ALGORITHMIC THINKING WITH PYTHON

- Course Code: UCEST105
- Course Type :Theory
- Teaching Hours/Week (L: T:P: R) 3:0:2:0
- Credits 4
- ESE Marks 60
- CIE Marks 40

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux,etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

- **Python was designed for readability, and has some similarities to the English language with influence from mathematics.**
- **Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.**
- **Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.**

Python Install

- Download the latest release of Python from www.python.org depending on your OS Windows, Linux or Mac.
-

  Complete the installation.This will also install IDLE(Integrated Development Environment).
  Visit https://docs.python.org/3/library/idle.html to get complete details of IDLE.

How Python works

- 1.Open IDLE by clicking the application icon
  2.Open File menu and click New File and type your first script (Eg: print("welcome to Python")
  3.Save your file with .py extension (Eg:test.py)
  4.From the **Run menu** click the **Run Module(** or press F5-short cut).This will run the script

  5.From the **File** menu choose **Exit** to quit from IDLE

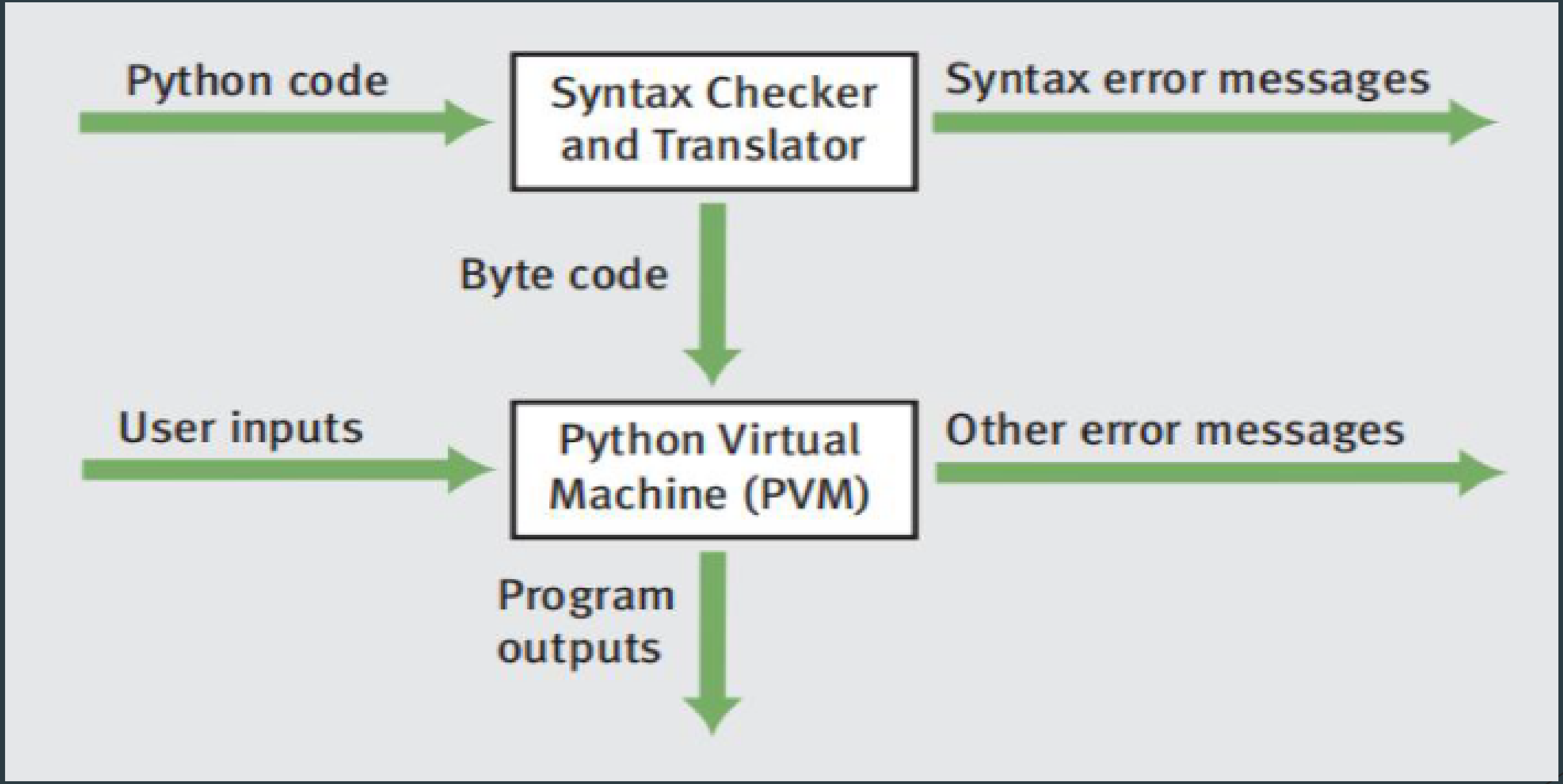
How to Run Python Code Interactively

- A widely used way to run Python code is through an interactive session. To start a Python interactive session, just open a command-line or terminal and then type in python, or python3 depending on your Python installation, and then hit Enter.
-

  Here's an example of how to do this on Linux:
- Open Terminal (Control-Alt-T)
  $ python3
- Python 3.6.7 (default, Oct 22 2018, 11:32:17) [GCC 8.2.0] on linux Type "help", "copyright", "credits" or "license" for more information.
- >>>The standard prompt for the interactive mode is >>>, so as soon as you see these characters, you'll know you are in.
- Now you can test your commands here interactively
- **>>>print("Welcome to Python")**
- Welcome to Python
- **>>>2+3**
- 5

- On Windows, the command-line is usually known as command prompt or MS-DOS console, and it is a program called cmd.exe.
- type cmd to get the command prompt then type python
- c:\>python (need PATH set to python.exe directory)
- or type python in the search program .This will open python prompt >>>.
- You can run the script previously created(test.py) by importing it in the python command line
- **>>>import test.py**

Behind the Scenes: How Python Works



Steps in interpreting a Python program

- The interpreter reads a Python expression or statement, also called the source code, and verifies that it is well formed.
- If a Python expression is well formed, the interpreter then translates it to an equivalent form in a low-level language called byte code.
- This byte code is next sent to another software component, called the Python virtual machine (PVM), where it is executed. If another error occurs during this step, execution also halts with an error message.

Jupyter

- Jupyter Notebook is used to create interactive notebook documents that can contain live code, equations, visualizations, media and other computational outputs.
- Jupyter Notebook is often used by programmers, data scientists and students to document and demonstrate coding workflows or simply experiment with code.

Jupyter Notebook is great for the following use cases:

- **Learn and try out Python**
- **Data processing / transformation**
- **Numeric simulation**
- **Statistical modeling**
- **Machine learning**
- **Jupyter Notebook is perfect for using Python for scientific computing and data analysis with libraries like numpy, pandas, and matplotlib.**

Setting Up Jupyter Notebook

- The first step to get started is to visit the project's website at http://www.jupyter.org:

Here you'll find two options:

- Try it in your browser
- Install the Notebook
- With the first option Try it in your browser you can access a hosted version of Jupyter Notebook. This will get you direct access without needing to install it on your computer.

- The second option Install the Notebook will take you to another page which gives you detailed instruction for the installation. There are two different ways:

- Installing Jupyter Notebook by using the Python's package manager pip(Preferred Installer Program)
- Installing Jupyter Notebook by installing the Anaconda distribution

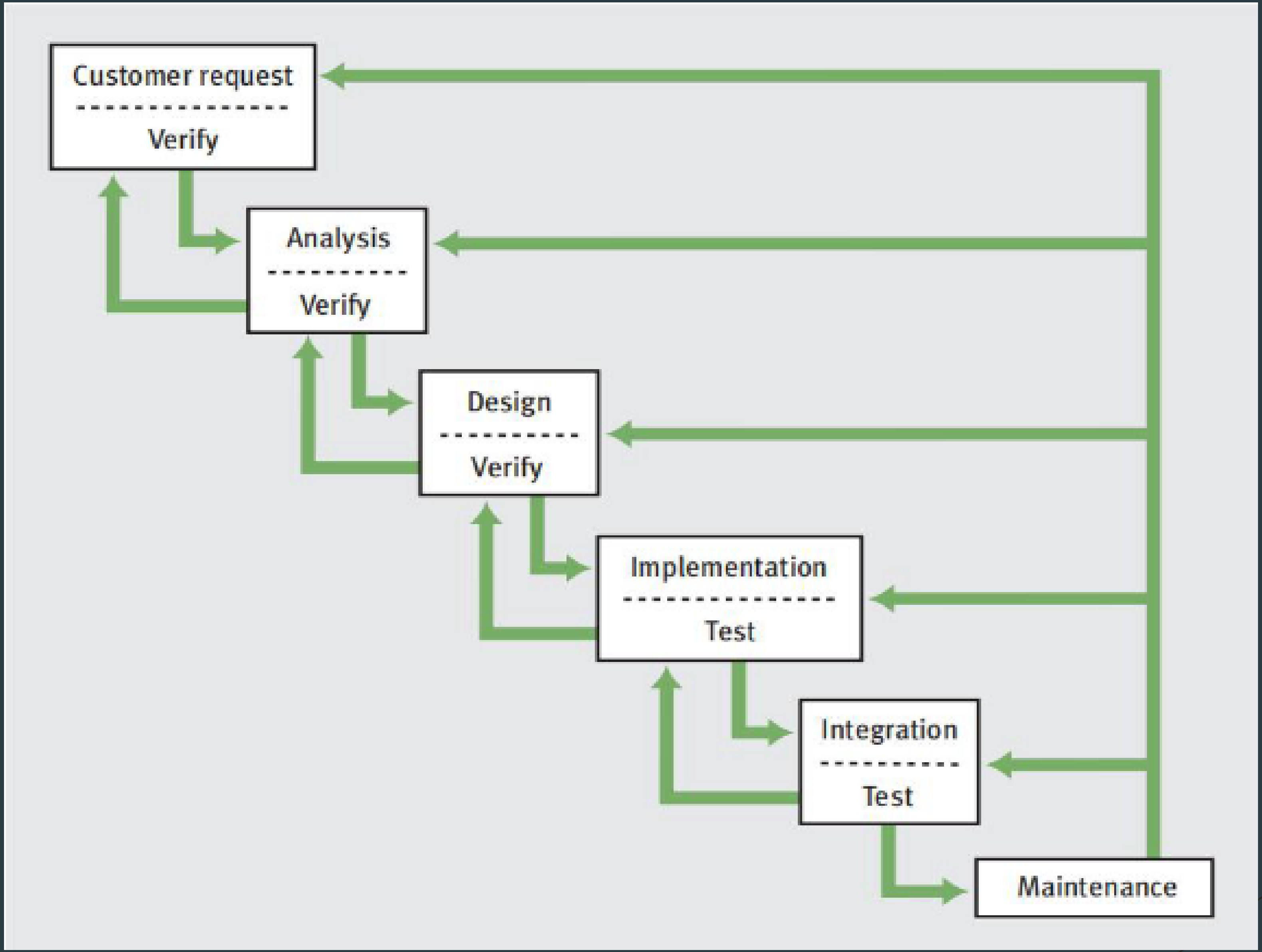
- The notebook itself consists of cells. A first empty cell is already available after having created the new notebook:
- This cell is of type “Code” and you can start typing in Python code directly. Executing code in this cell can be done by either clicking on the run cell button .The resulting output becomes visible right underneath the cell.
- A cell is active two modes distinguished:
  edit mode
  command mode
- If you just click in one cell the cell is opened in command mode which is indicated by a blue border on the left.
- The edit mode is entered if you click into the code area of that cell. This mode is indicated by a green border on the left side of the cell.

- Another cool function of Jupyter Notebook is the ability to create checkpoint. By creating a checkpoint you’re storing the current state of the notebook so that you can later on go back to this checkpoint and revert changes which have been made to the notebook in the meantime.

The software development process - Case Study.

- **Waterfall Model - Different Phases**
- **1. Customer request—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the user requirements phase.**
- **2. Analysis—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.**
- **3. Design—The programmers determine how the program will do its task.**

- 4) Implementation—The programmers write the program. This step is also called the coding phase.
- 5) Integration—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
- 6) Maintenance—Programs usually have a long life; a lifespan of 5 to15 years is common for software. During this time, requirements change, errors are detected, and minor or major modifications are made


Customer request
Verify
Analysis
Verify
Design
Verify
Implementation
Test
Integration
Test
Maintenance

Module-I

- PROBLEM-SOLVING STRATEGIES:- Problem-solving strategies defined, Importance of understanding multiple problem-solving strategies, Trial and Error, Heuristics, Means-Ends Analysis, and Backtracking (Working backward).

- THE PROBLEM-SOLVING PROCESS:- Computer as a model of computation, Understanding the problem, Formulating a model, Developing an algorithm, Writing the program, Testing the program, and Evaluating the solution.

- ESSENTIALS OF PYTHON PROGRAMMING:- Creating and using variables in Python, Numeric and String data types in Python, Using the math module, Using the Python Standard Library for handling basic I/O - print,input, Python operators and their precedence.

PROBLEM-SOLVING STRATEGIES

- **Well-Defined Problems**
- These problems have clear goals, a defined path to a solution, and specific criteria for determining when the problem is solved. Let's explore three examples:
-

  Mathematical Equation:
  - **Problem:** Solve the equation 2x+3=7.
  - **Why it's Well-Defined:** The problem has a clear goal (find the value of
  - x), a specific method (solve for xx using algebraic rules), and a definite solution (x=2).
- **Crossword Puzzle:**
  - **Problem:** Complete a standard crossword puzzle.
  - **Why it's Well-Defined:** The crossword has a clear objective (fill in all the squares with the correct words), a specific set of rules, and a single correct solution.
- **Recipe Execution:**
  - **Problem:** Bake a chocolate cake using a provided recipe.
  - **Why it's Well-Defined:** The goal (bake a chocolate cake) is clear, the process is outlined step-by-step in the recipe, and success is measurable by the outcome (a baked cake that meets the description).

Ill-Defined Problems

- These problems are ambiguous, lack clear criteria for solutions, and often have multiple possible solutions. Here are three examples:

- Designing a Sustainable City:
  - **Problem:** How can we design a sustainable city for the future?
  - **Why it’s Ill-Defined:** The problem is broad, with no single clear solution. Various factors like environmental impact, social equity, and economic viability come into play, and different stakeholders might have different views on what constitutes a "sustainable" city.

- Writing a Novel:
  - **Problem:** Write a compelling novel that appeals to a broad audience.
  - **Why it’s Ill-Defined:** The goal is subjective (what is "compelling"?), the process can vary widely, and success is difficult to measure. Different readers may have different interpretations of what makes the novel appealing.

Resolving Workplace Conflict:

- **Problem:** Resolve a conflict between two team members in the workplace.
- **Why it’s Ill-Defined:** The problem is complex with no clear solution. It involves interpersonal dynamics, emotions, and communication styles, and what works in one situation might not work in another. Multiple solutions may exist, and the "right" one depends on various factors.

Summary

- **Well-Defined Problems** are structured, with a clear path and solution.
- **Ill-Defined Problems** are open-ended, ambiguous, and often subjective.
- Understanding the nature of the problem helps in choosing the right approach to solving it.

Definition of Problem-Solving Strategies:

- **Problem-solving strategies** are systematic methods or approaches used to find solutions to challenges, issues, or complex situations. These strategies guide the process of identifying the problem, exploring possible solutions, and implementing the best course of action.
- Here are some common problem-solving strategies:
- **Trial and Error:**
  - **Description:** This strategy involves trying different solutions and learning from mistakes until the correct one is found. It's practical when you have no clear path to the solution and need to explore various options.
- **Heuristics:**
  - **Description:** Heuristics are mental shortcuts or "rules of thumb" that simplify decision-making and problem-solving. These strategies aim for quick, practical solutions that are "good enough" rather than perfect.
- **Means-End Analysis:**
  - **Description:** This strategy involves breaking down a problem into smaller, manageable parts (means) and addressing each part to reduce the difference between the current state and the goal state (end). It's useful for complex problems with clear goals.
- **Backtracking:**
  - **Description:** Backtracking is used when solving problems that involve exploring multiple options. If a chosen path doesn't work, you backtrack to the previous step and try a different approach, ensuring that all possibilities are considered.

- **Divide and Conquer:**
  - **Description**: This strategy involves breaking a large problem into smaller, more manageable sub-problems. Each sub-problem is solved individually, and their solutions are combined to address the larger problem.
- **Algorithmic Approach:**
  - **Description**: An algorithmic approach uses a step-by-step procedure or formula to solve a problem. It’s precise and methodical, ensuring that every step is logically correct, commonly used in mathematics and computer science.
- **Brainstorming:**
  - **Description**: Brainstorming involves generating as many ideas as possible without immediately evaluating them. It encourages creative thinking and can lead to innovative solutions when dealing with complex or open-ended problems.
- **Analogy:**
  - **Description**: This strategy involves solving a problem by finding similarities between the current problem and a previously solved problem. The solution to the old problem is then adapted to solve the new one.

Importance of Problem-Solving Strategies:

- **Systematic Approach:** They provide a structured way to approach challenges, making it easier to find effective solutions.

- **Efficiency:** By using an appropriate strategy, you can solve problems more quickly and with less effort.

- **Adaptability:** Different strategies can be applied to different types of problems, making them versatile tools in various situations.

Heuristic Method

- The **Heuristic Method** is a problem-solving strategy that uses practical shortcuts or "rules of thumb" to make decisions and find solutions more quickly and efficiently, even if the solution isn’t perfect. Here’s how it works:
- **What is the Heuristic Method?**
- **Definition:** A heuristic is a mental shortcut that allows people to solve problems and make judgments quickly and efficiently. It’s a way to simplify complex problems by focusing on the most important factors or by applying general principles that usually lead to a good enough solution.
- **Purpose:** Heuristics are used when you need a fast, efficient solution and are willing to accept that it may not be the absolute best or most accurate one.

Characteristics of Heuristics:

- **Simplification:** Heuristics simplify complex problems, making them easier to tackle.
- **Speed:** They provide quicker solutions by cutting down on the time and effort needed.
- **Satisfactory Results:** Heuristics aim for a "good enough" solution rather than a perfect one, which is often sufficient in practical situations.

Steps in Using Heuristics:

- **Recognize the Problem:**
  - Understand the problem and what you need to achieve. Identify if a heuristic approach is appropriate.
- **Choose an Appropriate Heuristic:**
  - Select a rule of thumb or shortcut that applies to the situation. Consider which heuristic might lead to a quick and reasonable solution.
- **Apply the Heuristic:**
  - Use the chosen heuristic to guide your decision-making or problem-solving process. This might involve making an educated guess, simplifying the problem, or using past experiences.
- **Evaluate the Outcome:**
  - Assess whether the heuristic led to a satisfactory solution. If it did, great! If not, you might need to try a different heuristic or combine several to get a better result.

When to Use Heuristics:

- **Everyday Decisions:** When you need quick answers to routine problems.
- **Complex Problems:** When a perfect solution is impractical, and you need a workable answer fast.
- **Uncertain Situations:** When there’s limited information or too many variables to consider, and a best guess is your best bet.

When Not to Use Heuristics:

- **Critical Decisions:** In situations where accuracy is crucial, and the consequences of a mistake are severe, you might need a more thorough and analytical approach.
- **Highly Detailed Problems:** When every detail matters and a precise solution is required, heuristics may oversimplify the issue.

Use Cases:

- **Shopping:** Choosing a product based on brand reputation rather than comparing every option.
- **Navigation:** Following the general direction toward your destination instead of using a detailed map.
- **Medical Diagnosis:** Doctors using common symptoms to make an initial diagnosis before conducting detailed tests.
- **Stock Investment:** Investors using past trends to make decisions on buying or selling stocks.
- **Job Interviews:** Hiring managers quickly assessing candidates based on first impressions or resumes.

1. Rule of Thumb:

- **Example**: "Measure twice, cut once" is a rule of thumb used in carpentry to avoid mistakes. Instead of precisely calculating every measurement, this heuristic encourages careful checking to prevent errors.

2. Availability Heuristic:

- **Example**: If you're deciding whether to buy travel insurance, you might rely on recent news about airplane crashes. Because these events are vivid and easily remembered, you might overestimate the risk of flying and decide to purchase the insurance.

3. Working Backwards:

- **Example**: If you're trying to solve a maze, you might start at the end and work your way backward to the start. This can often reveal the correct path more easily than starting from the beginning.

4. Anchoring Heuristic:

- **Example**: When negotiating the price of a car, the initial price offered (anchor) often strongly influences the final agreed price. Even if the initial price is arbitrary, it sets a mental benchmark that both parties adjust from.

5. Simplification:

- **Example**: When trying to estimate the total cost of groceries while shopping, you might round the prices of items to the nearest dollar. This simplifies the mental math and gives you a rough estimate without needing to add every cent.

6. Elimination by Aspects:

- **Example**: When choosing a new laptop, you might first eliminate all options that don’t meet your minimum requirements for battery life. Then, among the remaining options, you eliminate those that don’t have the desired screen size. You continue this process until one option remains.

7. Representativeness Heuristic:

- **Example**: If someone loves books and quiet environments, you might assume they are a librarian because they match the stereotype, even if statistically, they are more likely to have a different occupation.

8. Satisficing:

- **Example**: When looking for a place to eat dinner, instead of searching for the perfect restaurant, you might choose the first one that looks "good enough" to satisfy your hunger.

9. Guesstimation:

- **Example**: If you need to estimate the number of jellybeans in a jar, you might make a rough guess by estimating the number of beans along one dimension and then multiplying it by the other dimensions.

10. Familiarity Heuristic:

- **Example**: When faced with a choice between two brands of a product, you might choose the one you recognize, even if you know nothing about the quality or price, simply because it feels safer or more familiar.

Trial and Error

- The **Trial and Error** method is a straightforward and practical problem-solving strategy where you try different solutions and learn from mistakes until you find one that works. Let’s break it down:
- **What is the Trial and Error Method?**
- **Definition:** Trial and Error involves experimenting with various approaches or solutions until you achieve the desired result. If one attempt doesn’t work, you try another, learning from each failure or success along the way.
- **Purpose:** This method is used when you don’t have a clear solution in mind and need to explore different possibilities.

Steps in the Trial and Error Method:

- **Identify the Problem:**
  - Clearly understand what you’re trying to solve. Define the problem and your goal.
- **Try a Possible Solution:**
  - Start with one approach or solution. This could be based on your intuition, past experience, or simply a guess.
- **Test the Solution:**
  - Implement the solution and observe what happens. Does it solve the problem?
- **Analyze the Results:**
  - If the solution works, great! If not, analyze why it didn’t work. What went wrong? What did you learn from this attempt?
- **Repeat the Process:**
  - Based on what you’ve learned, try another solution. Keep experimenting with different approaches until you find one that works.

When to Use the Trial and Error Method:

- **Unfamiliar Problems:** When you're dealing with a problem you've never encountered before and don't know the exact solution.
- **Limited Information:** When there isn't enough information available to make an informed decision.
- **Creative Problem-Solving:** When you want to explore different possibilities and don't mind trying out several options.

When Not to Use It:

- **Time-Sensitive Problems:** If you need a quick solution, Trial and Error might be too slow, as it can take many attempts to find the right answer.
- **Complex Problems:** For very complex problems, other methods (like systematic analysis or expert consultation) might be more efficient.

Use Cases:

- **Puzzle Games:** Solving a jigsaw puzzle by trying different pieces until they fit.
- **Programming Debugging:** Testing various code modifications to fix a bug when the exact cause is unclear.
- **Chemical Experiments:** Mixing different chemicals in varying proportions to achieve a desired reaction in a lab setting.
- **Lock Combinations:** Trying different combinations of numbers to unlock a padlock when you’ve forgotten the code.
- **Home Repairs:** Experimenting with different tools or methods to fix a leaky faucet.

Trial and Error- Examples

1. Learning to Ride a Bicycle:

- **Example**: When learning to ride a bike, you might fall several times before figuring out how to balance and steer properly. Each attempt helps you learn what works and what doesn’t, leading to success.

2. Solving a Puzzle:

- **Example**: If you’re putting together a jigsaw puzzle, you might try several pieces in a spot before finding the one that fits. You test each piece until you find the correct one.

3. Cooking Without a Recipe:

- **Example**: If you're cooking a dish without a recipe, you might experiment with adding different amounts of spices or ingredients until you achieve the desired taste. You keep adjusting based on the results of each attempt.

4. Finding the Right Tool:

- **Example**: When assembling furniture, you might try several tools or screws to see which one fits properly. You continue testing until you find the one that works.

5. Solving a Mathematical Problem:

- **Example**: In math, if you’re unsure how to solve an equation, you might try different methods—like substitution, elimination, or graphing—until you find the one that gives you the correct solution.

6. Learning to Play a Musical Instrument:

- **Example**: When learning a new instrument, you might try different finger placements or techniques until you find the one that produces the right note or sound.

7. Figuring Out a Password:

- **Example**: If you forget a password, you might try several different combinations that you think might be correct until you successfully log in.

8. Finding the Best Route:

- **Example**: If you're driving to a new location, you might take different routes on different days until you find the fastest or most convenient one.

9. Fixing a Broken Appliance:

- **Example**: If an appliance isn't working, you might try various fixes, such as checking the power source, resetting the device, or replacing parts, until it starts working again.

10. Testing Software or Code:

- **Example**: When writing code, you might try several approaches to debug or solve an issue. You modify the code, run it, and see if it works, repeating the process until the error is fixed.

Backtracking

- The **Backtracking** method is a problem-solving strategy that involves exploring possible solutions to a problem by building them incrementally, step by step. If you reach a point where the current path doesn't lead to a solution, you backtrack—go back to the previous step—and try a different path. It's particularly useful for solving problems with multiple possible solutions or where the solution involves making a sequence of decisions.
- **What is the Backtracking Method?**
- **Definition:** Backtracking is a method where you try to solve a problem by exploring all possible options. If you find that a certain option doesn't lead to a valid solution, you undo (or "backtrack") that choice and try the next option.
- **Purpose:** The goal is to find the correct solution by systematically exploring all possibilities, while discarding paths that don't work.

Steps in the Backtracking Method:

- **Identify the Problem and Constraints:**
  - Clearly understand the problem and any constraints (rules or limitations) that need to be followed.
- **Start with an Initial Decision:**
  - Begin by making an initial choice or taking the first step in your solution process.
- **Explore Further:**
  - Move forward by making the next decision or taking the next step. Continue to build your solution incrementally.
- **Check for Validity:**
  - After each step, check if the current path is valid and satisfies the problem's constraints. If it's invalid, you need to backtrack.
- **Backtrack if Necessary:**
  - If you reach a point where the current path doesn't work, undo the last step (backtrack) and try a different option. This might involve going back multiple steps until you find a valid path.
- **Continue Until Solution is Found:**
  - Repeat the process of exploring and backtracking until you either find a solution that meets all the criteria or determine that no solution exists within the given constraints.

When to Use the Backtracking Method:

- **Puzzles and Games:** In puzzles like Sudoku, N-Queens, or crosswords where you need to place elements in a grid under certain constraints.
- **Search Problems:** When searching for a specific arrangement, combination, or sequence that meets all criteria.
- **Decision-Making:** In scenarios where each decision builds on the previous one, and you need to ensure that each step is valid before proceeding.

When Not to Use Backtracking:

- **Simple Problems:** If the problem has a straightforward solution without multiple paths, backtracking might be unnecessary and overcomplicated.
- **Time-Sensitive Situations:** Backtracking can be time-consuming, especially if there are many possible paths to explore. In time-sensitive situations, a more direct approach might be better.

- **Use Cases:**
- **Maze Solving:** Navigating through a maze by choosing paths, and backtracking if a path leads to a dead end.
- **Sudoku:** Filling in numbers, and backtracking when a mistake is realized.
- **Recursion in Programming:** Writing algorithms that solve problems by breaking them into smaller subproblems and backtracking when necessary.
- **Puzzles like N-Queens:** Placing queens on a chessboard such that no two queens attack each other, and backtracking when a conflict is detected.
- **Travel Planning:** Planning a multi-city trip, backtracking to change a route if it's not feasible or too expensive.

Backtracking-Examples

1. **Solving a Maze:**

- **Example**: Imagine you're trying to navigate a maze. You start at the entrance and choose a path. If you reach a dead end or find that your current path leads you in circles, you backtrack to the last junction and try a different direction until you find the exit.

2. **N-Queens Problem:**

- **Example**: The N-Queens problem involves placing N queens on an N x N chessboard so that no two queens threaten each other. You place a queen in a row, and if you find that placing more queens leads to a conflict, you backtrack by removing the last placed queen and trying a new position for it.

3. **Sudoku Puzzle:**

- **Example**: In a Sudoku puzzle, you fill in numbers in a grid according to specific rules. If placing a number in a cell violates the Sudoku rules, you backtrack by erasing that number and trying different numbers until the puzzle is solved.

4. **Generating Permutations:**

- **Example**: To generate all possible permutations of a set of elements, you start by fixing one element and recursively permute the remaining elements. If a permutation doesn’t meet the criteria, you backtrack by swapping elements back to their original positions and continuing with the next permutation.

5. **Crossword Puzzle:**

- **Example**: When filling in a crossword puzzle, you might try different words for a given clue. If a chosen word doesn’t fit with the intersecting words, you backtrack by changing the word and trying different options until the puzzle is correctly filled.

6. Subset Sum Problem:

- **Example**: Given a set of numbers, you want to find a subset that sums up to a specific value. You start by including some numbers in the subset and check if they sum to the target value. If not, you backtrack by removing numbers and trying different combinations.

7. Traveling Salesman Problem (TSP):

- **Example**: In the TSP, you need to find the shortest route that visits each city exactly once and returns to the starting city. You explore different routes and if a route doesn't meet the criteria for being the shortest, you backtrack and try a different path.

8. Combination Sum Problem:

- **Example**: Given a set of numbers and a target sum, you want to find all possible combinations that add up to the target. You start by including a number and then recursively explore further combinations. If a combination exceeds the target, you backtrack by removing the number and trying other combinations.

9. Word Search Puzzles:

- **Example**: In a word search puzzle, you try to find words in a grid. If you start a path for a word and find that it doesn't match the word, you backtrack by reversing your steps and trying different directions or starting points.

10. Solving Cryptographic Puzzles:

- **Example**: In cryptographic puzzles or ciphers, you might try different key values or decryption methods. If the decrypted text doesn't make sense, you backtrack by trying a different key or method until you find the correct solution.

Means-End Analysis

- Means-End Analysis is a problem-solving strategy where you break down the problem into a series of steps (means) to reach a desired outcome (end). Here's how it works:
- **What is Means-End Analysis?**
- Means-End Analysis involves identifying the current state (where you are now) and the goal state (where you want to be). The key idea is to reduce the difference between these two states by applying specific actions or steps (means) that bring you closer to the goal.

Steps in Means-End Analysis:

- **Identify the Goal (End):**
  - Clearly define what you want to achieve. This is the end state you are aiming for.
- **Assess the Current State:**
  - Understand your current position or condition relative to the goal. Identify the gap or difference between where you are and where you want to be.
- **Identify the Differences:**
  - Determine the differences between the current state and the goal state. These differences highlight what needs to be changed or achieved to reach the goal.
- **Select and Apply an Action (Means):**
  - Choose an action or step that will reduce the difference between the current state and the goal state. This action is a means to bring you closer to your goal.
- **Evaluate and Repeat:**
  - After applying the action, reassess the new current state. If the goal hasn’t been fully achieved, repeat the process by identifying the next difference and applying another action until the goal is reached.

Means-End Analysis- Examples

1. Planning a Vacation:

- **Example**:
  - **Goal**: Plan a vacation to a tropical island.
  - **Means**:
    - **Determine Budget**: Calculate how much you can spend on the trip.
    - **Choose Destination**: Research and select a suitable tropical island within your budget.
    - **Book Flights**: Find and book flights to the chosen destination.
    - **Book Accommodation**: Reserve a hotel or rental property on the island.
    - **Plan Activities**: Decide on activities and excursions to do during the trip.

2. Building a House:

- **Example**:
  - **Goal**: Build a new house.
  - **Means**:
    - **Design House**: Create architectural plans and design the layout.
    - **Obtain Permits**: Get necessary building permits and approvals.
    - **Prepare Site**: Clear and prepare the construction site.
    - **Build Foundation**: Lay the foundation and build the structural framework.
    - **Construct Interiors**: Complete the interior work, including plumbing, electrical, and finishing touches.

- Use Cases:
- **Project Management:** Breaking down a large project into smaller tasks and setting milestones to reach the final goal.
- **Programming Algorithms:** Developing an algorithm by identifying the final output and working backward to define the necessary steps.
- **Medical Treatment Plans:** Doctors setting treatment milestones (e.g., reducing pain, improving mobility) to achieve the final goal of full recovery.
- **Chess Strategy:** Players determining the end goal (checkmate) and working backward to create a sequence of moves to reach it.
- **Educational Goals:** Students setting long-term goals (e.g., graduating) and breaking them down into semester-wise targets and daily study plans.

Comparative Summary

- **Trial and Error:**
  - **Efficiency:** Often inefficient due to the potentially large number of steps.
  - **Best for:** Simple problems or when no clear solution path is available.
- **Heuristics:**
  - **Efficiency:** More efficient than Trial and Error but less reliable.
  - **Best for:** Quick decisions where perfect accuracy is not essential.
- **Backtracking:**
  - **Efficiency:** Moderately efficient, especially in structured problems.
  - **Best for:** Problems where multiple paths can be explored and retracted, like puzzles or algorithmic challenges.
- **Means-Ends Analysis:**
  - **Efficiency:** Generally efficient due to systematic problem breakdown.
  - **Best for:** Complex problems that can be decomposed into smaller sub-problems with a clear path to the goal.

Conclusion

- The efficiency of these problem-solving strategies varies based on the nature of the problem:
- **Trial and Error** can be the least efficient, with a potentially high number of steps.
- **Heuristics** are faster but may not always lead to the optimal solution.
- **Backtracking** is efficient in structured environments where wrong paths can be retraced.
- **Means-Ends Analysis** is often the most efficient for complex, multi-step problems.
- Understanding the problem type and applying the appropriate strategy will lead to better problem-solving efficiency.

Problem Solving Process

- **Problem Definition:** The first step is to clearly define the problem. This includes understanding the requirements, constraints, and the desired output. The problem should be broken down into smaller, manageable components.
- **Algorithm Design:** Once the problem is understood, the next step is to design an algorithm—a step-by-step procedure or set of rules to solve the problem. The algorithm should be efficient, both in terms of time and space complexity, and should be validated for correctness.
- **Implementation:** After designing the algorithm, it is translated into a computer program using a programming language. The program is composed of instructions that the computer can execute to perform the desired task. This involves coding, debugging, and testing.

- **Execution:** The computer executes the program, following the instructions sequentially or as directed by control structures (loops, conditionals, etc.). The computer, as a model of computation, processes the inputs and produces outputs according to the logic defined in the program.
- **Testing and Verification:** The program is tested with different inputs to ensure it behaves as expected. Testing helps in identifying any errors or inefficiencies in the implementation. Verification ensures that the program meets the specifications defined in the problem statement.
- **Optimization:** If necessary, the program is optimized for better performance, which might involve refining the algorithm or improving the code to reduce runtime or memory usage.
- **Documentation and Maintenance:** Finally, the process involves documenting the program, which includes explaining the code, algorithms, and design decisions. This documentation is crucial for future maintenance, updates, and debugging.

EXAMPLE

- **Problem Statement**
- **Find the sum of all even numbers between 1 and 100.**
- **Step 1: Problem Definition**
- **Input:** The range of numbers from 1 to 100.
- **Output:** The sum of all even numbers within this range.
- Our task is to identify all the even numbers between 1 and 100 and then add them together to get the total sum.
- Step 2: Algorithm Design
- Initialize a variable sum to 0. This will store the total sum of even numbers.
- Loop through all numbers from 1 to 100.
- Check if the current number is even. If it is, add it to sum.
- Return or print the value of sum after the loop ends.
- This algorithm is simple and efficient for our problem.

- Step 3: Implementation
-
```
sum = 0
for number in range(1, 101):
    if number % 2 == 0:
        sum += number
print("The sum of all even numbers between 1 and 100 is:", sum)
```

- Step 4: Execution
- It initializes sum to 0.
- It loops through each number from 1 to 100.
- For each number, it checks if the number is even (i.e., number % 2 == 0).
- If the number is even, it adds it to the sum variable.
- After the loop finishes, it prints the final value of sum.

- **Step 5: Testing and Verification**
- The sum of all even numbers between 1 and 100 is: 2550We can verify this by manually calculating or using a mathematical formula. The code has worked correctly as expected.
- **Step 6: Optimization (if needed)**
- In this case, the solution is already optimal for the problem size. However, if we were dealing with a much larger range or needed to run this operation frequently, we might look for ways to optimize, such as using a mathematical formula to calculate the sum of even numbers directly:
- Sum=2+4+6+⋯+100
- Sum=2+4+6+⋯+100
- This can be represented as:
- Sum=2×(1+2+3+⋯+50)
- Sum=2×(1+2+3+⋯+50)
- The sum of the first 50 natural numbers is given by:
- Sum=n×(n+1)2
- Sum=2n×(n+1)
- So, the sum becomes:
- Sum=2×50×512=2550
- Sum=2×250×51=2550

- **Step 7: Documentation and Maintenance**
- Finally, it's good practice to add comments and documentation to your code so that others (or you in the future) can easily understand what the code does.
-

```
# This program calculates the sum of all even numbers between 1 and 100.
# Initialize the sum variable

sum = 0

# Loop through numbers from 1 to 100

for number in range(1, 101):
# Check if the number is even
    if number % 2 == 0:
# Add the even number to the sum
    sum += number
# Output the result
print("The sum of all even numbers between 1 and 100 is:", sum)
```

- **Conclusion**
- In this example, we’ve followed the problem-solving process with a computer as a model of computation:
- Defined the problem.
- Designed an algorithm.
- Implemented the algorithm in a programming language.
- Executed the program.
- Tested and verified the output.
- Considered optimization.
- Documented the code.

ESSENTIALS OF PYTHON PROGRAMMING

Identifiers,Variables and Keyword

Identifiers

- An identifier is a name given to entities like variables, functions,class etc. It helps to differentiate one entity from another.

Rules for framing identifiers

1.Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore _.

Names like myClass, var_1 and print_this_to_screen, all are valid example.

2.An identifier cannot start with a digit. 1variable is

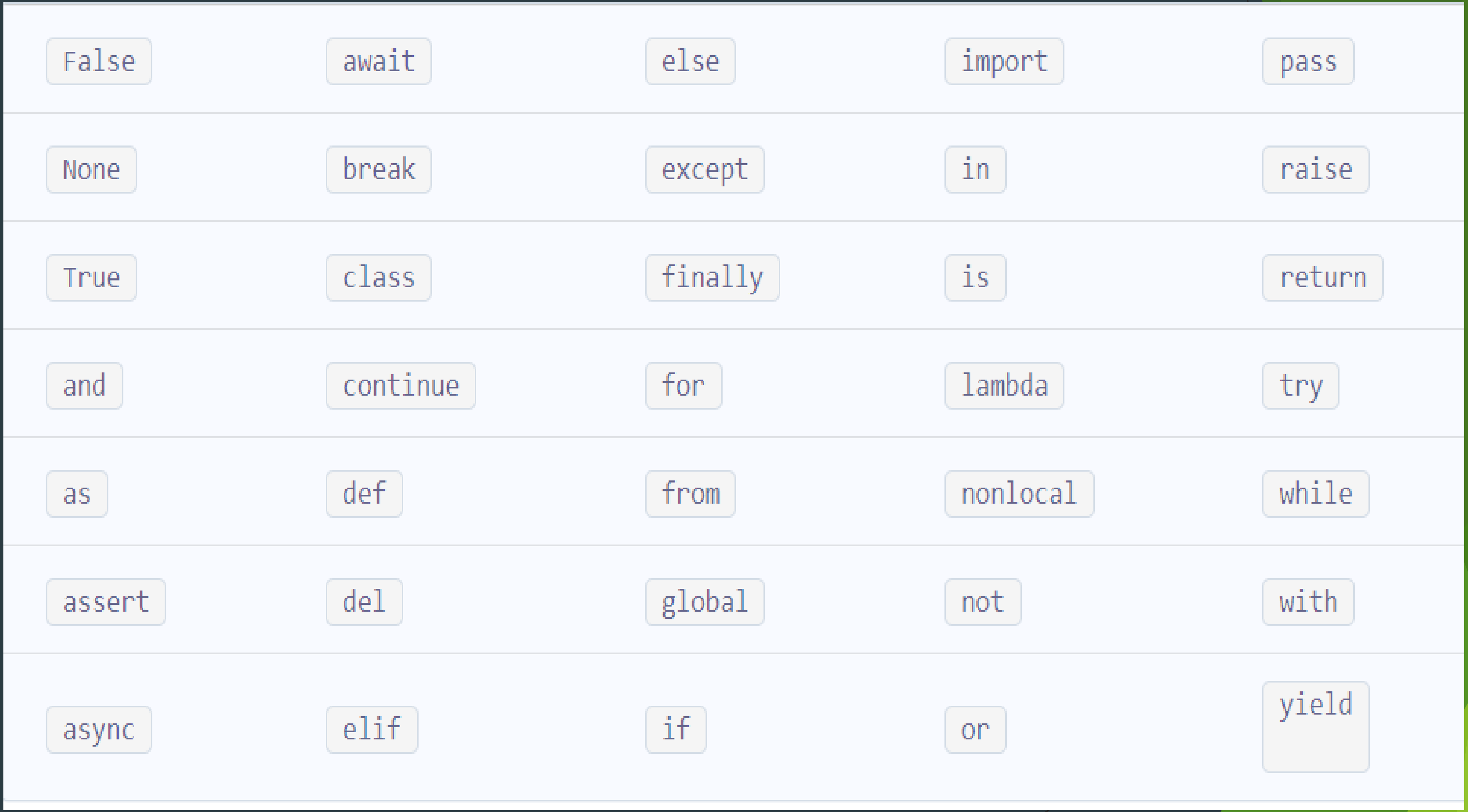
Variables

- A variable can be used to store a certain value or object. In Python, all numbers are objects. A variable is created through assignment. Their type is assigned dynamically.
- Eg:
  x=‘anu'
  y=23
  z=24.5
- use **type(object)** to know the type of the variable object
  Eg: type(x)   type(y)   type(z)
  <type 'str'>   <type 'int'>   <type 'float'>
- Variable can be mutable or immutable. A mutable variable is one whose value may change in place, whereas in an immutable variable change of value will not happen in place.

Keywords

- Keywords are the reserved words in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.

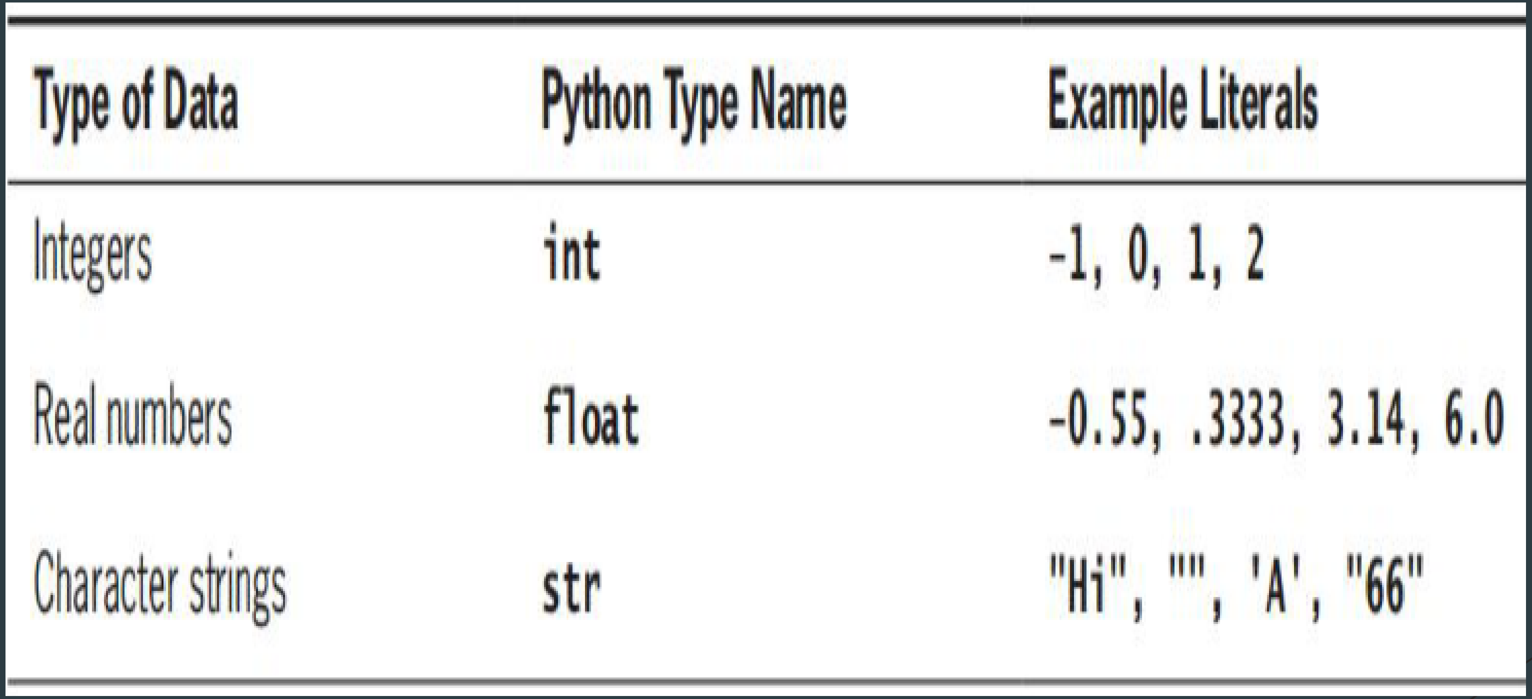
All the keywords except **True**, **False** and **None** are in lowercase and they must be written as they are. The list of all the keywords is given below.



`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

Data Types

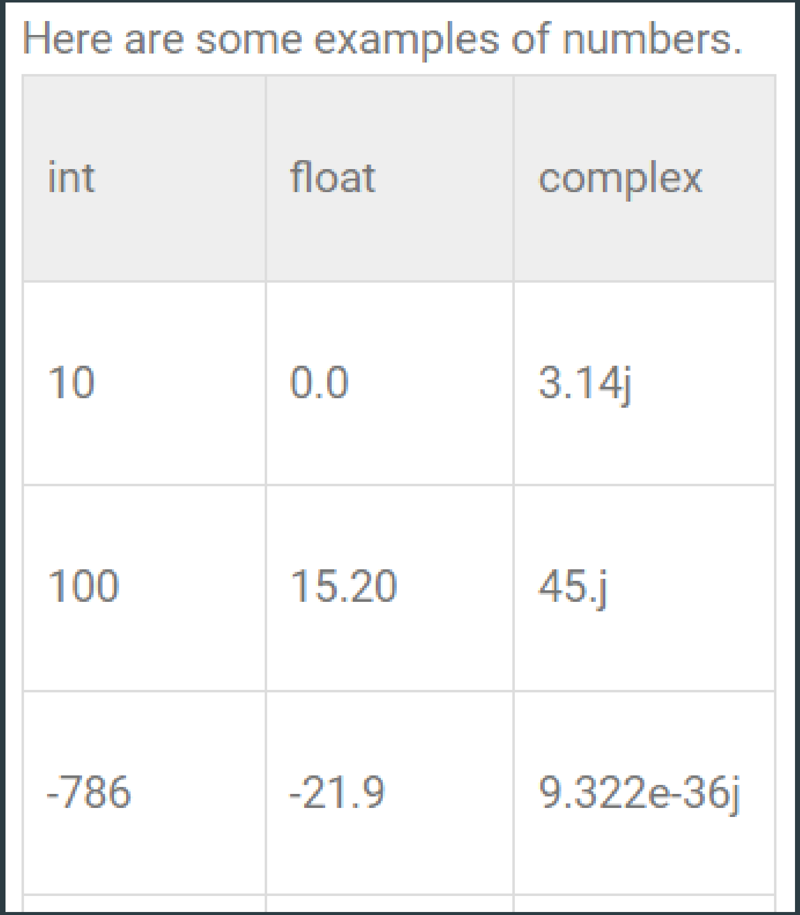
- A data type consists of a set of values and a set of operations that can be performed on those values.
- A literal is the way a value of a data type looks to a programmer. The programmer can use a literal in a program to mention a data value.



Type of Data	Python Type Name	Example Literals
Integers	int	-1, 0, 1, 2
Real numbers	float	-0.55, .3333, 3.14, 6.0
Character strings	str	"Hi", "", 'A', "66"

Python Numbers

- Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as **int, float and complex** classes in Python.



Here are some examples of numbers.

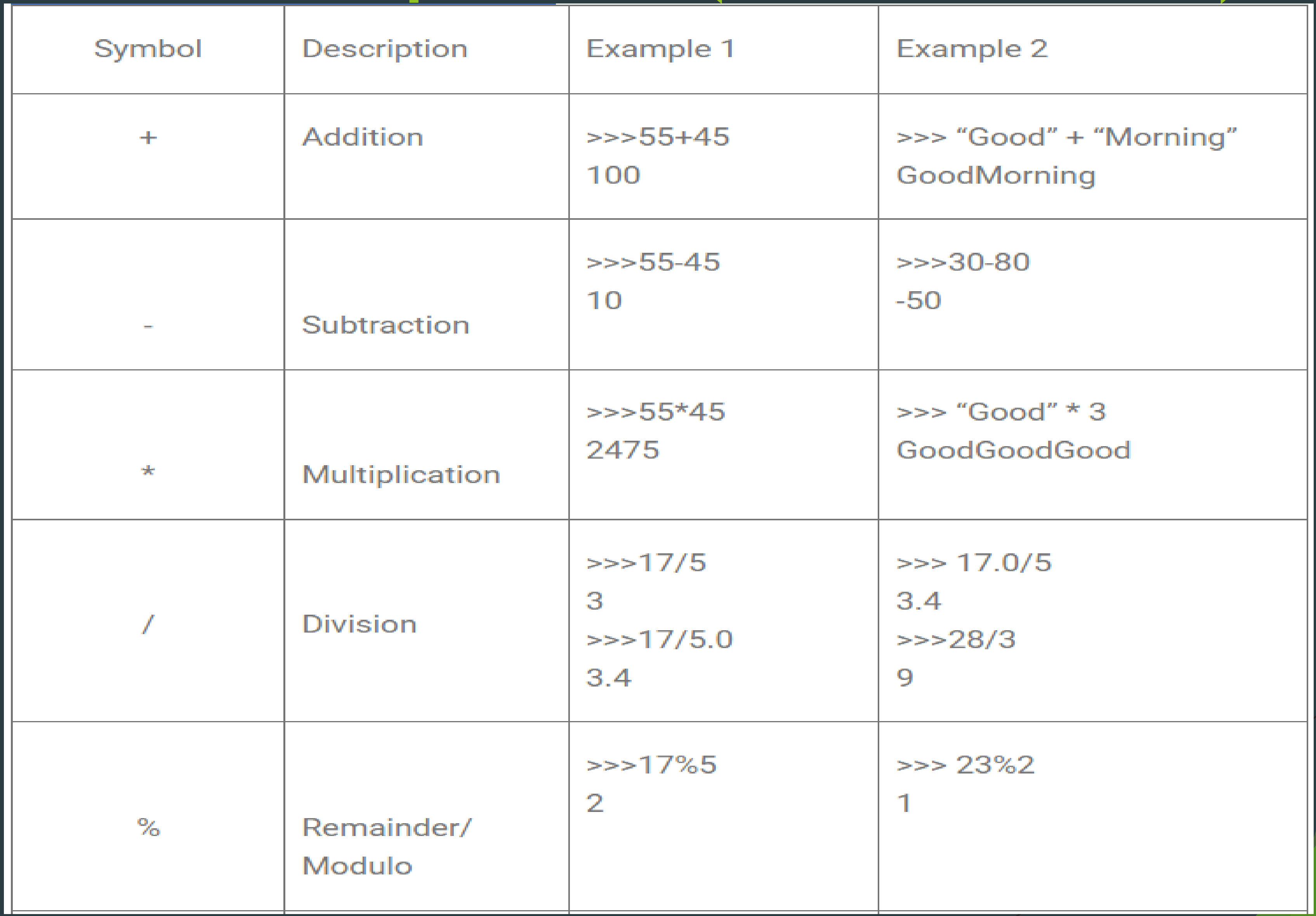
int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j

Operators

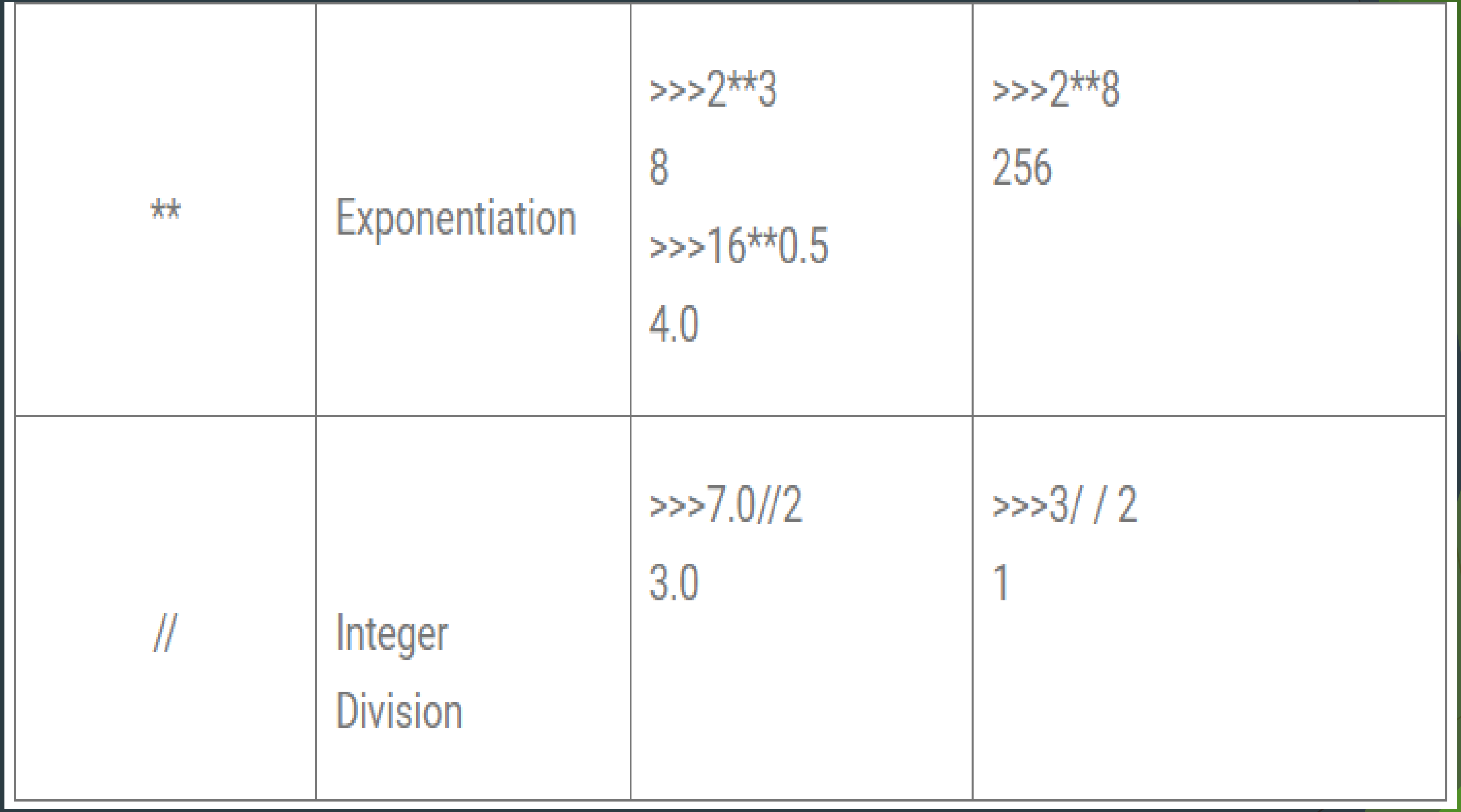
- Operators are special symbols which represents computation. They are applied on operand(s), which can be values(constants) or variables. Same operator can behave differently on different data types. Operators when applied on operands form an expression Operators are categorised as
-

  Arithmetic Operators
  Comparison (Relational) Operators
  Assignment Operators
  Logical Operators
  Bit wise Operators
  Membership Operators
  Identity Operators

Arithmetic Operators(Mathematical)



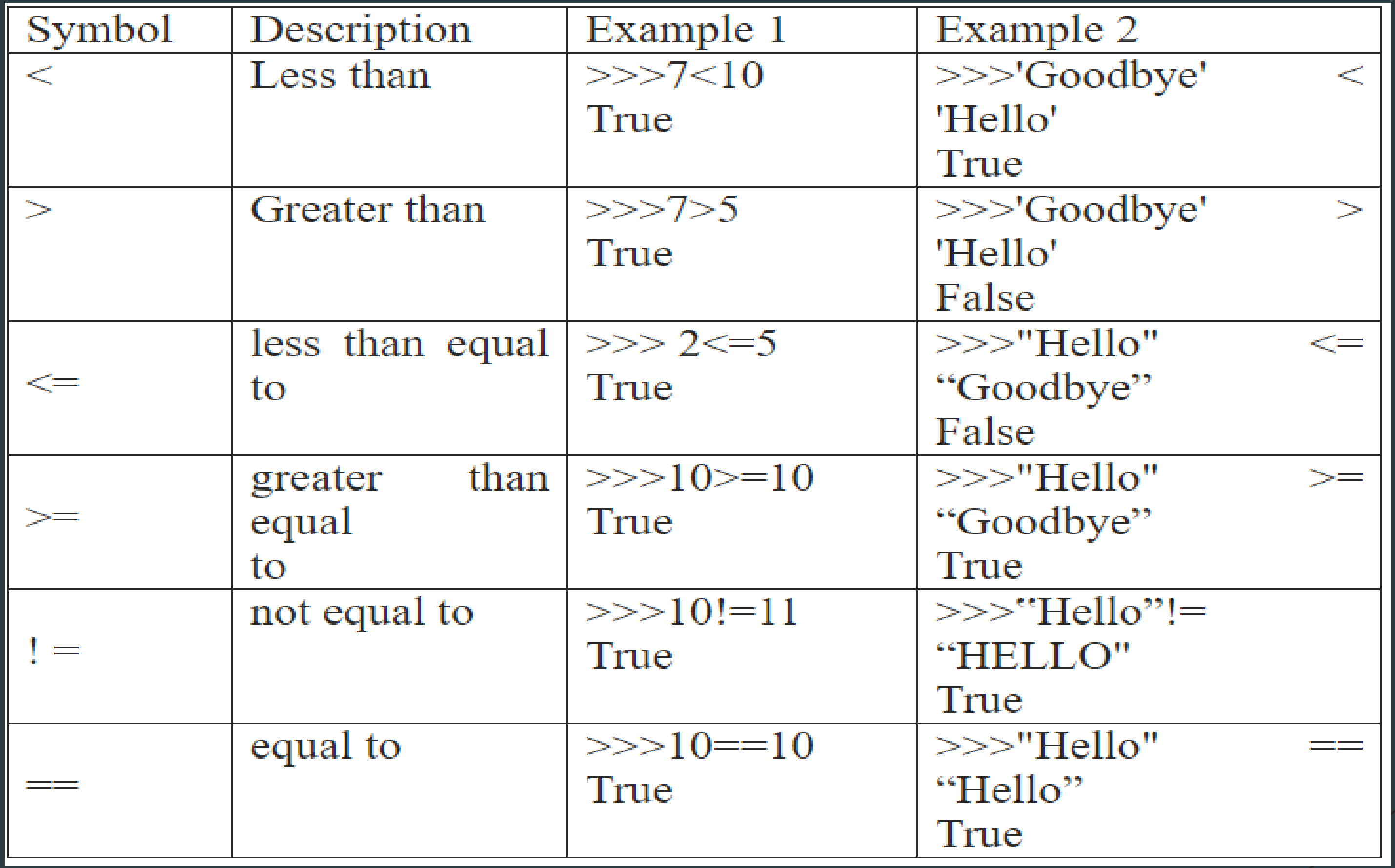
Symbol	Description	Example 1	Example 2
+	Addition	>>>55+45 100	>>> "Good" + "Morning" GoodMorning
-	Subtraction	>>>55-45 10	>>>30-80 -50
*	Multiplication	>>>55*45 2475	>>> "Good" * 3 GoodGoodGood
/	Division	>>>17/5 3 >>>17/5.0 3.4	>>> 17.0/5 3.4 >>>28/3 9
%	Remainder/ Modulo	>>>17%5 2	>>> 23%2 1



**	Exponentiation	>>>2**3 8 >>>16**0.5 4.0	>>>2**8 256
//	Integer Division	>>>7.0//2 3.0	>>>3/ / 2 1

Comparison (Relational) Operators

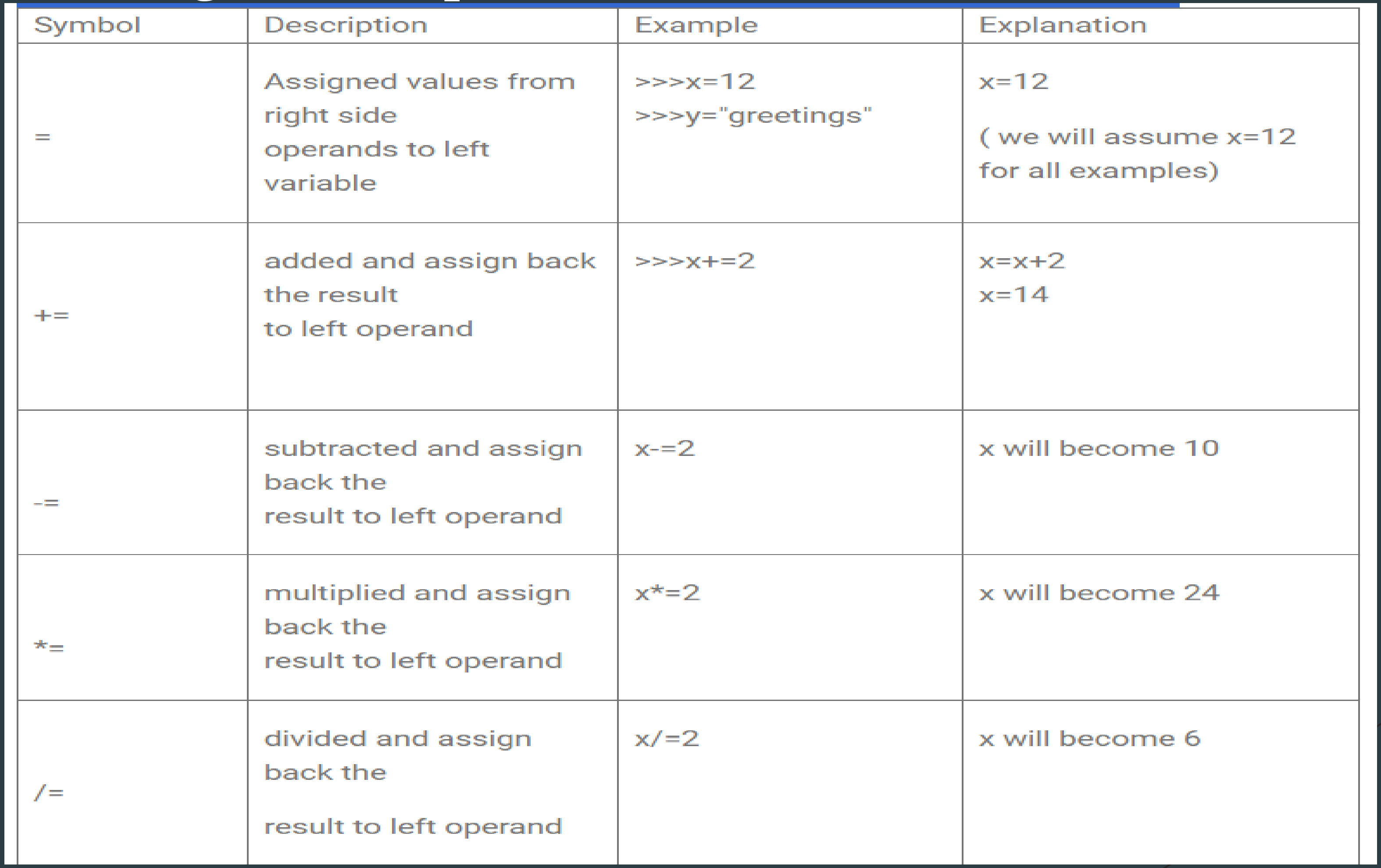
- These operators compare the values on either side of them and decide the relation among them.
  They are also called Relational operators.



Symbol	Description	Example 1	Example 2
<	Less than	>>>7<10 True	>>>'Goodbye' < 'Hello' True
>	Greater than	>>>7>5 True	>>>'Goodbye' > 'Hello' False
<=	less than equal to	>>> 2<=5 True	>>>"Hello" <= “Goodbye” False
>=	greater than equal to	>>>10>=10 True	>>>"Hello" >= “Goodbye” True
! =	not equal to	>>>10!=11 True	>>>“Hello”!= “HELLO" True
==	equal to	>>>10==10 True	>>>"Hello" == “Hello” True

Assignment Operators

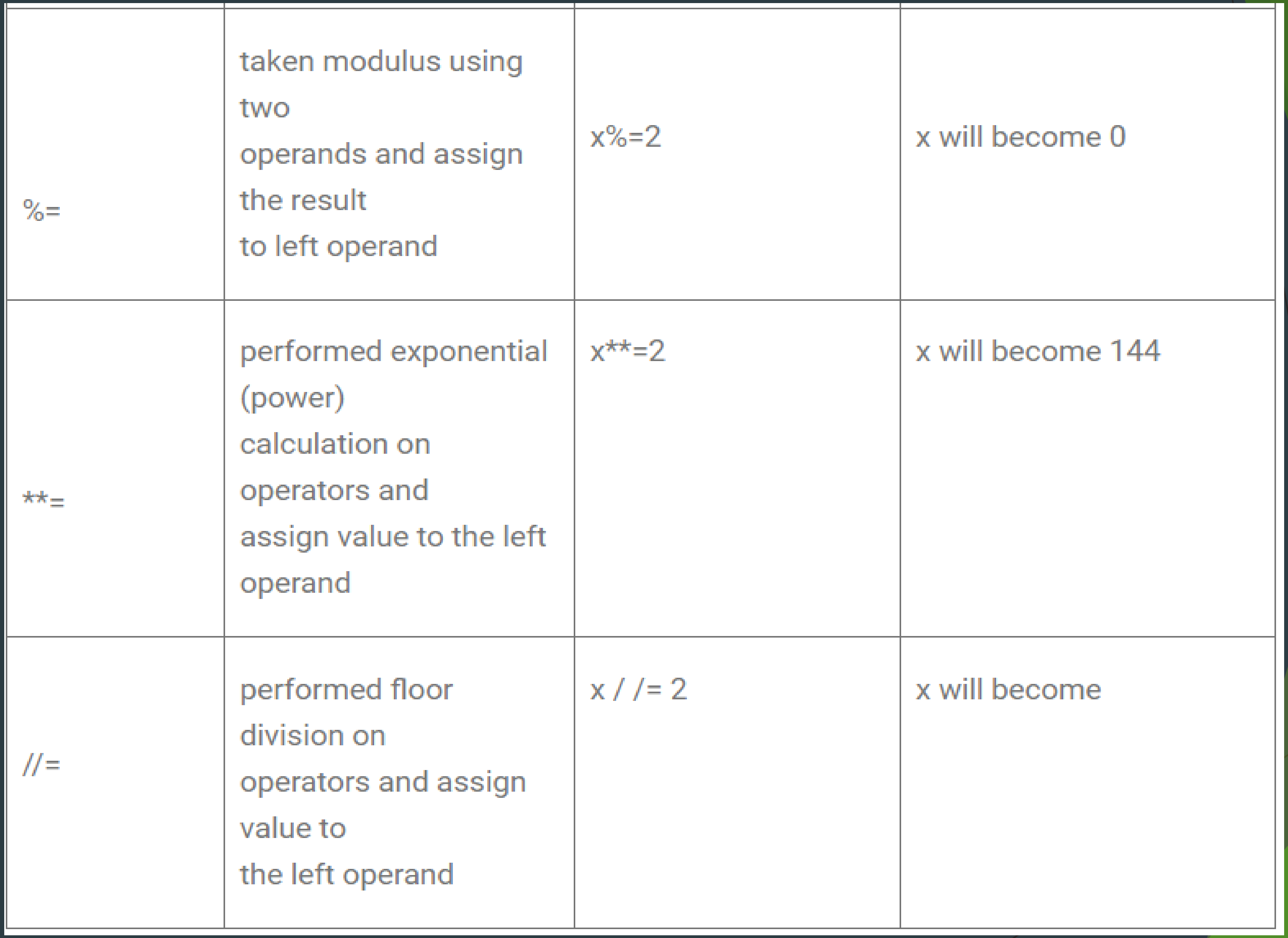
- Assignment Operator combines the effect of arithmetic and assignment operator



Symbol	Description	Example	Explanation
=	Assigned values from right side operands to left variable	>>>x=12 >>>y="greetings"	x=12 (we will assume x=12 for all examples)

+=	added and assign back the result to left operand	>>>x+=2	x=x+2
			x=14

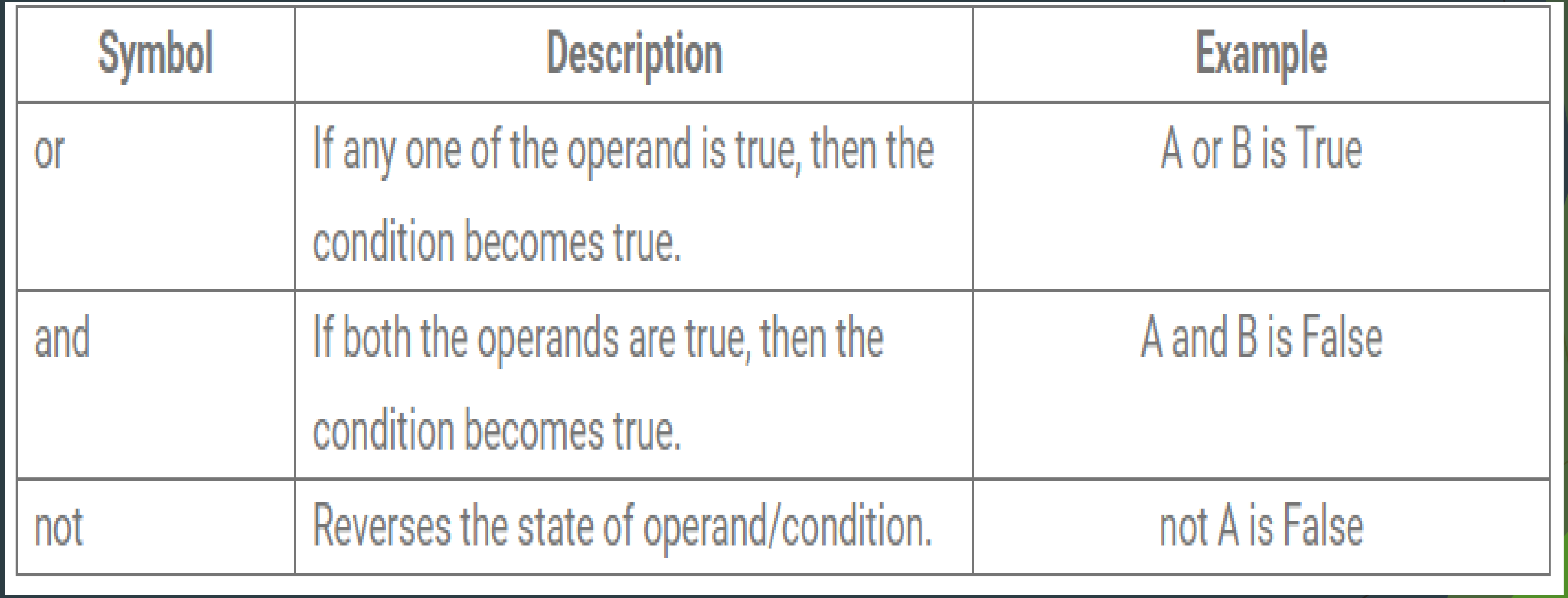
-=	subtracted and assign back the result to left operand	x-=2	x will become 10
*=	multiplied and assign back the result to left operand	x*=2	x will become 24
/=	divided and assign back the result to left operand	x/=2	x will become 6



%=	taken modulus using two operands and assign the result to left operand	x%=2	x will become 0
**=	performed exponential (power) calculation on operators and assign value to the left operand	x**=2	x will become 144
//=	performed floor division on operators and assign value to the left operand	x / /= 2	x will become

Logical Operators

- The following logical operators are supported by Python language. Assume variable A holds True and variable B holds False.

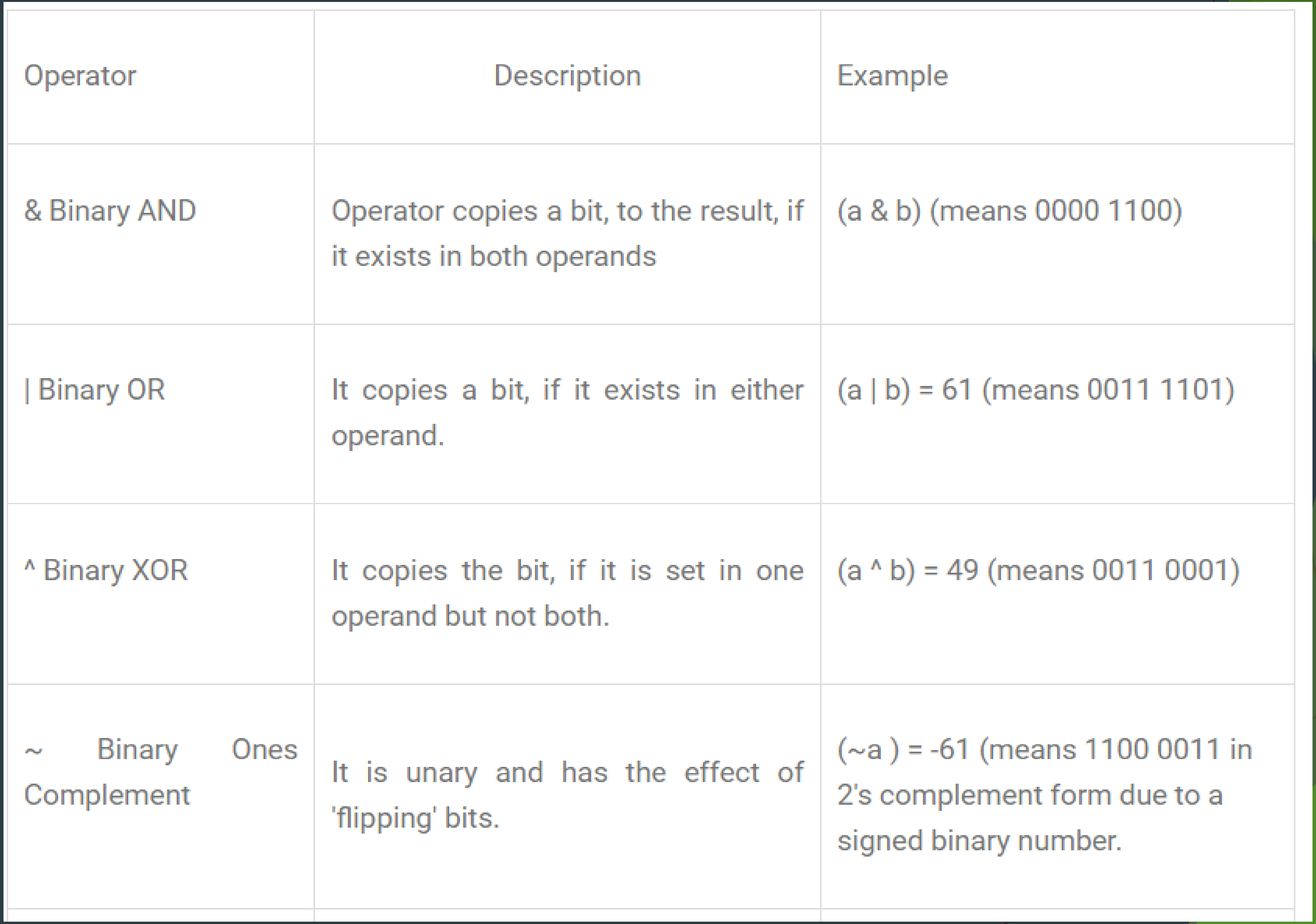


Symbol	Description	Example
or	If any one of the operand is true, then the condition becomes true.	A or B is True
and	If both the operands are true, then the condition becomes true.	A and B is False
not	Reverses the state of operand/condition.	not A is False

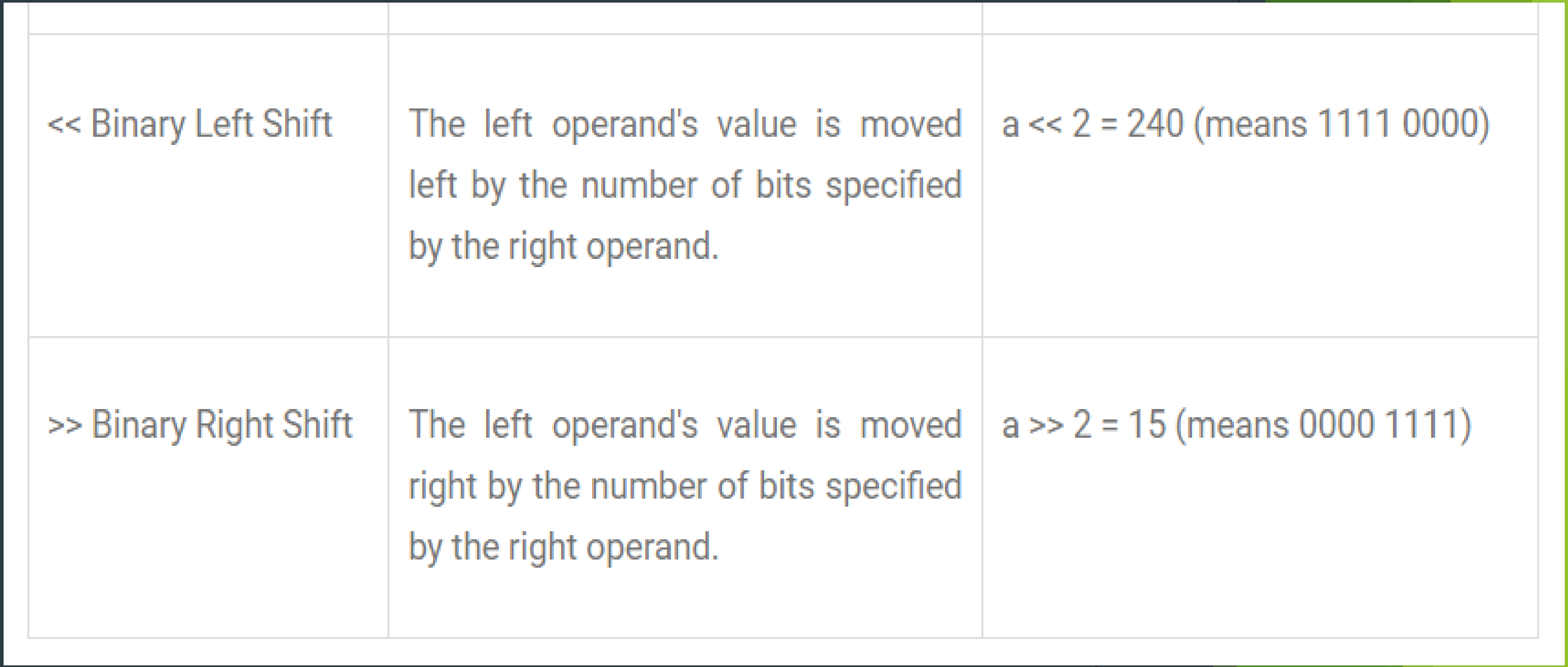
Bitwise Operators

- Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −
- a = 0011 1100
- b = 0000 1101
- -----------------
- a&b = 0000 1100
- a|b = 0011 1101
- a^b = 0011 0001(XOR)
- ~a = 1100 0011
- Python's built-in function bin() can be used to obtain binary representation of an integer number.

- The following Bitwise operators are supported by Python language



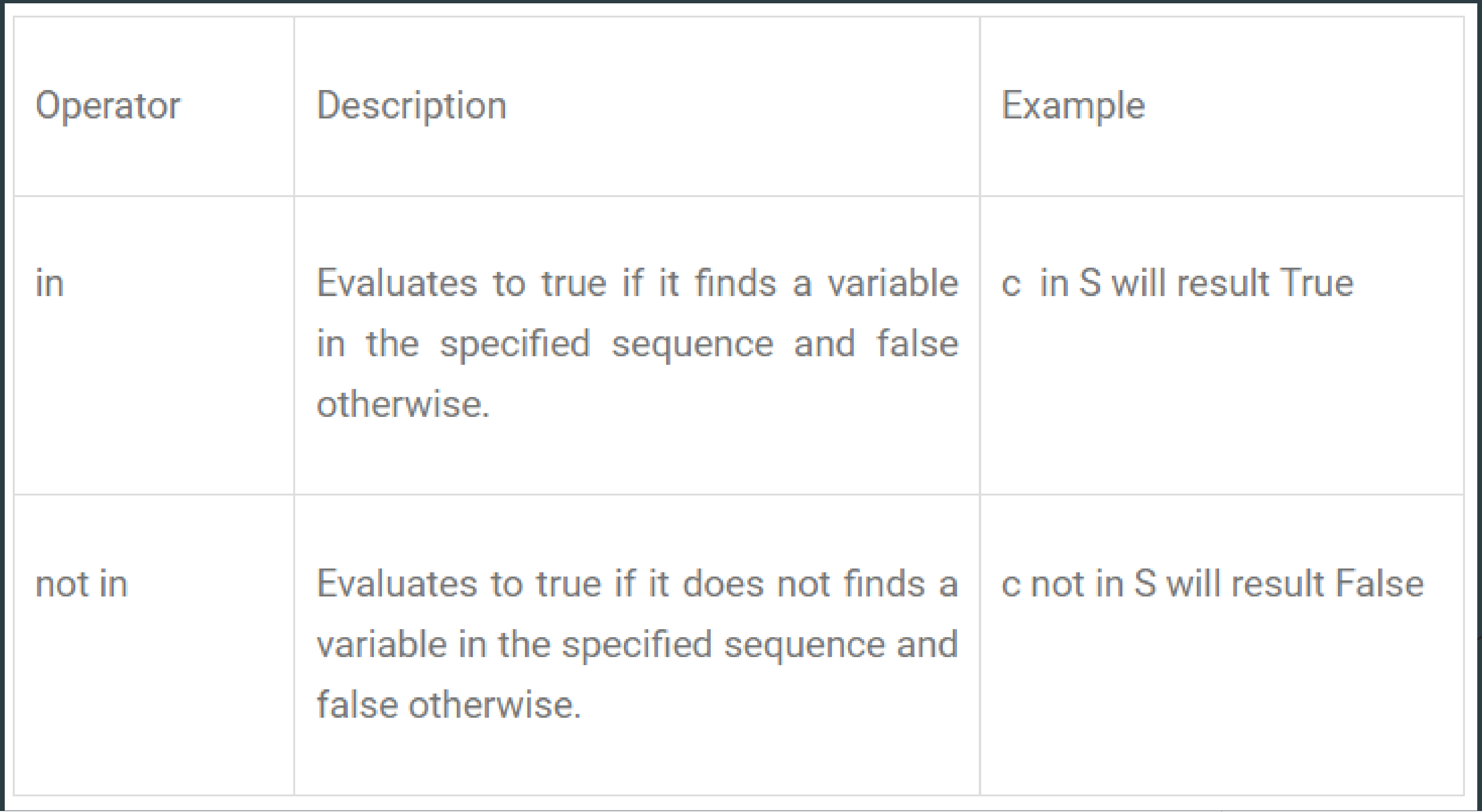
Operator	Description	Example
& Binary AND	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.



<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Membership Operators

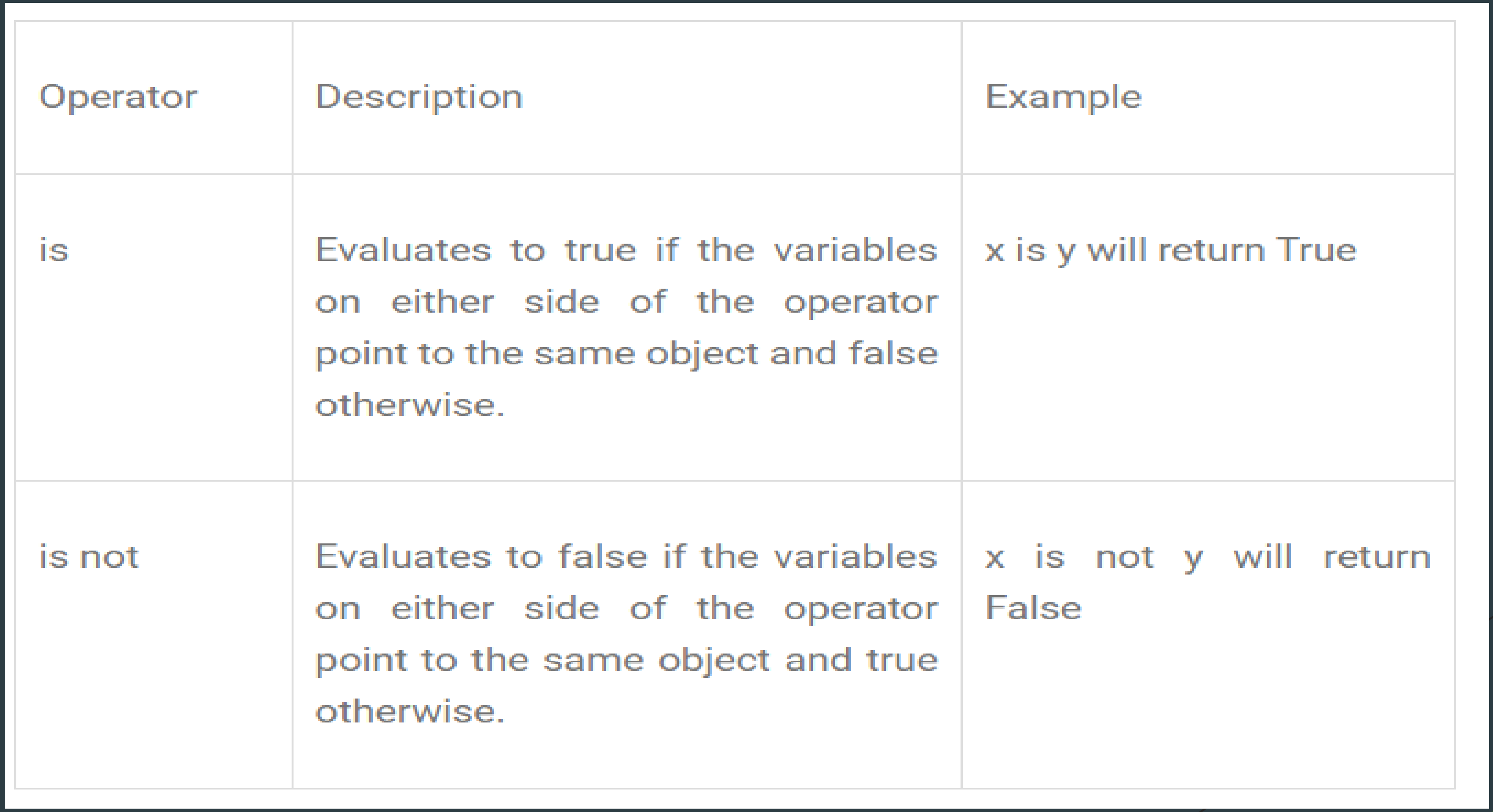
- Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below
- Let S=”Python” c=’o’



Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	c in S will result True
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	c not in S will result False

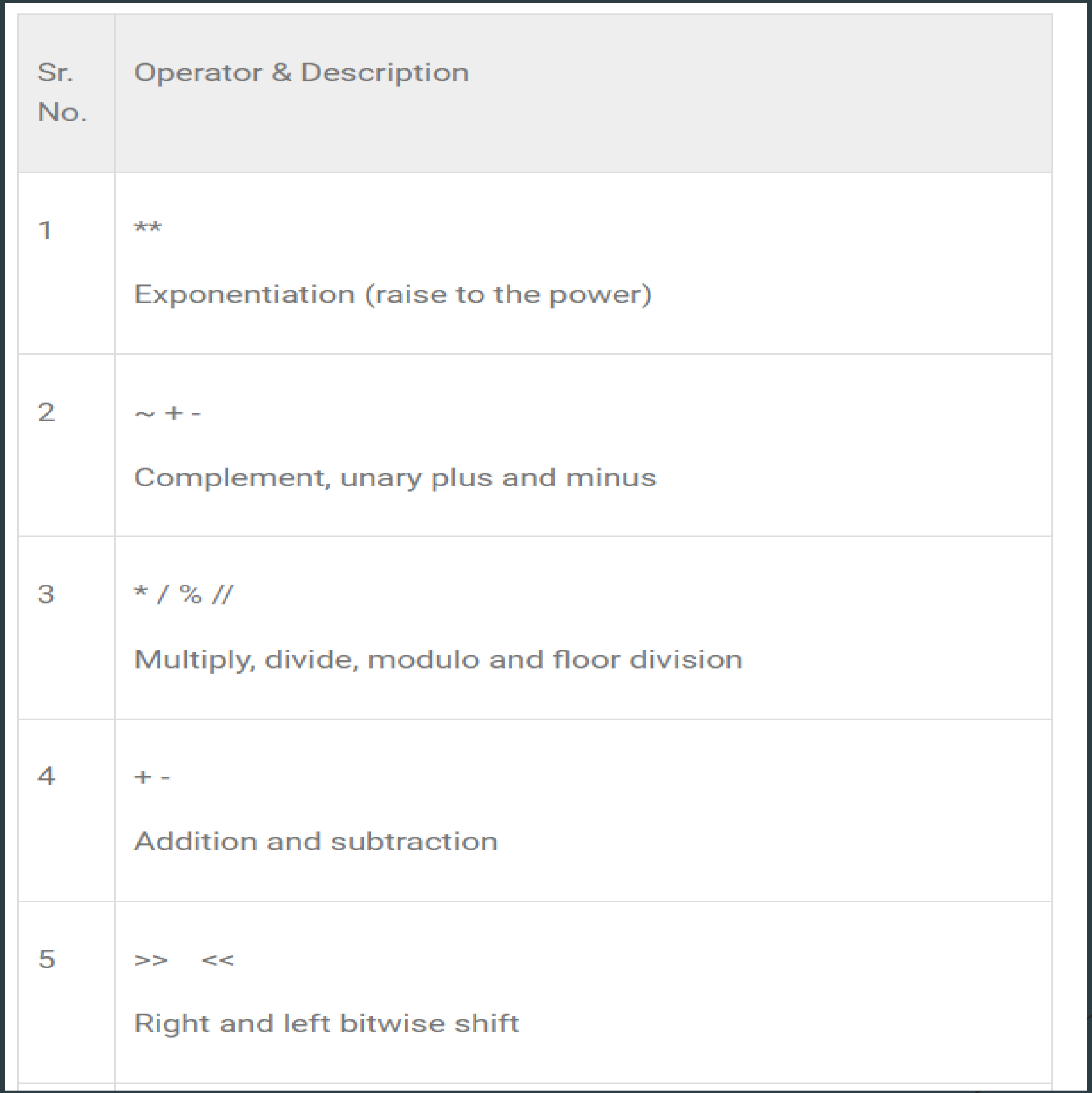
Identity Operators

- Identity operators compare the memory locations of two objects. There are two Identity operators as explained below
- Let x=10 and y=x
- id(x) and id(y) are same in this case

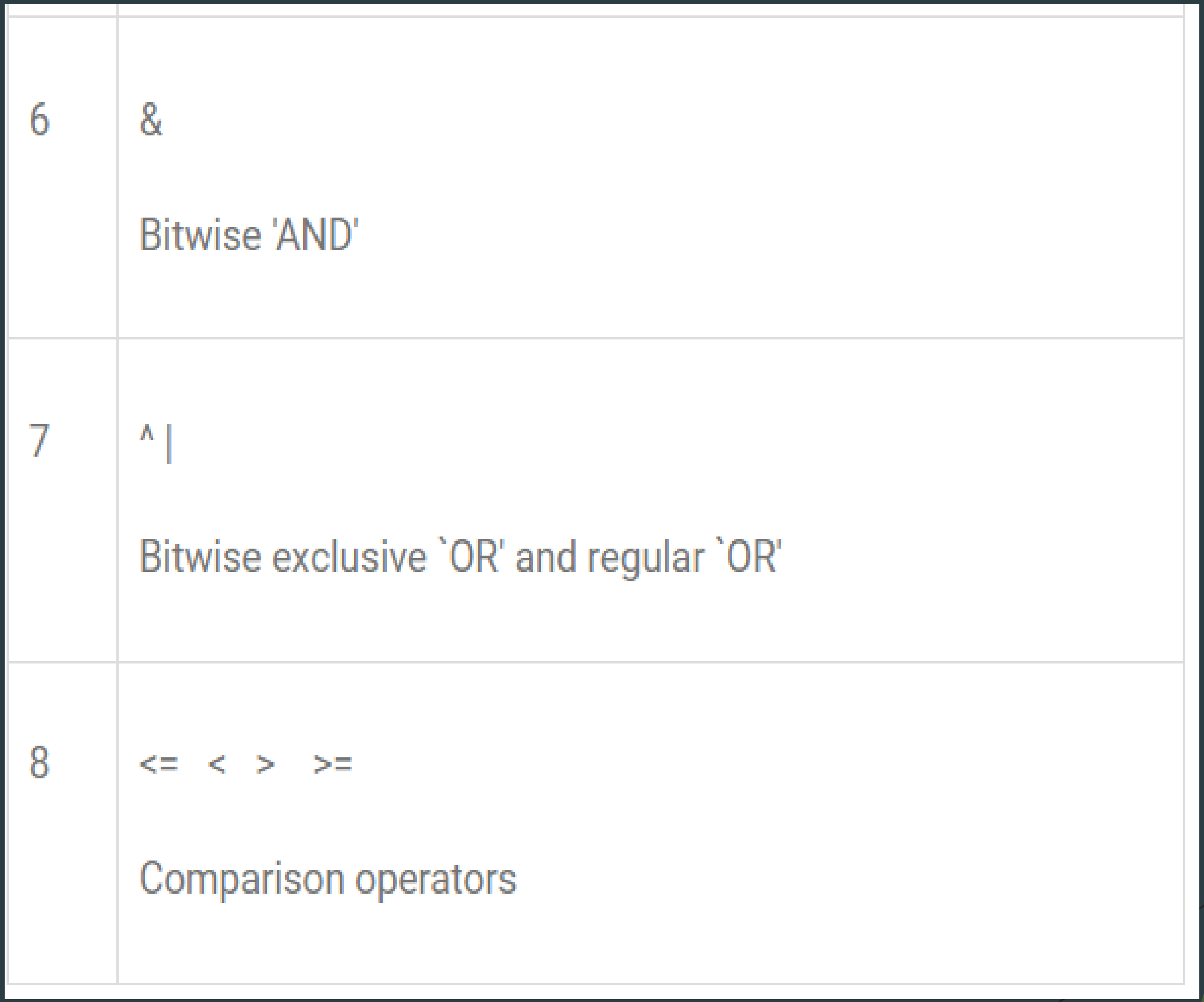


Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y will return True
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y will return False

Operator Precedence and Expression Evaluation



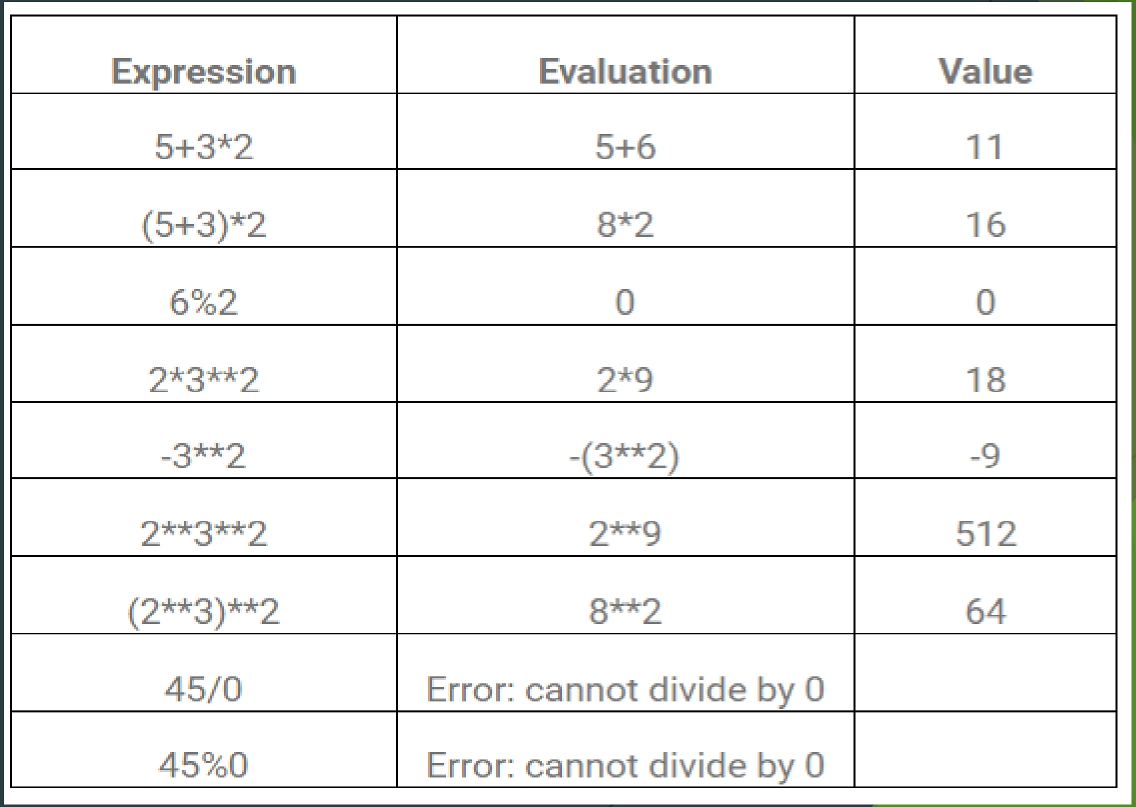
Sr. No.	Operator & Description
1	** Exponentiation (raise to the power)
2	~ + - Complement, unary plus and minus
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>> << Right and left bitwise shift



6	& Bitwise 'AND'
7	^ | Bitwise exclusive `OR' and regular `OR'
8	<= < > >= Comparison operators

Arithmetic Expressions and Evaluation

- Higher precedence operator is worked on before lower precedence operator.
- Operator associativity determines the order of evaluation when they are of same precedence and are not grouped by parenthesis. Parenthesis has high precedence.
- An operator may be Left-associative or Right –associative. In left associative, the operator falling on left side will be evaluated first, while in right assosiative operator falling on right will be evaluated first.



Expression	Evaluation	Value
5+3*2	5+6	11
(5+3)*2	8*2	16
6%2	0	0
2*3**2	2*9	18
-3**2	-(3**2)	-9
2**3**2	2**9	512
(2**3)**2	8**2	64
45/0	Error: cannot divide by 0	
45%0	Error: cannot divide by 0	

Output and Input functions- print() and input()

- **We use the print() function to output data to the standard output device (screen)**
- An example of its use is given below.
- \>>>print('This is a sample output')
  This is a sample output
- The actual syntax of the print() function is:
- **print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)**

  Here, objects is the value(s) to be printed.
  The sep separator is used between the values. It defaults into a space character.
  After all values are printed, end is printed. It defaults into a new line.
  The file is the object where the values are printed and its default value is sys.stdout (screen).

- \>>>print(1, 2, 3, 4, sep='*')
- 1*2*3*4
- \>>>print(1, 2, 3, 4, sep='#', end='&')
- 1#2#3#4&

- Sometimes we would like to format our output to make it look attractive. This can be done by using the **str.format() method**. This method is visible to any string object.
- `>>> x = 5; y = 10`
- `>>> print('The value of x is {} and y is {}'.format(x,y))`
- `The value of x is 5 and y is 10`
- Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).
- **`>>>print('I love {0} and {1}'.format('bread','butter'))`**
- `I love bread and butter`
- **`>>>print('I love {1} and {0}'.format('bread','butter'))`**

- `I love butter and bread`

  We can even use keyword arguments to format the string. **`>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))`**
- `Hello John, Goodmorning`

We can also format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

- >>> x = 12.3456789
- >>> print('The value of x is %6.2f' %x) # rounded to two decimal values
- The value of x is   12.35
- >>> print('The value of x is %13.4f' %x) # rounded to four decimal value
- The value of x is   12.3457

f-strings in Python

- F-strings are faster than the two most commonly used string formatting mechanisms, which are % formatting and str.format().
- #Python3 program introducing f-string
  name = ‘Anu'
  age = 50
  print(f"Hello, My name is {name} age is {age}")
  Output:
- Hello, My name is Anu age is 50

Escape sequences

- Escape sequences are the way Python express special characters in strings such as newline, tab and backspace etc.The back slash(\) is used for escape sequences, it must be escaped to appear as literal character in string. Thus print('\\') will print a single \ character. The following are some of the commonly used escape sequence in Python 3.

Escape Sequence	**Meaning**
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	The \ character
\'	Single quotation mark
\"	Double quotation mark
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Python Input

- The syntax for input() is:
- input([prompt])
- where prompt is the string we wish to display on the screen. It is optional.
- `>>> num = input('Enter a number: ')`
- Enter a number: 10
- `>>>print( num)`
- '10'
- `>>>type(num)`
- `<class 'str'>`

- Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.
- `>>> int('10')`
- 10
- `>>> float('10')`
- 10.0
- `num=int(input(‘enter a number’)`

- This same operation can be performed using the eval() function. This function evaluates a string as a Python expression.
- `>>> eval('2+3')`
- 5

Built-in Functions, Modules and Packages

- **Built-in Functions**

  A function is a named sequence of statement(s) that performs a computation. It contains line of code(s) that are executed sequentially from top to bottom by Python interpreter. They are the most important building blocks for any software development in Python.

- Built in functions are the function(s) that are built into Python Standard Library and can be accessed directly. Functions usually have arguments and return a value.

Some built in functions are given below with examples.

- abs(x) returns the absolute value of x . abs(-45) will return 45
- max(x,y,z) returns the maximum of x,y,z . max(10,20,30) will return 30
- min(x,y,z) returns the minimum of x,y,z . min(10,20,30) will return 10
- divmod(x,y) returns both the quotient and remainder . divmod(14,5) will return (2,4)
- cmp(x,y) returns 0 if x==y , 1 if x>y and -1 if x<y
- round(x,n) round x to n digits. round(3.14567,2) will return 3.15
- range(start,stop,step) will return a list from start to stop-1 with an increment of step.
- range(10) will return [0,1,2,…9] range(1,10,2) will return [1,3,5,7,9]
- type(x) will return the type of the variable object x.
- dir(x) will display the details of the object x.
- len(x) will return the length of the object.
- int(),float(),str(),bool(),chr(),long() these functions can be used for type conversions.
- bin(x), oct(x), hex(x) these functions will convert the decimal number x into corresponding base.

Modules

- When our program grows bigger, it is a good idea to break it into different modules.
- A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.
- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide re usability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

- Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

- For example, we can import the math module by typing the following line:
- **import math**

We can use the module in the following ways:

- >>>import math
- >>>print("The value of pi is", math.pi)
- The value of pi is 3.141592653589793

- **Import with renaming**
  >>>import math as m
- >>>print("sqr root of 25 is", m.sqrt(25))
- sqr root of 25 is 5
- We have renamed the math module as m.
- **We can import specific names from a module without importing the module as a whole. Here is an example.**
- # import only pi from math module **from math import pi** print("The value of pi is", pi)
- **We can also import multiple attributes as follows:**
- >>> from math import pi, e
- >>> pi
- 3.141592653589793
- >>> e
- 2.718281828459045
- **We can import all names(definitions) from a module using the following construct:**
  >>>from math import * # import all names from the standard module math

The dir() built-in function

- We can use the dir() function to find out names and functions that are defined inside a module.

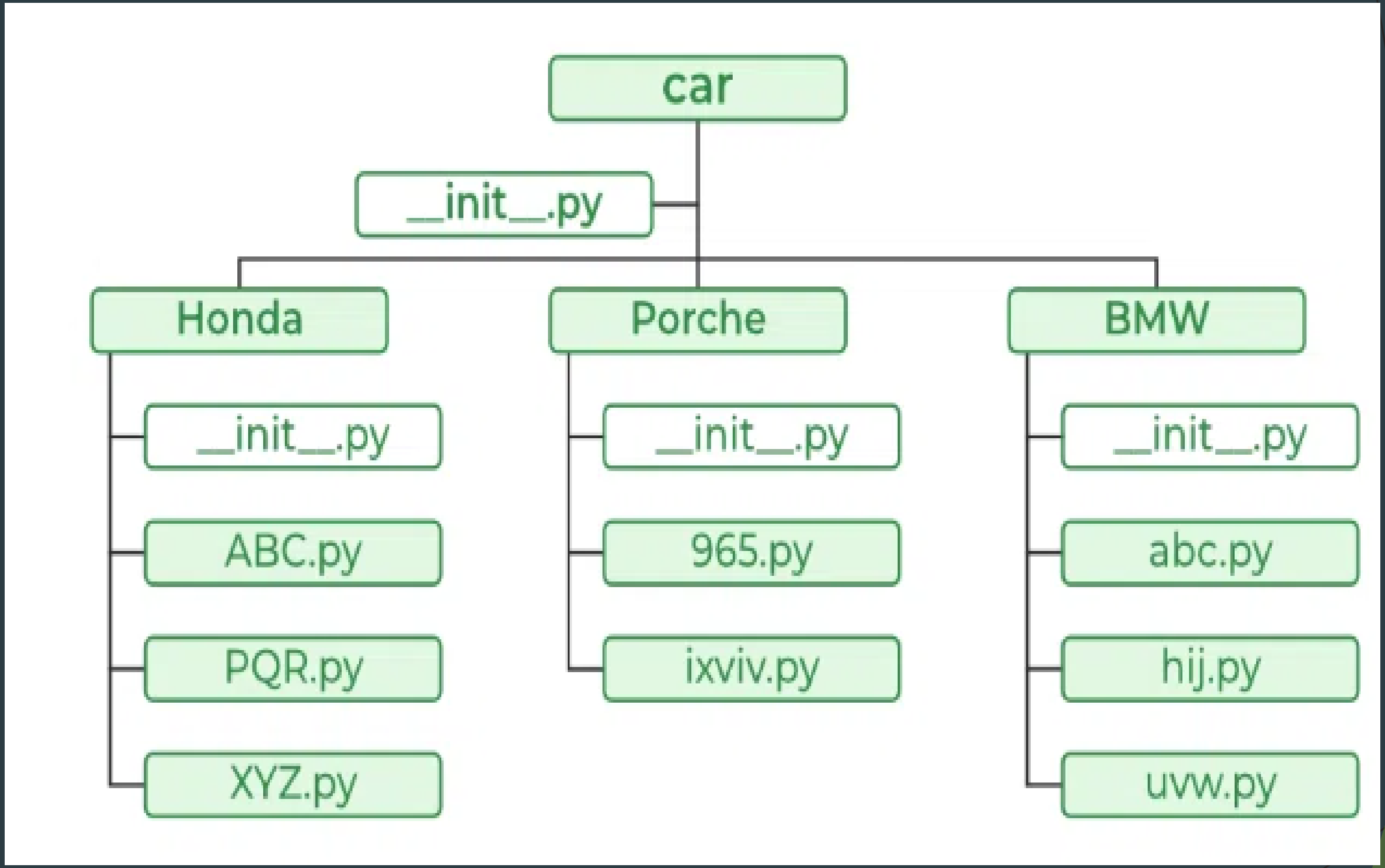
-
  ```
  >>>import math
  >>>dir(math)
  ```

-
  ```
  ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
  ```
- Click the following link to know various Python modules

https://docs.python.org/3.8/py-modindex.html

Packages

- Packages are namespaces which contain multiple packages and modules themselves. They are simply directories.
- Each package in Python is a directory which MUST contain a special file called __init__.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.
- If we create a directory called A, which marks the package name, we can then create a module inside that package called Apple. We also must not forget to add the __init__.py file inside the A directory.
- A package is imported like a "normal" module.To use the module Apple, we can import it in two ways:
- import A.Apple
- from A import Apple


car
__init__.py
Honda
Porche
BMW
__init__.py
ABC.py
PQR.py
XYZ.py
__init__.py
965.py
ixviv.py
__init__.py
abc.py
hij.py
uvw.py

Simple Programs to Begin with, Comments and statements

- **Program Format and Structure**
- It is always better to structure your Python program as follows
  Start with an introductory comment stating the author's name, the purpose of the program and other relevant information. This information should be in the form of comment or document string
- Then include statements that do the following
- Import any modules needed by the program
- Initialize important variables, suitably commented
- Prompt the user for input data and save the input data in variables
- Process the input to produce the results
- Display the results

Comments

- Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.
- Advantages of Using Comments
  Using comments in programs makes our code more understandable. It makes the program more readable which helps us remember why certain blocks of code were written.
- **Single-Line Comments in** Python, we use the hash (#) symbol to start writing a comment.It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

- #This is a comment

```
print('Hello')
```

- **Multi-Line Comments in Python**
  We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. For example:
- #This is a long comment
- #and it extends
- #to multiple lines
- Another way of doing this is to use triple quotes """.
- These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well
- """This is also a perfect example of

multi-line comments"""

Python Statement

- Instructions that a Python interpreter can execute are called statements. For example, a = 1 is an assignment statement. **if** statement, **for** statement, **while** statement, etc. are other kinds of statements which will be discussed later.

Multi-line statement

- In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:
- a = 1 + 2 + 3 + \
- 4 + 5 + 6 + \
- 7 + 8 + 9

This is an explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ], and braces { }.

- For instance, we can implement the above multi-line statement as:
- a = (1 + 2 + 3 +
- 4 + 5 + 6 +
- 7 + 8 + 9)

- We can also put multiple statements in a single line using semicolons, as follows:a = 1; b = 2; c = 3

Strings in Python

- Strings are created by using single quotes or double quotes or triple quotes.
- Example:
- >>>s1="Python Programming"
  >>>s2='Python Programs'
  >>>s3="""Python is a powerful
  Programming language"""
- The positions of string's characters are numbered from 0.So the 5th character is in position 4. When working with strings, the programmer sometimes must be aware of a string's length. Python **len()** function can be used to find the number of characters in the string
- Eg:
- >>> len('python')
- 6
- >>> len("")
- 0

- The string is an **immutable** data structure. This means that its internal characters can be accessed, but cannot be replaced, inserted or removed.

Subscript Operator

string[<an int expression>]

- **Examples:**

-
  ```
  >>>s="Python"
  >>>s[0]
  ‘P’
  >>>s[2]
  ‘t’
  >>>s[5]
  ‘n’
  ```
- we can also use negative index. -1 is the index of last character.
  ```
  >>>s[-1]
  ‘n’
  >>>s[-3]
  ‘h’
  >>>s[-6]
  ‘P’
  ```

- position ranges from 0 to len(string)-1.Invalid index will generate **IndexError.**
- The subscript operator is also useful in loops where you want to use the positions as well as the characters in a string
- for i in range(len(s)):
- print(i,s[i])
- 0 P
- 1 y
- 2 t
- 3 h
- 4 o
- 5 n
- >>>s[2]='c' will leads to an error because strings are immutable.

Slicing for substrings

- We can extract a portion of string called substring using a process called **slicing.** To extract substring, a colon(:) is placed in the subscript.
- s="Python"
- >>>s[1:3]
  'yt'
- If the first index is not mentioned, the slicing will start from 0.
  >>>s[:3]
  'Pyt'
- If the last index is not mentioned, the slicing will go till the end of the string.
  >>>s[3:]
- 'hon'
- >>>s[0:5:2]
  'Pto'

- We can also use –ve index in slicing s="Python"

```
>>>s[1:-1]
'ytho'
>>> s[-4:-1]
'tho'
>>> s[-6:-2:2]
'Pt'
```

Negative step size makes string to be printed in reverse

```
>>> s[5:0:-1]
'nohty'
>>>s[-1:-7:-1]
'nohtyP'
```

Print the full string

```
>>> s[:] or >>>s[::]
'Python'
```

Print the string in reverse

```
>>> s[::-1]
'nohtyP'
```

- **Write the output of following python code : (University Question)**
- S ="Computer"
- print(S[::2])
- print(S[::-1])
- print(S[:])

- output:
- Cmue
- retupmoC
- Computer

String Operations

- + operation can be used as concatenation operation.

```
>>>s1=”Python”
>>>s2=”Programming”
>>>s3=s1+s2
>>>s3
‘Python Programming’
```

‘*’ operation repeats a string specified number of times

```
>>>s4=s1*3 or >>>s4=3*s1
>>>s4
‘PythonPythonPython’
```

“in” and **“not in”** operator can be used to check whether a substring is present in another string

```
>>>”Py” in s1
True
>>>”Py” not in s1
```

String Methods

- Let **s** is Python string then **s.lower() and s.upper()** will return the lowercase and uppercase versions of the string **s.islower() and s.isupper()** can be used to test whether s is lower case or upper case.
- Example
- >>>s=”Python programming”
  >>>s.upper()
  PYTHON PROGRAMMING
  >>>s.lower()
  python programming
  >>>"binu".islower()
- True
- >>>"Binu".islower()
- False
  >>>"BINU".isupper()
- True
- >>>"BINu".islower()
- False

- **s.isalpha(),s.isdigit(),s.isalnum() s.isspace()** will test for alphabets, digits, alpha numeric and space.

```
>>> "abc".isalpha()
True
>>> "ab1".isalpha()
False
>>> "123".isdigit()
True
>>> "ab1".isdigit()
False
```

```
>>> "a12".isalnum()
True
>>> "a+".isalnum()
```

```
False
```

```
>>> " ".isspace()
True
>>> "abc".isspace()
```

- **s.find()** and **s.index()** can be used to find a substring in the given string.It returns the position(index) of the first occurrence of the substring.
- If it is not present it will return -1 in case of find() method and throws an error in case of index() method.
- We can also specify the start and end position of the string index so that the search will be done in this range only.We can also use **rfind()** to search for a substring from the right end of the string.

- >>>s=”Python programming”
- >>>s.find(‘pr’)
- 7
- >>>s.find(‘to’)

```
-1
>>> s.find('o')
4
>>> s.rfind('o')
9
>>>s.find(‘pr’,8,-1)
-1
>>> s.find('ton')
-1
>>> s.index('ton')
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
s.index('ton')
ValueError: substring not found
```

-
  ```
  >>>s.index('on')
  4
  ```
- **s.count()** can be used to count the number of occurrence of a substring.
  ```
  >>>s.count('m')
  2
  >>>s.count('mii')
  0
  ```
- **s.title()** converts into a title my making first character of each word capital. **s.istitle()** can be used to test whether the string is title cased.

-
  ```
  >>>s.title()
  'Python Programming'
  ```

- **s.capitalize()** Return a copy of the string s with only its first character capitalized.The remaining characters are turned into small case.

-
  ```
  >>>"hello MAN".capitalize()
  'Hello man'
  ```
- **s.swapcase()** will swap the case of each character.(upper case into lower case and vice versa)

-
  ```
  >>> s.swapcase()
  'pYTHON pROGRAMING'
  ```

  s.replace(old,new) will replace all the occurrence of old substring with new
  ```
  >>> s.replace('m',"M")
  'Python PrograMMing'
  ```
- **s.split(char)** will take a character and split s based on char. This function can be used to split a string into words with space as split character.

-
  ```
  >>> s.split(" ")
  ['Python', 'Programming']
  >>> s.split("n")
  ['Pytho', ' Programmi', 'g']
  ```

- The **partition()** function will partition the string into 3 parts. They are, string before partition string, the partition string and the string after partition string.

-
  ```
   >>> s="this is a test"
  >>> s.partition("is")
  ('th', 'is', ' is a test')
  ```

- The **lstrip()** and **rstrip()** finctions can be used to strip white spaces or a substring from left or right end. **strip()** method is used to do the stripping from both the ends. When no substring is specified, white spaces are striped.

-
  ```
  >>> s="this is a test"
  ```

-
  ```
  >>> s.lstrip('th')
  'is is a test'
  >>> s.rstrip('st')
  'this is a te'
  >>> s.strip('t')//the leading and trailing t will be removed
  'his is a tes'
  ```

- **join()** method joins the string elements with a separator.

```
>>> ":".join(s)
'P:y:t:h:o:n: :P:r:o:g:r:a:m:m:i:n:g'
```

This will create a reversed string

```
>>>''.join(reversed(s))
```

Program 1

- Program to find Area and Circumference of a Circle

```
#Python Program to find Area and Circumference of a Circle

#Standard formula to calculate the Area of a circle is: a=π r².

#Circumference c=2 π r.

import math
r=input("Enter radius :")
r=int(r)
a=math.pi * r * r
c=2* math.pi * r
print("Area of the circle",a)
print ("Circumference of the circle",c)
```

Program 2

- Input a time in seconds and print the time in HH:MM:SS format (university question)

-
  ```
  #proram to convert time in sec to HH:MM:SS format
  time=input("Enter time in seconds")
  time=int(time)
  timeinmin=time//60# Converts total seconds into minutes (ignoring remainder)
  timeinsec=time%60# Finds the remainder, which are the remaining seconds
  timeinhr=timeinmin//60# Converts total minutes into hours (ignoring remainder)
  timeinmin=timeinmin%60# Finds the remainder, which are the remaining minutes
  print("HH:MM::SS----{}:{}:{}".format(timeinhr,timeinmin,timeinsec))
  ```

Example:

If the input is 3671 seconds:

timeinmin = 3671 // 60 = 61 (so there are 61 total minutes)

timeinsec = 3671 % 60 = 11 (remaining seconds)

timeinhr = 61 // 60 = 1 (so there is 1 hour)

timeinmin = 61 % 60 = 1 (remaining minute)

PROGRAM 3

- **Write a program to convert temperature in degree Fahrenheit to Celsius. ((farenheit-32) *5/9= Celsius))**
-
```
tf= float(input('Enter the temperature in Farenheit:'))
tc=(tf-32)*5/9
print("Temperature in celsius %0.3f"% tc)
```

Assignment questions

- Write a program to read P,T, R and calculate simple interest.(SI=(P*T*R)/100)
- Enter length and breadth of a rectangle and find its area and perimeter.
- Enter name and marks in 6 subjects of a student and find the total, average and percentage.
- Find the biggest and smallest of three numbers.(use min and max function)
- Read a number and find its factorial. (use factorial function from math)
- Write a Python program to find the sum of even digits in a number
- Write a Python program to print all prime numbers less than 100.