

print(person2.age) # Output: 25

print(person1.greet()) # Output: Hello, my name is Alice and I'm 30 years old.

### Self key word

self is a special keyword that represents the instance (object) of a class. It is the first parameter of instance methods in a class and is passed automatically when calling the method. The purpose of self is to reference the current instance of the class, allowing you to access and modify attributes and call other methods within the class.

Here's how self is used in Python classes:

#### Within Instance Methods

In Python, instance methods within a class must have self as the first parameter, although you can choose any name for this parameter (although self is the convention and widely used).

class MyClass:

def \_\_init\_\_(self, x):

self.x = x # Assigning the value of 'x' to the instance attribute 'self.x'

def print\_x(self):

print(self.x) # Accessing the instance attribute 'x' using 'self'

# Creating an instance of MyClass

obj = MyClass(10)

# Calling an instance method using the object

obj.print\_x() # Output: 10

#### Accessing Instance Attributes

Within instance methods, self is used to access instance attributes (variables) and call other instance methods. Without self, Python would not know which instance's attributes or methods you're referring to.

#### Referencing the Current Instance

When you call an instance method on an object, Python automatically passes the instance itself as the first argument (self). This allows you to work with the data associated with the specific instance that the method is being called on.

#### Differentiating Between Instance and Class Variables

Using self helps differentiate between instance variables (belonging to a specific instance) and class variables (shared among all instances of the class). By prefixing instance variables with self, you make it clear that they belong to the instance.

Ex:

Here's an example demonstrating the use of self within a class method:

class Counter:

def \_\_init\_\_(self):

self.count = 0

def increment(self):

self.count += 1

def get\_count(self):

return self.count

# Creating an instance of Counter

counter = Counter()

# Calling methods on the instance

counter.increment()

counter.increment()

print(counter.get\_count()) # Output: 2

In this example, self.count refers to the count attribute of the specific instance (counter), allowing each instance of Counter to maintain its own count.

### Constructors

A constructor is a special method used for initializing new instances (objects) of a class. The constructor method is called \_\_init\_\_() and is defined within a class. When a new instance of the class is created, the constructor method \_\_init\_\_() is automatically called.

Ex:

class MyClass:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def display\_info(self):

print("Name:", self.name)

print("Age:", self.age)

# Creating an instance of the class with constructor arguments

obj1 = MyClass("Alice", 30)

# Calling a method of the object

obj1.display\_info()

### Class Inheritance

Python supports class inheritance, allowing you to create subclasses that inherit attributes and methods from a parent (base) class. Subclasses can also define additional attributes and methods.

class Person:

def \_\_init\_\_(self, name):

self.name = name

def displayName(self):

print(f"The person name is: {name}")

# Creating a subclass of Employee

class Employee(Person):

def \_\_init\_\_(self, empName, empNo):

super().\_\_init\_\_(empName)

self.empNo = empNo

def displayEmployee(self):

print(f"{self.name} --> {self.empNo}")

# Creating an object of the Employee class

emp = Employee("ABC", 70)

emp.displayEmployee() # Output: ABC --> 70

### Class Documentation

You can provide documentation for a class using docstrings, which are enclosed in triple quotes (""" ... """) and appear as the first statement in the class definition.

class Person:

"""A class representing a person."""

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

...

In summary, classes in Python provide a way to define custom data types with attributes and methods. They encapsulate related data and functionality, promoting code organization, modularity, and reusability. By creating and using classes, you can model complex systems and build object-oriented software solutions in Python.

### Private Variables

can create private variables in a class by prefixing the variable name with double underscores \_\_. This naming convention indicates that the variable is intended to be private and should not be accessed directly from outside the class. However, it's important to note that Python does not enforce strict access control like some other programming languages, so the concept of private variables is more of a convention than a rule.

Here's how you can define and use private variables in a class:

class Circle:

def \_\_init\_\_(self, radius):

self.\_\_radius = radius # Private instance attribute

def get\_radius(self):

return self.\_\_radius # Getter method for radius

def set\_radius(self, radius):

if radius > 0:

self.\_\_radius = radius # Setter method for radius

def area(self):

return 3.14 \* self.\_\_radius \*\* 2

# Creating an instance of the Circle class

circle = Circle(5)

# Accessing private variable indirectly using getter method

print("Radius:", circle.get\_radius()) # Output: Radius: 5

# Trying to access private variable directly

# This will result in an AttributeError

# print("Radius:", circle.\_\_radius)

# Modifying private variable indirectly using setter method

circle.set\_radius(7)

print("Radius after setting:", circle.get\_radius()) # Output: Radius after setting: 7

# Calculating area using public method

print("Area:", circle.area()) # Output: Area: 153.86

# 24th March

### Iterators in Classes

Iterators are objects that implement the iterator protocol, which consists of the \_\_iter\_\_() and \_\_next\_\_() methods. An iterator is an object that represents a stream of data and allows you to iterate over its elements one at a time. You can create iterators in classes by implementing these methods within the class definition.

Here's an example of how to create an iterator in a class:

class MyIterator:

def \_\_init\_\_(self, max\_value):

self.max\_value = max\_value

self.current\_value = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current\_value < self.max\_value:

self.current\_value += 1

return self.current\_value

else:

raise StopIteration

# Creating an instance of the MyIterator class

iterator = MyIterator(5)

# Iterating over the iterator using a for loop

for num in iterator:

print(num)

In this example:

* We define a class MyIterator that implements the iterator protocol by providing the \_\_iter\_\_() and \_\_next\_\_() methods.
* The \_\_init\_\_() method initializes the iterator with a maximum value.
* The \_\_iter\_\_() method returns the iterator object itself.
* The \_\_next\_\_() method generates the next value in the iterator's sequence and raises a StopIteration exception when the sequence is exhausted.
* We create an instance of the MyIterator class with a maximum value of 5.
* We iterate over the iterator using a for loop, which internally calls the \_\_iter\_\_() and \_\_next\_\_() methods to retrieve each element from the iterator.

## Polymorphism

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to be used to manipulate objects of different types, providing flexibility and extensibility in software design.

Polymorphism can be achieved by using following two methods:

* Method Overriding
* Method Overloading

### Method Overriding

Method overriding allows subclasses to provide a specific implementation of a method that is already defined in its superclass.

Ex:

class Animal:

def speak(self):

pass # Placeholder method to be overridden by subclasses

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

# Function that takes any Animal object and calls its speak() method

def make\_sound(animal):

return animal.speak()

# Creating instances of different subclasses

dog = Dog()

cat = Cat()

# Calling the make\_sound function with different types of animals

print(make\_sound(dog)) # Output: Woof!

print(make\_sound(cat)) # Output: Meow!

### Method Overloading

Method overloading enables multiple methods with the same name but different signatures to exist within the same class.

This can be achieved using below two methods:

* Variable Length Argument Lists
* Default Parameter Values

#### Variable Length Argument Lists

Ex:

class MathOperations:

def add(self, \*nums):

result = 0

for num in nums:

result += num

return result

# Creating an instance of MathOperations class

math\_ops = MathOperations()

# Calling the add() method with different number of arguments

print(math\_ops.add(2, 3)) # Output: 5

print(math\_ops.add(2, 3, 4, 5)) # Output: 14

print(math\_ops.add(2, 3, 4, 5, 6)) # Output: 20

#### Default Parameter Values

Ex:

class MathOperations:

def add(self, a, b=0):

return a + b

# Creating an instance of MathOperations class

math\_ops = MathOperations()

# Calling the add() method with different number of arguments

print(math\_ops.add(2, 3)) # Output: 5

print(math\_ops.add(5)) # Output: 5 (default value of b is used)

#### KWARGS

kwargs is a special syntax used in function definitions to allow a function to accept an arbitrary number of keyword arguments. The \*\*kwargs parameter collects any keyword arguments passed to the function into a dictionary.

Ex:

def my\_function(\*\*kwargs):

for key, value in kwargs.items():

print(key, ":", value)

# Calling the function with multiple keyword arguments

my\_function(name="Alice", age=30, city="New York")

## The Pass Keyword

The pass keyword is a null statement that does nothing when executed. It is used as a placeholder to indicate that no action needs to be taken or no code needs to be executed at a particular point in the program.

Here are some common use cases for the pass keyword:

### Placeholder for Future Code

We can use pass as a placeholder when writing skeleton code or defining functions, classes, or conditional blocks that you plan to implement later.

Ex:

def some\_function():

pass # Placeholder for future implementation

class MyClass:

def some\_method(self):

pass # Placeholder for future implementation

if condition:

pass # Placeholder for future implementation

### Empty Code Blocks

In cases where Python syntax requires an indented block of code but you don't want to execute any code within that block, you can use pass to satisfy the syntax requirements.

Syntax:

if condition:

# Empty if block

pass

### Infinite Loops

pass can be used as the body of an infinite loop when you want to create a loop structure that does nothing until terminated by an external condition.

Syntax:

while True:

pass # Infinite loop

### Placeholders in Classes and Functions

We can use pass as a placeholder inside classes or functions to temporarily satisfy syntax requirements without implementing any functionality.

Ex:

class MyClass:

pass # Placeholder for class definition

def my\_function():

pass # Placeholder for function definition

In summary, the pass keyword is a no-operation statement used as a placeholder in Python code where a statement is syntactically required but no action needs to be taken. It is often used to indicate that a block of code is intentionally left empty or as a placeholder for future implementation.

## File Operation

### Opening a File

We can open a file using the built-in open() function. It takes two parameters: the file path and the mode (e.g., "r" for reading, "w" for writing, "a" for appending, etc.).

Ex:

# Open a file for reading

file = open("example.txt", "r")

### Reading from a File

We can read the contents of a file using various methods like read(), readline(), or readlines().

Ex:

# Read the entire contents of the file

content = file.read()

# Read one line from the file

line = file.readline()

# Read all lines from the file into a list

lines = file.readlines()

#### End of File

In Python, there isn't a special "end of file" (EOF) character or marker like some other programming languages. Instead, Python file objects have a behavior that allows them to indicate the end of the file when you try to read from it.

When you read from a file using methods like read(), readline(), or readlines(), Python returns an empty string ('') when it reaches the end of the file. This empty string serves as an indication that there are no more characters or lines left to read.

Ex:

file2 = open("example.txt", "r")

while True:

content = file2.readline()

# Keep reading lines until the end of the file

if (content == ''):

print("reached end of the file")

break

else:

print(content)

file2.close()

### Writing to a File

We can write data to a file using the write() method. If the file does not exist, it will be created. If it already exists, its contents will be overwritten.

Ex:

# Open a file for writing

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

# Write data to the file

file.write("Hello, World!\n")

file.write("This is a new line.")

### Appending to a File

We can append data to the end of a file using the "a" mode or the write() method.

Ex:

# Open a file for appending

file = open("example.txt", "a")

# Append data to the file

file.write("\nThis is appended data.")

### Closing a File

After you're done with a file, it's good practice to close it using the close() method to release system resources.

Ex:

# Close the file

file.close()

### Using "with" Statement (Context Manager):

Python supports the "with" statement, which automatically closes the file when the block is exited, ensuring proper resource cleanup.

Ex:

with open("example.txt", "r") as file:

content = file.read()

# File is automatically closed when the block is exited

### File Seek and Tell

You can use the seek() method to change the file's current position, and the tell() method to get the current position within the file.

Ex:

file.seek(0) # Move to the beginning of the file

position = file.tell() # Get the current position within the file

### File Modes

when you open a file using the open() function, you can specify a mode that determines how the file should be opened and what operations are allowed on it. Here are the different modes you can use:

1. **Read Mode ('r')**:
   * Opens the file for reading.
   * The file must exist. If it doesn't, an error will occur.
   * File pointer is placed at the beginning of the file.
   * This is the default mode if no mode is specified.
2. **Write Mode ('w')**:
   * Opens the file for writing.
   * If the file already exists, its contents are truncated (i.e., deleted). If the file does not exist, a new file is created.
   * File pointer is placed at the beginning of the file.
3. **Append Mode ('a')**:
   * Opens the file for appending.
   * If the file exists, new data is written at the end of the file. If the file does not exist, a new file is created.
   * File pointer is placed at the end of the file.
4. **Read and Write Mode ('r+')**:
   * Opens the file for both reading and writing.
   * The file must exist.
   * File pointer is placed at the beginning of the file.
5. **Write and Read Mode ('w+')**:
   * Opens the file for both reading and writing.
   * If the file exists, its contents are truncated. If the file does not exist, a new file is created.
   * File pointer is placed at the beginning of the file.
6. **Append and Read Mode ('a+')**:
   * Opens the file for both reading and appending.
   * If the file exists, new data is written at the end of the file. If the file does not exist, a new file is created.
   * File pointer is placed at the end of the file.
7. **Binary Mode ('b')**:
   * Appended to any of the above modes to indicate that the file should be treated as a binary file.
   * For example, 'rb' for reading a binary file, 'wb' for writing a binary file, etc.

You can combine these modes as needed, for example, 'r' for reading, 'w' for writing, 'a' for appending, 'r+' for reading and writing, etc.

Here's an example of opening a file in different modes:

Ex:

# Open a file for reading

file = open("example.txt", "r")

# Open a file for writing (creates a new file if it doesn't exist)

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

# Open a file for appending (creates a new file if it doesn't exist)

file = open("example.txt", "a")

# Open a file for reading and writing (creates a new file if it doesn't exist)

file = open("example.txt", "r+")

# Open a binary file for reading and writing

file = open("example.bin", "rb+")

# Open a text file for reading and writing

file = open("example.txt", "rt+")

When you're done working with a file, remember to close it using the close() method or by using a with statement.

## Exception Handling

Exception handling in Python allows you to deal with runtime errors or exceptional situations gracefully, preventing your program from crashing unexpectedly. Python provides a mechanism to catch and handle exceptions using the try, except, else, and finally blocks.

Here's a brief overview of each part of the exception handling mechanism:

1. **try**: The try block is used to enclose the code that may potentially raise an exception.
2. **except**: The except block is used to handle specific exceptions that occur within the try block. You can have multiple except blocks to handle different types of exceptions. If an exception of the specified type occurs in the try block, control flows to the corresponding except block.
3. **else**: The else block is optional and is executed only if no exceptions occur in the try block.
4. **finally**: The finally block is optional and is always executed, regardless of whether an exception occurs or not. It is typically used to perform cleanup actions, such as closing files or releasing resources.

Ex:

try:

# Code that may raise an exception

num1 = int(input("Enter a number: "))

num2 = int(input("Enter another number: "))

result = num1 / num2

except ZeroDivisionError as zde:

# Handle division by zero exception

print("Error: Cannot divide by zero.")

except ValueError:

# Handle invalid input exception

print("Error: Invalid input. Please enter a valid number.")

except Exception as e:

# Handle invalid input exception

print(f"got exception {e}")

else:

# Execute if no exception occurs

print("Result:", result)

finally:

# Always execute, regardless of exceptions

print("Exiting the program.")

### Raise

the raise statement is used to explicitly raise exceptions or errors during the execution of a program. It allows you to generate custom exceptions or propagate built-in exceptions to higher levels of code for handling.

Here's how the raise statement is typically used:

**Raising Built-in Exceptions**: We can raise any built-in exception by specifying its type.

raise ValueError("Invalid input")

**Raising Custom Exceptions**: We can define your own custom exceptions by creating a new exception class that inherits from the built-in Exception class, and then raise instances of that class.

class CustomError(Exception):

pass

raise CustomError("This is a custom error message")

**Raising Exceptions with Arguments**: We can provide additional information or arguments when raising exceptions.

divisor = 0

if divisor == 0:

raise ZeroDivisionError("Cannot divide by zero")

**Raising Caught Exceptions**: We can re-raise exceptions that were caught and handled by a try-except block, allowing them to propagate to higher levels of code for further handling.

def example\_function():

try:

# Some code that may raise exceptions

raise ValueError("An error occurred in example\_function")

except ValueError as e:

# Handle the exception

print("Caught and handled the exception:", e)

# Re-raise the exception to propagate it further

raise

try:

# Call the function that may raise exceptions

example\_function()

except ValueError as e:

# Handle the re-raised exception

print("Caught the re-raised exception:", e)

When we use the raise statement, Python interrupts the normal flow of execution and searches for the nearest exception handler (try-except block) to handle the raised exception. If no suitable handler is found, the program terminates, and Python displays the exception traceback.

The raise statement is useful for signaling errors or exceptional situations in your code and allows you to gracefully handle them using exception handling mechanisms.

## Decorators

decorators are a powerful and flexible feature that allows you to modify or extend the behavior of functions or methods without directly modifying their code. Decorators are functions themselves that wrap other functions or methods to add functionality to them. They are commonly used for implementing cross-cutting concerns such as logging, authentication, caching, and more.

Here's a basic overview of how decorators work:

1. **Decorator Function**: A decorator is a function that takes another function as its argument, adds some functionality to it, and then returns the modified function.
2. **Syntax**: Decorators use the @decorator\_name syntax, where decorator\_name is the name of the decorator function. This syntax is placed on top of the function to be decorated.
3. **Execution Order**: Decorators are executed at the time the function they decorate is defined, not when the function is called.

Ex1:

# Decorator function

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func() # Call the original function

print("Something is happening after the function is called.")

return wrapper

# Function to be decorated

@my\_decorator

def say\_hello():

print("Hello!")

# Calling the decorated function

say\_hello()

Output:

Something is happening before the function is called.

Hello!

Something is happening after the function is called.

In this example:

* We define a decorator function my\_decorator, which takes another function func as its argument.
* Inside my\_decorator, we define a nested function wrapper, which adds some functionality before and after calling the original function func.
* We decorate the say\_hello function using @my\_decorator syntax. This is equivalent to calling say\_hello = my\_decorator(say\_hello).
* When we call say\_hello(), it actually calls the wrapper function, which in turn calls the original say\_hello function with additional functionality added by the decorator.

Decorators are widely used in Python for various purposes such as logging, caching, authentication, performance monitoring, and more. They provide a clean and concise way to enhance the behavior of functions or methods without modifying their source code directly.

Ex2:

This example uses the parameters in the method.

import time

# Decorator function

def myDecorator(func):

def myWrapper(**name**):

start\_time = time.time()

print("Something is happening before the function is called.")

func(name) # Call the original function

print("Something is happening after the function is called.")

end\_time = time.time()

print(f"The time taken to execute the method {func} is {end\_time-start\_time:.20f}")

return myWrapper

# Function to be decorated

@myDecorator

def say\_hello(name):

print(f"Hello using decorators {name}!")

# Calling the decorated function

say\_hello("test")

Output:

Something is happening before the function is called.

Hello using decorators test!

Something is happening after the function is called.

The time taken to execute the method <function say\_hello at 0x000001A335BFE700> is 0.00099968910217285156

Ex3:

In this example we are re-using above defined decorators

# New function to be decorated

@myDecorator

def newMethod(name):

print("Inside new Method block")

Output:

Something is happening before the function is called.

Inside new Method block

Something is happening after the function is called.

The time taken to execute the method <function newMethod at 0x000001A336324B80> is 0.00225019454956054688

# 30th March

## Static Method

A static method is a method that belongs to a class but does not operate on instances of that class. Unlike instance methods, which require an instance of the class as the first parameter (self by convention), static methods do not have access to the instance or class attributes and can be called directly from the class itself.

Static methods are defined using the @staticmethod decorator or by using the staticmethod() built-in function. They are typically used when a method does not depend on instance or class state and can be implemented as a utility function associated with the class.

Ex:

class MathUtils:

@staticmethod

def add(x, y):

return x + y

@staticmethod

def subtract(x, y):

return x - y

# Calling static methods directly from the class

sum\_result = MathUtils.add(5, 3)

difference\_result = MathUtils.subtract(5, 3)

print("Sum:", sum\_result) # Output: 8

print("Difference:", difference\_result) # Output: 2

## Custom Environment

Note:

* ‘Base’ is the default environment

Following command creates an environment

>conda create -n intellipaat\_test python==3.10

Below command lists all available environments

> conda info --envs

Following command uses/switches an environment

> conda activate intellipaat\_test

Following command is to deactivate any environment

> conda deactivate

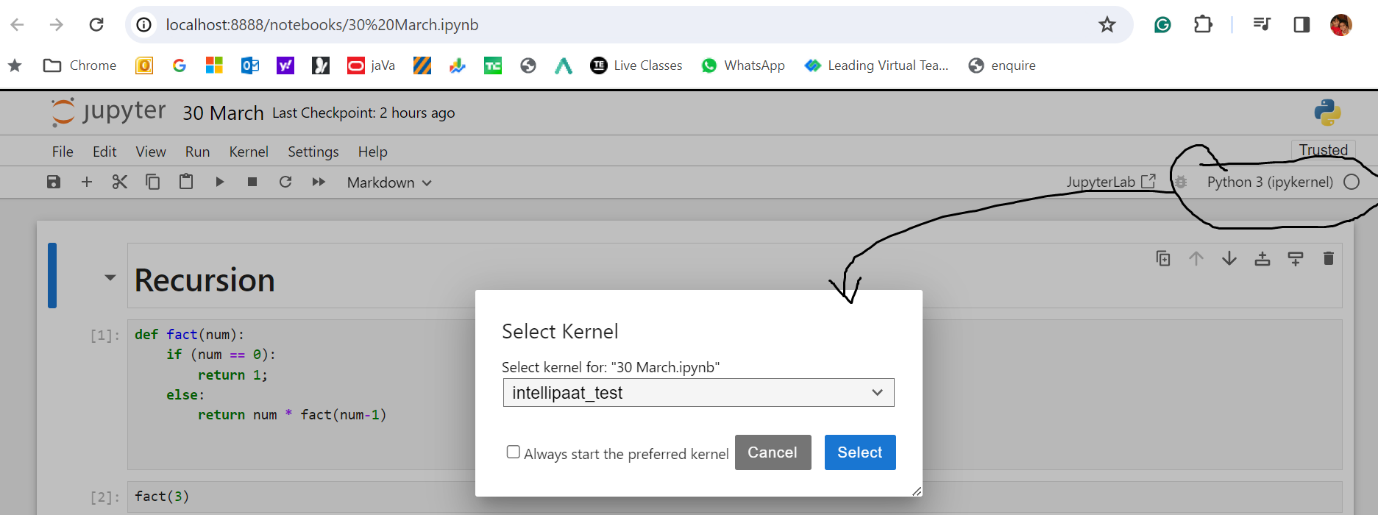
Following is the command to install ipykernal

>pip install ipykernel

Following is the command to activate the environment in Jupiter notebook

>python -m ipykernel install --user --name=intellipaat\_test

Once we open Jupiter notebook we have to select the Kernel from the top right corner which is depicted in the below image.



# 31st March

## NumPy

NumPy, which stands for Numerical Python, is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is an essential library for scientific computing and data analysis in Python.

Here are some key features of NumPy:

1. **Multi-dimensional Arrays**: NumPy provides an ndarray (n-dimensional array) data structure, which is a powerful and efficient container for arrays of homogeneous data types. These arrays can be one-dimensional, two-dimensional, or multi-dimensional.
2. **Vectorized Operations**: NumPy allows you to perform mathematical and logical operations on arrays without the need for explicit looping, using a syntax that is similar to scalar operations on individual elements.
3. **Broadcasting**: NumPy provides broadcasting, which is a powerful mechanism that allows arrays of different shapes to be combined together in arithmetic operations. This makes it possible to perform operations on arrays of different shapes without explicitly reshaping them.
4. **Mathematical Functions**: NumPy includes a wide range of mathematical functions for performing arithmetic, statistical, and linear algebra operations on arrays. These functions are optimized for performance and can operate on entire arrays efficiently.
5. **Random Number Generation**: NumPy includes a random module (numpy.random) for generating random numbers from various probability distributions. This is useful for simulations, random sampling, and generating synthetic data.
6. **Integration with Other Libraries**: NumPy is the foundation for many other scientific computing libraries in Python, including SciPy (Scientific Python), Pandas (Data Analysis), Matplotlib (Data Visualization), and scikit-learn (Machine Learning).

Ex:

import numpy as np

# Create a NumPy array

arr = np.array([1, 2, 3, 4, 5])

# Perform arithmetic operations

print("Sum:", np.sum(arr)) # Output: 15

print("Mean:", np.mean(arr)) # Output: 3.0

print("Square root:", np.sqrt(arr))

# Output: [1. 1.41421356 1.73205081 2. 2.23606798]

NumPy's efficient implementation of arrays and mathematical functions makes it a cornerstone of numerical computing and data analysis in Python.

### Advanced Indexing

Advanced indexing in NumPy refers to the method of selecting subsets of data from NumPy arrays using arrays of indices or boolean masks. It allows for more complex and flexible indexing operations compared to basic indexing (using integers or slices).

There are two main types of advanced indexing in NumPy:

#### Integer Array Indexing

* + Integer array indexing involves using arrays of integers to select specific elements from a NumPy array. Each integer in the index array specifies the position of the element to be selected along a particular axis.
  + Integer array indexing allows for arbitrary selection of elements from different positions in the array.
  + Here's an example of integer array indexing:

python

* 
* import numpy as np
* arr = np.array([[1, 2], [3, 4], [5, 6]])
* indices = np.array([0, 2])
* result = arr[indices, :]
* print(result)
* # Output:
* # [[1 2]
* # [5 6]]

#### Boolean Array Indexing

* Boolean array indexing involves using boolean arrays (also called boolean masks) to select elements from a NumPy array based on a condition. Each element in the boolean array corresponds to a position in the original array, and elements corresponding to True values are selected.
* Boolean array indexing is useful for filtering data based on conditions or criteria.
* Ex:

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

mask = arr > 2

result = arr[mask]

print(result) # Output: [3 4 5]

In this example, the boolean mask arr > 2 creates a boolean array indicating which elements of arr are greater than 2. Then, the boolean array is used to select elements from arr where the corresponding value in the mask is True.

Advanced indexing in NumPy provides a powerful mechanism for selecting subsets of data based on complex criteria, enabling more sophisticated data manipulation and analysis tasks.

#### Complex Indexing Examples

Create a NumPy

student\_marks = np.array([[20,30,40], [10,20,30],[40,50,65]])

Get the marks which have more than 40

student\_marks[student\_marks>40]

Get the marks which have greater than mean of the student marks

student\_marks[student\_marks > np.mean(student\_marks)]

student\_marks[(student\_marks > 20) & (student\_marks < 60)]

### Slicing Operator

the colon (:) is used as a slicing operator. It is commonly used to create slices of sequences, such as lists, strings, and arrays.

In the context of NumPy arrays, the colon (:) is used inside square brackets ([]) to specify slicing along a particular axis. When used in the form start:end, it selects elements starting from the index start up to, but not including, the index end.

Here's a breakdown of the syntax start:end:

* If start is not provided, the slice starts from the beginning of the sequence (index 0).
* If end is not provided, the slice extends to the end of the sequence.
* If both start and end are provided, the slice includes elements from the index start up to, but not including, the index end.

In NumPy arrays, the colon (:) is often used in conjunction with other slicing operators to create more complex slices. For example:

* : selects all elements along a particular axis.
* start: selects elements starting from the index start up to the end of the axis.
* :end selects elements from the beginning of the axis up to, but not including, the index end.
* start:end:step selects elements starting from the index start, up to, but not including, the index end, with a step size of step.

In the context of the code snippet student\_data[:, :2], the colon (:) is used twice to specify slicing along both axes:

* student\_data[:, :2] selects all rows (:) and the first two columns (:2) of the student\_data array.

Ex:

student\_data = np.array([

[18,85,88,92],

[20,75,85,80],

[19,95,90,94],

[21,78,85,88],

[22,82,88,87]

])

student\_data[3:, ]

#output

array([[21, 78, 85, 88],

[22, 82, 88, 87]])

student\_data[:2, ]

#output

array([[18, 85, 88, 92],

[20, 75, 85, 80]])

student\_data[:, 2: ]

#output

array([[18, 85],

[20, 75],

[19, 95],

[21, 78],

[22, 82]])

student\_data[:, 2:]

#output

array([[88, 92],

[85, 80],

[90, 94],

[85, 88],

[88, 87]])

student\_data[:2, :2]

#output

array([[18, 85],

[20, 75]])

#### Where Function

NumPy provides a function called np.where() which serves a similar purpose. The np.where() function is used to return elements chosen from two arrays based on a condition.

Syntax:

np.where(condition, x, y)

* condition: A boolean array-like object or a condition that evaluates to a boolean array. If True, elements from x are chosen, otherwise elements from y.
* x: The value to be used where the condition is True.
* y: The value to be used where the condition is False.

Ex:

np.where(student\_data>70, "Grade A", "Grade B")

#output

array([['Grade B', 'Grade A', 'Grade A', 'Grade A'],

['Grade B', 'Grade A', 'Grade A', 'Grade A'],

['Grade B', 'Grade A', 'Grade A', 'Grade A'],

['Grade B', 'Grade A', 'Grade A', 'Grade A'],

['Grade B', 'Grade A', 'Grade A', 'Grade A']], dtype='<U7')

### Select Function

NumPy provides various functions for selecting elements from arrays based on certain conditions. One such function is np.select(), which allows you to choose elements from a list of arrays based on multiple conditions.

Syntax:

result = np.select(condlist, choicelist, default=0)

Here:

* condlist: A list of boolean arrays or conditions.
* choicelist: A list of arrays or values to choose from based on the corresponding conditions in condlist.
* default (optional): The default value to choose when none of the conditions in condlist are True.

Ex1:

conditions = [ (student\_data >= 90),

(student\_data >= 60) & (student\_data < 90),

(student\_data >= 35) & (student\_data < 60),

(student\_data <35)

]

choices = [ “Grade A”, “Grade B”, “Grade C”, “Grade D”]

np.select(conditions, choices)

#output

array([[‘Grade D’, ‘Grade B’, ‘Grade B’, ‘Grade A’],

[‘Grade D’, ‘Grade B’, ‘Grade B’, ‘Grade B’],

[‘Grade D’, ‘Grade A’, ‘Grade A’, ‘Grade A’],

[‘Grade D’, ‘Grade B’, ‘Grade B’, ‘Grade B’],

[‘Grade D’, ‘Grade B’, ‘Grade B’, ‘Grade B’]], dtype=’<U7’)

Ex2:

Below writes same above example but uses **default** option:

conditions = [ (student\_data >= 90),

(student\_data >= 60) & (student\_data < 90),

(student\_data >= 35) & (student\_data < 60)

]

choices = [ “Grade A”, “Grade B”, “Grade C”]

np.select(conditions, choices, “**Grade D**”)

### arange function

Ex:

np.arange(1,10)

#output

array([1, 2, 3, 4, 5, 6, 7, 8, 9])

Ex2:

# Create an array of numbers from 2 to 20 with a step of 2 array2 = np.arange(2, 21, 2)

print("Array 2:", array2)

### Reshape Function

The reshape() function is used to change the shape (dimensions) of an array without changing its data. It returns a new array with a modified shape, while maintaining the same elements as the original array.

Syntax:

numpy.reshape(array, newshape, order='C')

Here:

* array: The array to be reshaped.
* newshape: The new shape (dimensions) of the array. It can be a tuple specifying the desired dimensions or an integer specifying the size of one dimension (in which case, the other dimensions are inferred).
* order (optional): Specifies the order in which the elements of the array are read. It can be 'C' for row-major (C-style) order or 'F' for column-major (Fortran-style) order. By default, it is 'C'.

Ex:

# Create a 1D array with 12 elements

arr = np.arange(12)

# Reshape the array into a 3x4 matrix

reshaped\_arr = np.reshape(arr, (3, 4))

print("Original array:")

print(arr)

print("\nReshaped array with default option 'C':")

print(reshaped\_arr)

print("\nReshaped array with option 'F':")

print(reshaped\_arr1)

#output

Original array:

[ 0 1 2 3 4 5 6 7 8 9 10 11]

Reshaped array with default option 'C':

[[ 0 1 2 3]

[ 4 5 6 7]

[ 8 9 10 11]]

Reshaped array with option 'F':

[[ 0 3 6 9]

[ 1 4 7 10]

[ 2 5 8 11]]

### Flattening Array

Ex:

arr = np.arange(12)

arr.reshape(1, -1) # row based flattening

arr.reshape(-1, 1) # Column based flattening

#### 3D- reshape

Ex:

arr = np.arange(12)

array\_3d = arr.reshape(2,3,2)

#output

array([[[ 0, 1],

[ 2, 3],

[ 4, 5]],

[[ 6, 7],

[ 8, 9],

[10, 11]]])

# Todo, Have to go recording from 12:30 PM

# 6th April

Download the titanic dataset

Open anaconda prompt

Run below command to activate intellipaat\_test environment

>conda activate intellipaat\_test

Run below command to install pandas

> pip install pandas

## Pandas

Pandas is a popular open-source Python library used for data manipulation and analysis. It provides data structures and functions that make it easy to work with structured data, such as tables and time series data. The primary data structures in Pandas are Series and DataFrame.

* **Series**: A one-dimensional array-like object that can hold any data type, such as integers, floats, strings, etc. Each element in a Series has an associated index.
* **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types. It is similar to a spreadsheet or SQL table, where data is arranged in rows and columns. DataFrames can be thought of as dictionaries of Series objects.

Pandas provides a wide range of functionalities for data manipulation, including:

* Reading and writing data from various file formats (e.g., CSV, Excel, SQL databases, JSON, HTML)
* Data cleaning and preprocessing (e.g., handling missing values, removing duplicates, data normalization)
* Indexing, slicing, and subsetting data
* Aggregation and grouping operations
* Merging, joining, and concatenating datasets
* Time series analysis and manipulation
* Data visualization using integration with other libraries like Matplotlib and Seaborn

Overall, Pandas is widely used in data science, machine learning, and data analysis workflows due to its ease of use, flexibility, and powerful features for handling and analyzing structured data.

Ex for data frame:

import pandas as pd

titanic\_df = pd.read\_csv("titanic.csv")

titanic\_df

#output

It display the titanic.csv file content in tabular form

### Category vs Object Data Types

| **Feature** | **object Data Type** | **category Data Type** |
| --- | --- | --- |
| Purpose | Generic data type for any Python objects, including strings, lists, dictionaries, etc. | Specifically designed for categorical data with a fixed and limited number of unique values. |
| Memory Usage | Can consume more memory, especially for large datasets or columns with heterogeneous data types. | Typically consumes less memory, especially for columns with a limited number of unique values. |
| Performance | Operations may be slower, especially for grouping, sorting, and value counts, due to heterogeneous data types and dynamic nature. | Operations can be faster, especially for grouping, sorting, and value counts, due to efficient storage and handling of categorical values. |
| Efficiency | Less memory-efficient, especially for categorical variables with repetitive values. | More memory-efficient, especially for categorical variables with a limited number of unique values. |
| Flexibility | Provides flexibility to store heterogeneous data types within the same column. | More specialized, suitable for categorical variables with a fixed set of values. |
| Use Cases | Suitable for columns with mixed or heterogeneous data types. | Suitable for categorical variables with a small number of unique values, such as gender, country, product categories, etc. |

This table summarizes the key differences between object and category data types in

### Series

A Series in Pandas is a one-dimensional labeled array capable of holding any data type (integers, floats, strings, Python objects, etc.). It is essentially a single column of data. Each element in a Series has a corresponding index, which by default starts from 0 and increases sequentially.

#### Creating a Series

You can create a Pandas Series from various data types like lists, arrays, dictionaries, etc. Here's how you can create a Series:

import pandas as pd

# Creating a Series from a list

data = [10, 20, 30, 40, 50]

series = pd.Series(data)

print(series)

#Output:

0 10

1 20

2 30

3 40

4 50

dtype: int64

#### Custom Indexing

By default, Pandas assigns numeric indices to the elements in a Series. However, you can provide custom indices as well:

data = [10, 20, 30, 40, 50]

custom\_index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=custom\_index)

print(series)

#Output:

a 10

b 20

c 30

d 40

e 50

dtype: int64

#### Accessing Elements

You can access elements in a Series using index labels:

print(series['b']) # Accessing element with index label 'b'

print(series[1]) # Accessing element at index position 1

#Output:

20

20

#### Operations on Series

Pandas Series supports various mathematical operations, broadcasting, and aggregation:

# Adding a scalar to all elements

print(series + 5)

# Multiplying all elements by a scalar

print(series \* 2)

# Computing the mean of all elements

print(series.mean())

Output:

a 15

b 25

c 35

d 45

e 55

dtype: int64

a 20

b 40

c 60

d 80

e 100

dtype: int64

30.0

#### Series Attributes and Methods

Series objects also have various attributes and methods for exploring and manipulating data, such as dtype, size, shape, head(), tail(), describe(), etc.

print(series.dtype) # Datatype of elements in the Series

print(series.size) # Number of elements in the Series

print(series.shape) # Shape of the Series (returns a tuple)

print(series.head()) # Returns the first few elements of the Series

print(series.describe()) # Generates descriptive statistics of the Series

These are some of the basic functionalities of Pandas Series. It's a powerful tool for handling and analyzing one-dimensional data in Python.

### Data Frame

### Descriptive statistics

Descriptive statistics in Pandas refer to statistical summary measures that provide insights into the characteristics and distribution of a dataset. These summary statistics help analysts and data scientists understand the central tendency, variability, and shape of the data. Pandas provides a variety of methods to compute descriptive statistics for DataFrame and Series objects. Some commonly used descriptive statistics include:

1. **Count**: Number of non-null observations in the dataset.
2. **Mean**: Average value of the observations.
3. **Median**: Middle value of the dataset when it is sorted in ascending order.
4. **Standard Deviation**: Measure of the dispersion or spread of the dataset.
5. **Minimum**: Smallest value in the dataset.
6. **Maximum**: Largest value in the dataset.
7. **Quantiles**: Values that divide the dataset into equal parts, such as quartiles, deciles, or percentiles.
8. **Sum**: Total sum of all values in the dataset.
9. **Describe**: Generates a comprehensive summary of the dataset, including count, mean, standard deviation, minimum, quartiles, and maximum.

These descriptive statistics can be computed using methods available in Pandas DataFrame and Series objects. For example:

#### For a DataFrame

import pandas as pd

# Create a DataFrame

df = pd.DataFrame({

'A': [1, 2, 3, 4, 5],

'B': [5, 10, 15, 20, 25]

})

# Compute descriptive statistics

print(df.**describe()**)

#### For a Series

import pandas as pd

# Create a Series

series = pd.Series([1, 2, 3, 4, 5])

# Compute descriptive statistics

print(series.**describe()**)

These methods provide a quick way to understand the distribution and summary characteristics of the data, which is essential for exploratory data analysis and data preprocessing tasks. Descriptive statistics can reveal outliers, skewness, and other patterns in the data, guiding further analysis and modeling decisions.

#### Explanation on the output

The describe() method in Pandas generates a comprehensive summary of the dataset, providing various descriptive statistics for each numeric column in the DataFrame or Series. Let's break down each component of the result produced by the describe() method:

Here's an example DataFrame:

import pandas as pd

# Create a DataFrame

data = {

'A': [1, 2, 3, 4, 5],

'B': [5, 10, 15, 20, 25]

}

df = pd.DataFrame(data)

print(df.describe())

#output

A B

count 5.000000 5.000000

mean 3.000000 15.000000

std 1.581139 7.905694

min 1.000000 5.000000

25% 2.000000 10.000000

50% 3.000000 15.000000

75% 4.000000 20.000000

max 5.000000 25.000000

Here:

1. **count**: Number of non-null values in each column. It indicates how many observations are present in each numeric column. In this example, both columns 'A' and 'B' have 5 non-null values.
2. **mean**: Average value of the observations in each column. It represents the central tendency of the data. In this case, the mean of column 'A' is 3 and the mean of column 'B' is 15.
3. **std**: Standard deviation, a measure of the dispersion or spread of the data points around the mean. It indicates the variability of the data. A higher standard deviation implies greater variability. In this example, the standard deviation of column 'A' is approximately 1.58, and for column 'B' it's approximately 7.91.
4. **min**: The smallest value in each column.
5. **25% (1st Quartile)**: The value below which 25% of the data falls. Also known as the first quartile or lower quartile.
6. **50% (2nd Quartile / Median)**: The median, which represents the middle value of the data when sorted in ascending order. Half of the observations fall below this value, and half above.
7. **75% (3rd Quartile)**: The value below which 75% of the data falls. Also known as the third quartile or upper quartile.
8. **max**: The largest value in each column.

### Renaming the Columns

Renaming columns in a Pandas DataFrame can be necessary for various reasons, such as improving readability, conforming to a specific naming convention, or resolving naming conflicts. Pandas provides several methods to rename columns effectively:

1. **Using the rename() method**: The rename() method allows you to rename one or more columns by specifying a dictionary where the keys are the current column names and the values are the new column names. You can use this method to rename one or more columns at once.

Example:

import pandas as pd

# Create a DataFrame

data = {

'old\_name1': [1, 2, 3],

'old\_name2': [4, 5, 6]

}

df = pd.DataFrame(data)

# Rename columns

df.rename(columns={'old\_name1': 'new\_name1', 'old\_name2': 'new\_name2'}, inplace=True)

 **Direct assignment to columns attribute**: You can directly assign a list of new column names to the columns attribute of the DataFrame.

Example:

# Directly assign new column names

df.columns = ['new\_name1', 'new\_name2']

 **Using list comprehension or string methods**: You can also use list comprehension or string methods to generate new column names based on existing ones.

Example:

# Using list comprehension

df.columns = ['new\_' + col for col in df.columns]

# Using string methods

df.columns = df.columns.str.replace('old', 'new')

 **Using the set\_axis() method**: The set\_axis() method can be used to set new column names directly. This method provides more flexibility, allowing you to specify axis, inplace, and other parameters.

Example:

# Using set\_axis()

df.set\_axis(['new\_name1', 'new\_name2'], axis='columns', inplace=True)

Remember to set the inplace parameter to True if you want to modify the original DataFrame in place. Otherwise, Pandas will return a new DataFrame with the modified column names. Renaming columns is a common operation in data manipulation workflows and is essential for data preprocessing and analysis.

### Drop Null Values using dropna

The dropna() method in Pandas is used to remove missing values (NaN or None) from a DataFrame or Series. Missing values can occur due to various reasons such as incomplete data, data corruption, or data manipulation operations. Dropping these missing values can be necessary to ensure accurate analysis and modeling.

The dropna() method provides flexibility in handling missing values and allows you to specify different parameters to control the behavior of the operation. Here's how the method works:

### DataFrame:

import pandas as pd

# Create a DataFrame with missing values

data = {

'A': [1, 2, None, 4],

'B': [5, None, 7, 8],

'C': [None, None, None, None]

}

df = pd.DataFrame(data)

# Drop rows with any missing values

cleaned\_df = df.dropna()

# Drop columns with any missing values

cleaned\_df = df.dropna(axis=1)

# Drop rows where all values are missing

cleaned\_df = df.dropna(how='all')

# Drop rows with a minimum number of non-null values

cleaned\_df = df.dropna(thresh=2)

### 

### Series

import pandas as pd

# Create a Series with missing values

series = pd.Series([1, 2, None, 4])

# Drop missing values

cleaned\_series = series.dropna()

### Parameters:

* **axis**: Specifies whether to drop rows (axis=0) or columns (axis=1) containing missing values. By default, it drops rows (axis=0).
* **how**: Specifies whether to drop rows or columns where any or all values are missing. Options are 'any' (default) or 'all'.
* **thresh**: Specifies the minimum number of non-null values required to keep a row or column. Rows or columns with fewer than this threshold will be dropped.
* **subset**: Specifies a subset of columns or rows to consider for dropping missing values. Only those columns or rows specified will be examined.

By default, dropna() returns a new DataFrame or Series with missing values removed. However, you can use the inplace=True parameter to modify the original DataFrame or Series in place.

### Drop Method

#### Delete Columns

titanic\_df.drop(["Name", "Ticket"], axis=1, inplace=True)

Here:

* We are deleting column wise deletion
  + It will be indicates by column axis = 1
* Deleting columns
  + Name
  + Ticket
* Inplace=True means delete without displaying the output

#### Delete Rows

titanic\_df.drop(1)

titanic\_df.drop(1, axis=0, inplace=True)

Here:

* We are deleting a row
* Deleting a row which have index as ‘1’
* Axis attribute is optional and default value is ‘0’ which indicates row deletion.

### Filling the Null Values using fillna method

titanic\_df["Age"] = titanic\_df["Age"].fillna(titanic\_df["Age"].mean())

### Change the column Type

Below code snippet change the data type to Category type

titanic\_df["Sex"] = titanic\_df["Sex"].astype('category')

### Indexing using iloc and loc

By using two different methods we can use indexes in the Pandas

* Iloc
  + Ex:
    - titanic\_df.iloc[:5,3:5]
* loc
  + Ex:
    - titanic\_df.loc[1:3, "Name":"Age"]
      * It will display 1& 2 rows
      * It will display column starting from “Name” to “Age”
    - #output

|  | **Name** | **Sex** | **Age** |
| --- | --- | --- | --- |
| **1** | Cumings | female | 38.0 |
| **2** | Heikkinen | female | 26.0 |
| **3** | Futrelle | female | 35.0 |

#### Loc vs iloc

| **Feature** | **loc** | **iloc** |
| --- | --- | --- |
| Syntax | DataFrame.loc[row\_label, column\_label] | DataFrame.iloc[row\_index, column\_index] |
| Indexing | Label-based indexing | Integer-based indexing |
| Selection | Selects data based on row and column labels | Selects data based on row and column integer indices |
| Usage | Primarily used with label indices | Primarily used with integer indices |
| Inclusive/Exclusive | Inclusive of the end index | Exclusive of the end index |
| Error handling | Raises KeyError if labels don't exist | Raises IndexError if index is out of range |
| Flexibility | Allows selection based on labels that don't exist | Requires existing integer indices |

### Sort\_values method

Ex:

titanic\_df.sort\_values(by='Age')

titanic\_df.sort\_values(by=['Age'])

titanic\_df.sort\_values(by='Age', ascending=False)

titanic\_df.sort\_values(by=['Age', 'Fare'])

### Boolean Indexing

Ex:

titanic\_df[titanic\_df['Age'] > 35]

titanic\_df[(titanic\_df['Age'] > 35) & (titanic\_df['Age'] < 50)]

### GroupBy

The groupby() function in Pandas is used to group DataFrame rows based on one or more columns, allowing for efficient data aggregation and analysis within each group. Here are some examples demonstrating the use of groupby():

Example 1: Grouping by a Single Column and Computing Aggregations

import pandas as pd

# Create a DataFrame

data = {

'Category': ['A', 'B', 'A', 'B', 'A'],

'Value': [10, 20, 30, 40, 50]

}

df = pd.DataFrame(data)

# Group by 'Category' column and compute mean value for each group

grouped = df.groupby('Category').mean()

print(grouped)

Output:

Value

Category

A 30

B 30

Example 2: Grouping by Multiple Columns and Computing Aggregations

# Group by multiple columns ('Category' and 'Type') and compute sum for each group

grouped = df.groupby(['Category', 'Type']).sum()

print(grouped)

Output:

Value

Category Type

A X 40

Y 60

B X 20

Y 40

Example 3: Iterating Over Groups

# Iterating over groups and printing each group

for group\_name, group\_data in df.groupby('Category'):

print(f'Category: {group\_name}')

print(group\_data)

print()

Output:

Category: A

Category Value

0 A 10

2 A 30

4 A 50

Category: B

Category Value

1 B 20

3 B 40

Example 4: Applying Custom Aggregation Functions

# Define custom aggregation function

def my\_custom\_sum(x):

return x.sum() \* 2

# Apply custom aggregation function to 'Value' column

grouped = df.groupby('Category')['Value'].agg(my\_custom\_sum)

print(grouped)

Output:

Category

A 180

B 120

Name: Value, dtype: int64

These examples demonstrate the versatility of the groupby() function in Pandas for grouping and aggregating data based on one or more columns. It's a powerful tool for performing complex data analysis and manipulation tasks efficiently.

## Lambda Functions

A lambda function is a small anonymous function that can have any number of arguments but can only have one expression. They are often referred to as "anonymous" functions because they do not have a name like a regular function defined with the def keyword. Instead, they are defined using the lambda keyword.

Syntax:

lambda arguments: expression

Here,

* lambda is the keyword that indicates the definition of a lambda function.
* arguments are the parameters of the function.
* expression is a single expression that is evaluated and returned by the function.

### Example:

Here's a simple example of a lambda function that calculates the square of a number:

square = lambda x: x \*\* 2

# Calling the lambda function

result = square(5)

print(result) # Output: 25

### Use Cases

Lambda functions are typically used in situations where you need a small function for a short period and don't want to define a formal function using the def keyword. Some common use cases include:

* Passing a function as an argument to higher-order functions like map(), filter(), and sorted().
* Writing short and concise code in situations where a full function definition might be unnecessary or cumbersome.
* Creating simple callbacks and inline functions.

### Limitations

* Lambda functions are limited to a single expression. They cannot contain multiple expressions or statements.
* They are often less readable than regular functions, especially for complex operations.
* Debugging lambda functions can be challenging due to their anonymous nature.

Although lambda functions are useful for simple and short-lived operations, it's generally recommended to use regular functions for more complex tasks or when the code needs to be more readable and maintainable.

### Using map() with Lambda Function

The map() function applies a given function to each item of an iterable (such as a list) and returns a new iterable with the results.

Example: Using map() to square each element in a list.

# Original list

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

# Using map() with a lambda function to square each element

squared\_numbers = list(map(lambda x: x \*\* 2, numbers))

print(squared\_numbers) # Output: [1, 4, 9, 16, 25]

### Using filter() with Lambda Function

The filter() function filters elements from an iterable based on a given function (predicate) and returns an iterator containing the elements for which the function returns True.

Syntax:

filter(lambda argument: expression, iterable)

Here:

* lambda argument: expression: This is the lambda function that defines the condition for filtering. The argument represents each element of the iterable, and the expression evaluates to True or False based on some condition.
* iterable: This is the iterable (e.g., list, tuple, etc.) from which elements will be filtered based on the lambda function.

Example: Using filter() with a lambda function to filter even numbers from a list.

# Original list

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

# Using filter() with a lambda function to filter even numbers

even\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even\_numbers) # Output: [2, 4]

### Sorting with Lambda Function

The sorted() function sorts the elements of a list or iterable based on a key function. Lambda functions are commonly used as the key function to specify custom sorting criteria.

Example: Sorting a list of tuples based on the second element of each tuple.

# Original list of tuples

data = [('John', 25), ('Alice', 20), ('Bob', 30)]

# Sorting the list based on the second element of each tuple (age)

sorted\_data = sorted(data, key=lambda x: x[1])

print(sorted\_data) # Output: [('Alice', 20), ('John', 25), ('Bob', 30)]

In each example, the lambda function is used to define a custom operation (squaring, filtering, or sorting) that is applied to each element of the iterable. Lambda functions provide a convenient way to define these operations inline without the need for separate function definitions.

## Read data from the excel

Reading data from Excel files (.xlsx) into Pandas DataFrames is a common task. You can use the read\_excel() function provided by Pandas to accomplish this.

To load excel we have install openpyxl by using below command

> pip install openpyxl

Ex:

sheet1\_df = pd.read\_excel("dummy\_data.xlsx", sheet\_name='Sheet1')

excel\_sheet1\_df

sheet2\_df = pd.read\_excel("dummy\_data.xlsx", sheet\_name='Sheet2')

excel\_sheet2\_df

## Read data from the CSV

To read data from a CSV file into a Pandas DataFrame, you can use the read\_csv() function provided by Pandas.

Ex:

import pandas as pd

titanic\_df = pd.read\_csv("titanic.csv")

titanic\_df

#output

It display the titanic.csv file content in tabular form

# 07 April

# Have to go with recording till 10.40 AM

Open anaconda prompt

Run below command to activate intellipaat\_test environment

>conda activate intellipaat\_test

## CSV joins

In pandas, joining CSV files involves reading the CSV files into DataFrame objects and then using pandas' merge function to join them based on common columns. Here's a step-by-step guide on how to do this:

import pandas as pd

df1 = pd.read\_csv('file1.csv')

df2 = pd.read\_csv('file2.csv')

print(df1.head())

print(df2.head())

merged\_df = pd.merge(df1, df2, on='common\_column\_name', how='inner')

Here:

* on: specifies the common column name to join on.
* how: specifies the type of join. Options include **'inner'**, **'outer'**, **'left'**, and **'right'**. The **default** is 'inner'.

 **Save the merged DataFrame to a new CSV file if needed:**

merged\_df.to\_csv('merged\_file.csv', index=False)

This process assumes that both CSV files have a common column that you want to join on. If the column names are different but represent the same information, you can specify the left\_on and right\_on parameters instead of on. If the column names are different and you want to join on the indices of the dataframes, you can use left\_index=True and right\_index=True.

### Joins

Pandas supports several types of joins, which you can specify using the how parameter in the merge() function. Here are the different types of joins:

#### Inner Join (how='inner')

Returns only the rows where there is a match in both DataFrames based on the specified join columns.

pd.merge(df1, df2, on='common\_column\_name', how='inner')

#### Outer Join (how='outer')

Returns all rows from both DataFrames, with NaNs where there are no matches in the other DataFrame.

pd.merge(df1, df2, on='common\_column\_name', how='outer')

#### Left Join (how='left')

Returns all rows from the left DataFrame and matching rows from the right DataFrame. If there is no match, returns NaN.

pd.merge(df1, df2, on='common\_column\_name', how='left')

#### Right Join (how='right')

Returns all rows from the right DataFrame and matching rows from the left DataFrame. If there is no match, returns NaN.

pd.merge(df1, df2, on='common\_column\_name', how='right')

#### Cross Join (Cartesian Join)

Returns the Cartesian product of the rows from both DataFrames. It matches each row of the first DataFrame with every row of the second DataFrame.

pd.merge(df1, df2, how='cross')

These are the main types of joins supported by pandas. Depending on your use case and the data you're working with, you can choose the appropriate type of join to combine your DataFrames effectively.

## MatPlotLib

First we have to install matplotlib by using below command in anaconda console

> pip install matplotlib

### Description

Matplotlib is a comprehensive library in Python used for creating static, interactive, and animated visualizations. It provides a wide range of tools for creating plots, charts, and other graphical representations of data. Here's an explanation of some key concepts and components of Matplotlib:

1. **Figure:**
   * The entire area where your plot and any accompanying elements are drawn.
   * You can think of it as the canvas on which you're going to paint your visualization.
2. **Axes:**
   * The region of the figure where the data is plotted.
   * An individual plot is typically represented by a set of axes.
   * An Axes object contains various elements like x-axis, y-axis, ticks, labels, and the actual plotted data.
3. **Plot Types:**
   * Matplotlib supports various types of plots, including line plots, scatter plots, bar plots, histograms, pie charts, etc.
   * You can choose the appropriate plot type based on the nature of your data and the story you want to convey.
4. **Plotting Functions:**
   * Matplotlib provides a wide range of functions for creating different types of plots.
   * For example, plot() for line plots, scatter() for scatter plots, bar() for bar plots, hist() for histograms, etc.
5. **Customization:**
   * Matplotlib allows extensive customization of plots to meet specific requirements.
   * You can customize various aspects such as colors, line styles, markers, labels, titles, gridlines, legends, etc.
6. **Subplots:**
   * Subplots are multiple plots displayed within the same figure.
   * They allow you to visualize multiple datasets or different views of the same dataset side by side.
7. **Backend:**
   * Matplotlib supports different backends for rendering graphics, including interactive backends for use in GUI applications and non-interactive backends for generating images, PDFs, SVG files, etc.
8. **Integration:**
   * Matplotlib is highly compatible with other Python libraries like NumPy, Pandas, and SciPy, making it easy to visualize data from these libraries.
9. **Object-oriented Interface:**
   * Matplotlib can be used through both a state-based and an object-oriented interface.
   * The object-oriented approach provides more control and flexibility over the creation and customization of plots.
10. **Extensibility:**
    * Matplotlib is highly extensible, allowing users to create custom plots, styles, and functionalities beyond the built-in capabilities.

In summary, Matplotlib is a powerful and versatile library for data visualization in Python, providing a wide range of tools and capabilities to create publication-quality plots for various applications and domains.

Following are various graphs/plots available in matplotlib:

1. Line Plot
2. Scatter Plot
3. Bar Plot
4. Histogram
5. Pie Chart
6. Box Plot (Box-and-Whisker Plot)
7. Heatmap
8. Violin Plot
9. Area Plot
10. Contour Plot
11. Quiver Plot
12. Stem Plot
13. Polar Plot
14. 3D Plot
15. Error Bar Plot
16. Hexbin Plot
17. Streamplot
18. Spider (Radar) Plot
19. Sankey Diagram
20. Treemap

### Line Plot Graph

A line plot is a graph that displays data points connected by straight line segments. It's commonly used to visualize trends or patterns in data over time or across different variables.

Here's an example of how to create a simple line plot using Matplotlib:

import matplotlib.pyplot as plt

# Sample data

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

y = [2, 3, 5, 7, 11]

# Create a line plot

plt.plot(x, y)

# Add labels and title

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Example Line Plot')

# Display the plot

plt.show()

In this example:

* We import the matplotlib.pyplot module as plt.
* We define two lists x and y containing our sample data points.
* We use the plot() function to create the line plot by passing x and y as arguments.
* We add labels to the x-axis and y-axis using the xlabel() and ylabel() functions, respectively.
* We set the title of the plot using the title() function.
* Finally, we display the plot using the show() function.

When you run this code, it will display a simple line plot with the data points connected by straight lines.

You can customize the appearance of the plot by adding parameters to the plot() function, such as line color, line style, marker style, etc. For example:

plt.plot(x, y, color='red', linestyle='--', marker='o', markersize=8, label='Data Points')

This would create a line plot with red dashed lines, circular markers at data points, and a legend labeled as 'Data Points'.

Line plots are useful for visualizing continuous data and understanding trends or relationships between variables. They are straightforward to create and provide a quick way to explore and interpret your data.

### Bar Graph

### Histogram

### PieChart

Have to look for color attribute

### Scatter Plot

# 13th April

# 14th April

# 20th April

Only Hands on session on Pandas

# 21st April

Hands on Session on Visualization using the Seaborn(sns)

# 4 May

## Statastics / stats

It is a branch of mathematics that deals with collection of data, presentation of data for performing analysis on it to extract meaningful information.

Stats majorly classified into two categories:

1. **Descriptive statistics**
   * It is a branch of stats that deals with collection and interpretation of data mainly Mathematical operations.
2. **Inferential statistics**
   * It is a branch of stats that deal with predictions and drawing conclusion based on sample and populations.

### **Descriptive statistics**

It is a branch of stats that deals with collection and interpretation of data mainly Mathematical operations.

Again, it is classified into 2 parts:

1. **Measure of Central Tendancy -** It is used to find where does the centre of our data lies. It includes
   1. Mean
   2. Median
   3. Mode
2. **Measure of Dispersion** - It is used to find how much my data is scaterred or spread out in all the direction. It includes
   1. **Range**
   2. **Variance**
   3. **Standard Deviation**
   4. **Outliers**
   5. **Percentile**
   6. **Quartile**
   7. **Correlation**

#### Mean / Average

The mean, often referred to as the average, is a measure of central tendency that represents the typical value in a data set. It's calculated by adding up all the values in the data set and then dividing by the total number of values.

Mean = Sum of elements / number of elements

Ex:

if you have the data set [3, 7, 1, 5, 9], the mean is calculated as:

(3 + 7 + 1 + 5 + 9) / 5 = 5

So, the mean of this data set is 5.

#### Meadian

In statistics,

* the median is a measure of central tendency, which represents the middle value of a data set when it's arranged in ascending or descending order.
* If the data set has an odd number of observations, the median is simply the middle value.
* If the data set has an even number of observations, the median is the average of the two middle values.

Formula:

For even,

value of (n+1​)/2 observation

For odd,

(value of (n/2​) observation+value of ((n/2)​+1) observation​)/2

Ex:

* Odd number of values:
  + in the data set [3, 7, 1, 5, 9],
    - the median is 5 because it's the middle value when the data set is arranged in ascending order.
* Even Number of values:
  + In the data set [2, 4, 6, 8],
    - the median is (4 + 6) / 2 = 5 because there is an even number of observations.

#### Mode

The mode is a measure of central tendency that represents the value that appears **most** **frequently** in a dataset. Unlike the mean and median, which are based on numerical calculations, the mode is determined by observing the data directly.

A dataset can have:

1. **No mode:** If all values occur with the same frequency, or if no value repeats.
2. **Unimodal:** If there is only one mode, meaning one value occurs most frequently.
3. **Bimodal:** If there are two modes, meaning two values occur with the same highest frequency.
4. **Multimodal:** If there are three or more modes, meaning three or more values occur with the same highest frequency.

Ex:

In the dataset [3, 5, 5, 6, 8, 8, 8, 9], the mode is 8 because it appears more frequently than any other value.

The mode is especially useful for categorical data or when dealing with discrete values, but it can also be used with continuous data by creating intervals or categories. It's a simple and intuitive measure of central tendency, but it may not always fully represent the dataset, especially if there is no clear mode or if the dataset is skewed.

Note:

If we have same number of repetitions for two values then first occurrence among those two will be treated as mode. I.e., in the data[1, 5, 3, 4, 6, 4, 3], then mod will be 3 for this data as 3 occurred first in the list.

#### Range

The difference between min and max values of a dataset is known as range.

Ex:

Data = [25, 10, 20, 75, 40, 35, 50, 65]

Range = 75-10, range = 65

data = [1000, 990, 3000, 2500, 12000, 500, 100000]

range = np.ptp(data) # the ptp (Peek to Peek) is used to calculate the range in NumPy

print(f"The range of the data {data} is: {range}")

Output

The range of the data [1000, 990, 3000, 2500, 12000, 500, 100000] is: 99500

#### Variance

variance measures the dispersion or spread of a set of data points. It quantifies how much the data values differ from the mean.

* A high variance indicates that the data points are spread out over a wider range
* A low variance indicates that the data points are closer to the mean.

Mathematically, the variance σ2σ2 of a dataset x1,x2,…,xn ​with mean μ is calculated using the formula:

A black square and square symbols

Description automatically generated with medium confidence

Where:

* xi​ represents each individual data point.
* μ represents the mean of the dataset.
* n represents the total number of data points.

Variance is often used in conjunction with the standard deviation, which is the square root of the variance. The standard deviation provides a measure of the average deviation of the data points from the mean, in the same units as the data.

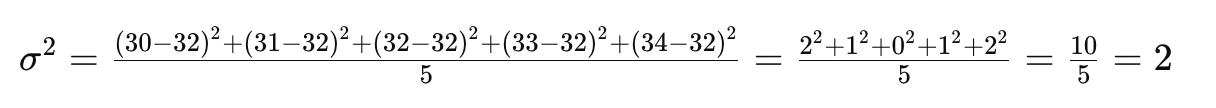
In practical terms, variance helps in understanding the distribution of data and assessing the degree of variability or dispersion within a dataset.

**Ex for Low Variance:**

Suppose we have a dataset representing the ages of a group of people in a small town:

[30,31,32,33,34]

The mean age is 32. If we calculate the variance using the formula, we find it's quite low. Each age is close to the mean, so there's little variability:



So, the variance is 2, indicating low dispersion.

data = [30,31,32,33,34]

variance\_value = np.var(data)

print(f"The varinace value for data {data} is: {variance\_value}")

Output:

The varinace value for data [30, 31, 32, 33, 34] is: 2.0

**Ex for High Variance:**

Now, let's consider a dataset representing the incomes of people in a city:

[20,30,40,70,100]

The mean income is 52. If we calculate the variance using the formula, we find it's higher than in the previous example. Incomes vary significantly from the mean:

So, the variance is 856, indicating high dispersion.

data = [20,30,40,70,100]

mean = stats.mean(data)

print(f"The mean value for the data {data} is: {mean}")

variance\_value = np.var(data)

print(f"The varinace value for data {data} is: {variance\_value}")

The output

The mean value for the data [20, 30, 40, 70, 100] is: 52

The varinace value for data [20, 30, 40, 70, 100] is: 856.0

#### Standard Deviation

In general, Stadard deviation is square root of variance.

The standard deviation is a measure of the amount of variation or dispersion in a set of data points. It quantifies the average distance of each data point from the mean of the dataset.

Mathematically, the standard deviation σ of a dataset x1,x2,…,xn with mean μ is calculated as the square root of the variance:

A black and white math equation

Description automatically generated with medium confidence

Where:

* xi represents each individual data point.
* μ represents the mean of the dataset.
* n represents the total number of data points.

Standard deviation provides a measure of the average deviation of the data points from the mean. A high standard deviation indicates that the data points are spread out over a wider range, while a low standard deviation indicates that the data points are closer to the mean.

Standard deviation is particularly useful because it's expressed in the same units as the original data, making it easier to interpret in practical terms. It's widely used in various fields of study, including science, engineering, finance, and social sciences, to understand the variability or dispersion within a dataset.

Ex:

data = [30,31,32,33,34]

stdDeviation = np.std(data)

print(f"The standard deviation of data {data} is: {stdDeviation}")

The output

The standard deviation of data [30, 31, 32, 33, 34] is: 1.4142135623730951

#### Percentile

a percentile is a measure used to indicate the value below which a given percentage of observations in a group of observations fall. It's a way of dividing a dataset into 100 equal parts, with each part representing 1% of the data.

To find a specific percentile, you first sort the data in ascending order, then find the position of the percentile within the dataset. For example, the 25th percentile is the value below which 25% of the data fall, the 50th percentile is the median (the middle value), and the 75th percentile is the value below which 75% of the data fall.

Percentiles are commonly used in various fields such as education, healthcare, finance, and environmental science to understand the distribution of data and identify specific points within a dataset. They provide a way to compare individual data points to the rest of the dataset and to interpret how values are distributed relative to each other.

Ex:

Suppose we have a class of 20 students, and their test scores are as follows:

65,70,72,75,78,80,81,82,83,85,87,88,90,92,94,95,97,98,99,10065,70,72,75,78,80,81,82,83,85,87,88,90,92,94,95,97,98,99,100

Now, let's find the percentile of a few specific values:

1. **25th Percentile (Q1):** The 25th percentile is the value below which 25% of the data fall. To find this, we need to locate the value at 25100×20=510025​×20=5th position in the sorted dataset.

Sorting the dataset: 65,70,72,75,78,80,81,82,83,85,87,88,90,92,94,95,97,98,99,100

So, the 25th percentile is 78.

1. **50th Percentile (Median):** The 50th percentile is the median, which is the value that separates the lower half from the upper half of the data. Since we have an even number of data points, we take the average of the two middle values.

Sorting the dataset: 65,70,72,75,78,80,81,82,83,85,87,88,90,92,94,95,97,98,99,100

So, the 50th percentile is (78+80)/2=79278+80​=79.

1. **75th Percentile (Q3):** The 75th percentile is the value below which 75% of the data fall. To find this, we need to locate the value at 75100×20=1510075​×20=15th position in the sorted dataset.

Sorting the dataset: 65,70,72,75,78,80,81,82,83,85,87,88,90,92,94‾,95,97,98,99,10065,70,72,75,78,80,81,82,83,85,87,88,90,92,94​,95,97,98,99,100

So, the 75th percentile is 94.

test\_scores = [65, 70, 72, 75, 78, 80, 81, 82, 83, 85, 87, 88, 90, 92, 94, 95, 97, 98, 99, 100]

# Calculate percentiles

percentiles = np.percentile(test\_scores, [25, 50, 75])

print("25th Percentile (Q1):", percentiles[0])

print("50th Percentile (Median):", percentiles[1])

print("75th Percentile (Q3):", percentiles[2])

The output

25th Percentile (Q1): 78.0

50th Percentile (Median): 83.5

75th Percentile (Q3): 92.0

Explanation:

* Here 25 percentile values are less than 78 values
* 50 percentile values are less than 83.5 values
* 80 percentile values are less than 92 values

##### Percentage vs percentile

**Percentage:**

* A percentage is a way to express a proportion or fraction as a number out of 100.
* It represents a part of the whole and is often used to describe relative quantities or proportions.
* For example, if you score 80 out of 100 on a test, you've achieved 80%, which means you've answered 80% of the questions correctly.

**Percentile:**

* A percentile is a specific type of percentage that indicates the relative standing of a particular value within a dataset.
* It divides a dataset into 100 equal parts, each containing 1% of the data.
* Percentiles are often used in statistics to understand the distribution of data and identify specific points within the dataset.
* For example, if you score at the 90th percentile on a standardized test, it means that your score is higher than 90% of the scores in the dataset.

Note:

One can achieve 100 percentage when he scored 100 out of 100 but he can’t get 100 percentile as he can’t cross every one in the class but not him self. So at max highest percentile can 99.99 but not 100

#### Quartile / quantile

It will divide your percentile in equal parts.

Quantiles divide a dataset into equally sized contiguous subgroups, where each subgroup contains an equal number of data points.

Commonly used quantiles include:

1. **Median (50th Quantile):** The median divides the dataset into two equal parts. 50% of the data points fall below the median, and 50% fall above.
2. **Quartiles (25th, 50th, and 75th Quantiles):** Quartiles divide the dataset into four equal parts. The first quartile (Q1) represents the 25th quantile, the second quartile (Q2) represents the 50th quantile (median), and the third quartile (Q3) represents the 75th quantile.
3. **Percentiles (Any Quantile):** Percentiles divide the dataset into 100 equal parts. For example, the 20th percentile represents the 20th quantile, dividing the dataset into 100 equal parts with 20% of the data points falling below and 80% falling above.

Quantiles are used to understand the distribution of data and to compare specific data points or subsets of data to the rest of the dataset. They are particularly useful for analyzing large datasets and identifying key summary statistics.

# 5th May

#### Outliers

Outliers are unwanted data.

The outliers are data points that significantly differ from other observations in a dataset. They are typically much higher or lower than the rest of the data points and can skew statistical analyses and interpretations.

Outliers can occur due to various reasons, such as:

* Measurement or recording errors
* Natural variability in the data
* Experimental or sampling errors
* Anomalies in the data-generating process
* Data entry mistakes

There are various methods to detect and handle outliers, including:

* Visual inspection using scatter plots, box plots, or histograms
* Statistical methods such as the z-score, which measures how many standard deviations a data point is from the mean
* Robust statistical techniques that are less sensitive to outliers, such as median-based measures

Once outliers are identified, they can be treated in different ways, including:

* Removing them from the dataset if they are determined to be errors
* Transforming the data to make it more normally distributed
* Using robust statistical methods that are less affected by outliers

##### Side Effects

**Skewing measures of central tendency:** Outliers can significantly affect measures of central tendency, such as the mean. Since the mean is sensitive to extreme values, outliers can pull the mean towards them, giving a distorted view of the typical value in the dataset.

1. **Inflating measures of dispersion:** Outliers can increase the variability of the dataset, leading to inflated measures of dispersion, such as the standard deviation or variance. This can make the spread of the data appear larger than it actually is.
2. **Distorting relationships:** In regression analysis and other modeling techniques, outliers can distort the relationships between variables. They can influence the slope and intercept of regression lines, leading to biased estimates and inaccurate predictions.
3. **Reducing model performance:** Outliers can negatively impact the performance of predictive models by introducing noise and reducing the model's ability to generalize to new data. This can lead to overfitting, where the model learns the noise in the data rather than the underlying patterns.
4. **Misleading interpretations:** Outliers can lead to incorrect interpretations of data and relationships. Analysts may draw incorrect conclusions or make faulty decisions if outliers are not properly identified and addressed.
5. **Violating assumptions:** Outliers can violate the assumptions of many statistical tests and models, such as the assumption of normality or homoscedasticity. This can invalidate the results of analyses and undermine the validity of conclusions drawn from the data.

##### Identify Outliers

1. **Visual inspection:** Plotting the data using scatter plots, box plots, histograms, or QQ plots can help identify outliers visually. Outliers will appear as points that are significantly distant from the bulk of the data.
2. **Summary statistics:** Calculating summary statistics such as the mean, median, standard deviation, and quartiles can help identify potential outliers. Observations that fall far from the mean or median, or are outside the range defined by the quartiles, may be considered outliers.
3. **Z-score:** The z-score measures how many standard deviations a data point is from the mean. Data points with a z-score greater than a certain threshold (e.g., 2 or 3) are considered outliers. The formula for calculating the z-score is: z=x−μσz=σx−μ​, where xx is the data point, μμ is the mean, and σσ is the standard deviation.
4. **Interquartile Range (IQR):** The IQR is the range between the first quartile (Q1) and the third quartile (Q3) of the data. Data points outside the range will be:



considered outliers.

1. **Box plots:** Box plots visually display the quartiles and any potential outliers in the dataset. Points beyond the "whiskers" of the box plot are typically considered outliers.
2. **Statistical tests:** There are various statistical tests for identifying outliers, such as Grubbs' test, Dixon's Q test, and Tukey's range test. These tests compare the data points to a statistical model and determine whether they are significantly different from the rest of the data.
3. **Domain knowledge:** Finally, domain knowledge and understanding of the context in which the data was collected can help identify outliers. Some values that may appear as outliers in one context may actually be valid data points in another context.

Following is the code snippet to find the outliers

data = [5.5, 5.6, 6.1, 6.2, 5.9, 5.11, 6.5, 50.5, 0.5, 60.1]

q1 = np.percentile(data, 25)

q3 = np.percentile(data, 75)

IQR = q3-q1

upper\_fence = q3 + (1.5\*IQR)

lower\_fence = q1 - (1.5\*IQR)

print(f"The Lower fence for the data {data} is: {lower\_fence}")

print(f"The Upper fence for the data {data} is: {upper\_fence}")

for dataPoint in data:

if dataPoint > upper\_fence or dataPoint < lower\_fence:

print(f"Outliers data points are: {dataPoint}")

#The output

The Lower fence for the data [5.5, 5.6, 6.1, 6.2, 5.9, 5.11, 6.5, 50.5, 0.5, 60.1] is: 4.175000000000001

The Upper fence for the data [5.5, 5.6, 6.1, 6.2, 5.9, 5.11, 6.5, 50.5, 0.5, 60.1] is: 7.774999999999999

Outliers data points are: 50.5

Outliers data points are: 0.5

Outliers data points are: 60.1

#### Correlation

The relationship between two columns and two values in a data.

Correlation is a statistical measure that describes the extent to which two variables change together. In other words, it quantifies the strength and direction of the linear relationship between two variables.

The correlation coefficient, often denoted by r, ranges from -1 to 1:

* r=1: Perfect positive correlation. This means that as one variable increases, the other variable also increases, and vice versa, following a linear pattern.
* r=−1r=−1: Perfect negative correlation. This means that as one variable increases, the other variable decreases, and vice versa, following a linear pattern.
* r=0r=0: No correlation. This means that there is no linear relationship between the two variables.

### Inferential statistics

Inferential statistics is a branch of statistics that involves making inferences or predictions about a population based on data collected from a sample of that population. It allows us to draw conclusions, make predictions, and test hypotheses about populations using sample data.

It have below categories:

1. **Hypothesis testing:** Testing hypotheses about population parameters to determine if there is enough evidence to support or reject a claim.
   1. **Null Hypothesis:** It is a random guess without using any data. Means data is non stationary
   2. **Alternative Hypothesis:** it is a guess after analysing previous data means data is stationary.
2. **Regression Analysis:** Examining the relationship between one or more independent variables and a dependent variable to make predictions or identify associations.
   1. **Independent variables**

also known as predictor variables or explanatory variables, are the variables that are used to predict or explain the values of the dependent variable.

* 1. **Dependent variables**

also known as the response variable or outcome variable, is the variable that is being predicted or explained by the independent variables. The role of the dependent variable is central to regression analysis as it is the focus of the analysis and the variable of interest.

Note:

Independent variables are used to predict the value of dependent variables. i.e., independent variables are fundamental components of regression analysis, as they are used to build models that explain and predict the values of the dependent variable based on the values of the independent variables.

# 11th May

Have to go again these recordings as we missed this in MacBook

# 12th May

Have to go again these recordings as we missed this in MacBook

# 18th May

Have to go again these recordings as we missed this in MacBook

# 19th May

Have to go again these recordings as we missed this in MacBook