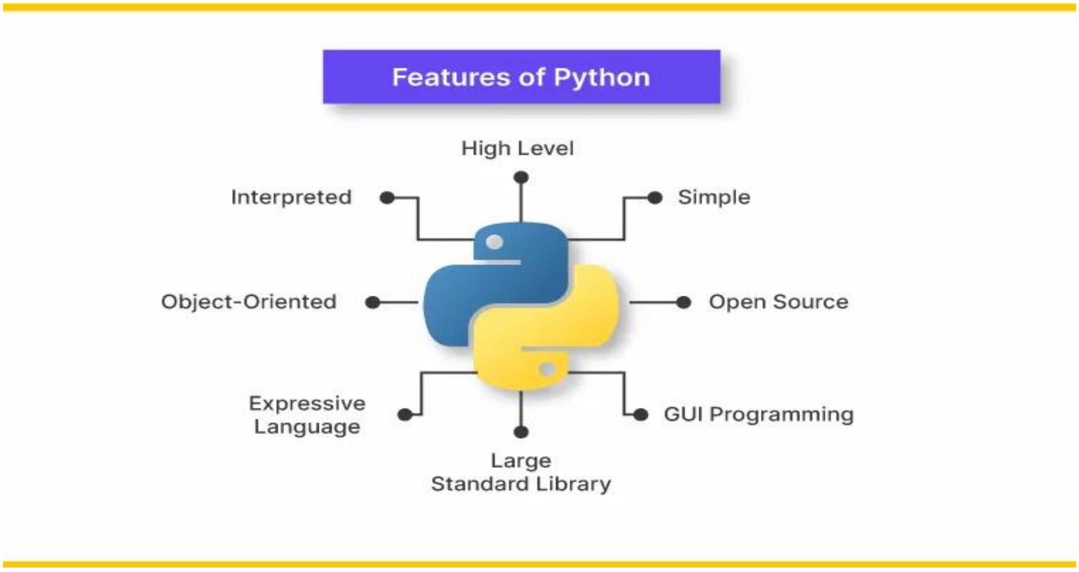
Python Notes: From Basics to Advanced

1. Introduction to Python

What is Python? Python is a high-level, interpreted, general-purpose programming language. It is known for its simplicity and readability, often referred to as "executable pseudocode." Python supports multiple programming paradigms, including object-oriented, imperative, and functional programming. Python is widely used in academic, corporate, and hobbyist environments due to its readability and rich ecosystem.



Why Python?

- **Simplicity and Readability:** Easy to learn and use, with a clear syntax that makes code easy to read and understand.
- **Versatility:** Used in web development, data science, artificial intelligence, machine learning, scientific computing, automation, game development, and more.
- **Large Community & Libraries:** A vast and active community provides extensive support and a rich ecosystem of libraries and frameworks.
- **Cross-platform:** Runs on various operating systems like Windows, macOS, and Linux.
- **Open Source:** Freely available and can be modified and distributed.

Example (Hello World):

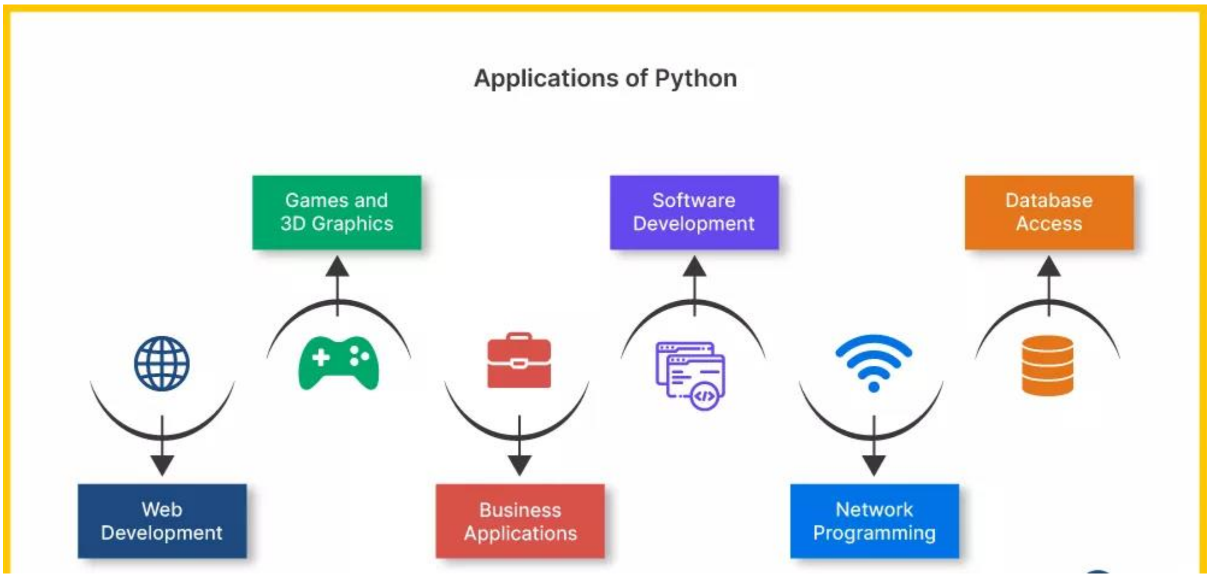
Python

```
print("Hello, Python!")
```

Beneficial Visuals:

- **Mind Map:** A central "Python" node branching out to "Uses," "Key Features," and "Why Learn?" with sub-branches for specific examples (e.g., "Web Dev: Django," "Data Science: Pandas").

- **Diagram:** A simple flowchart showing input -> Python processing -> output, highlighting its interpreted nature.



2. History of Python

Python was created by Guido van Rossum in the late 1980s and first released in 1991. Its name was inspired by the British comedy group Monty Python. Guido van Rossum remained Python's benevolent dictator for life (BDFL) until 2018.

Key Milestones:

- **1991:** Python 0.9.0 released.
- **2000:** Python 2.0 released, introducing features like list comprehensions and garbage collection.

• 2008: Python 3.0 released — a major overhaul that is not backward-compatible with Python 2.x, introducing better Unicode support, integer division behavior, and other improvements.Beneficial Visuals:

- **Timeline:** A chronological timeline highlighting key Python versions and their release dates.
- **Picture:** A picture of Guido van Rossum.

3. Installation

Installing Python is straightforward. The recommended way is to download the installer from the official Python website.

Steps (General):

1. Go to python.org.
2. Download the latest stable version of Python for your operating system.
3. Run the installer. **Important:** On Windows, make sure to check the box "Add Python X.Y to PATH" during installation. This allows you to run Python from the command line.

4. Verify installation by opening a terminal or command prompt and typing: python --version or python3 --version.
5. 5. Open Python (IDLE or terminal) and try running: print("Installation Successful!") to confirm everything works.

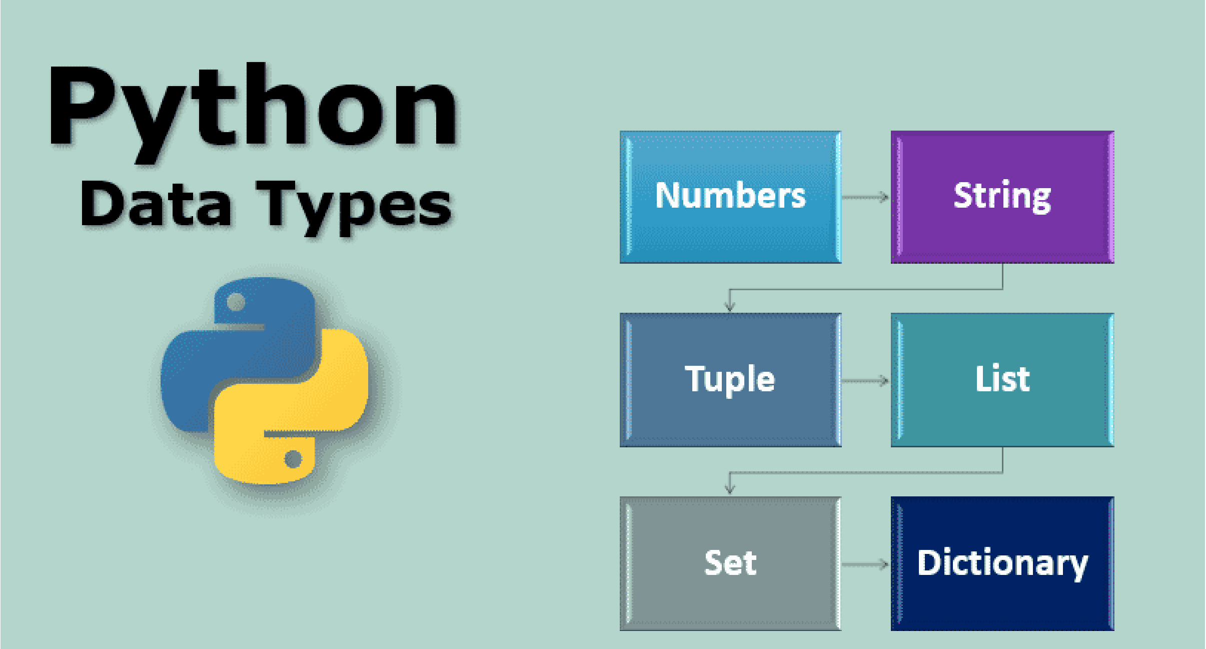
Example (Verifying Installation):

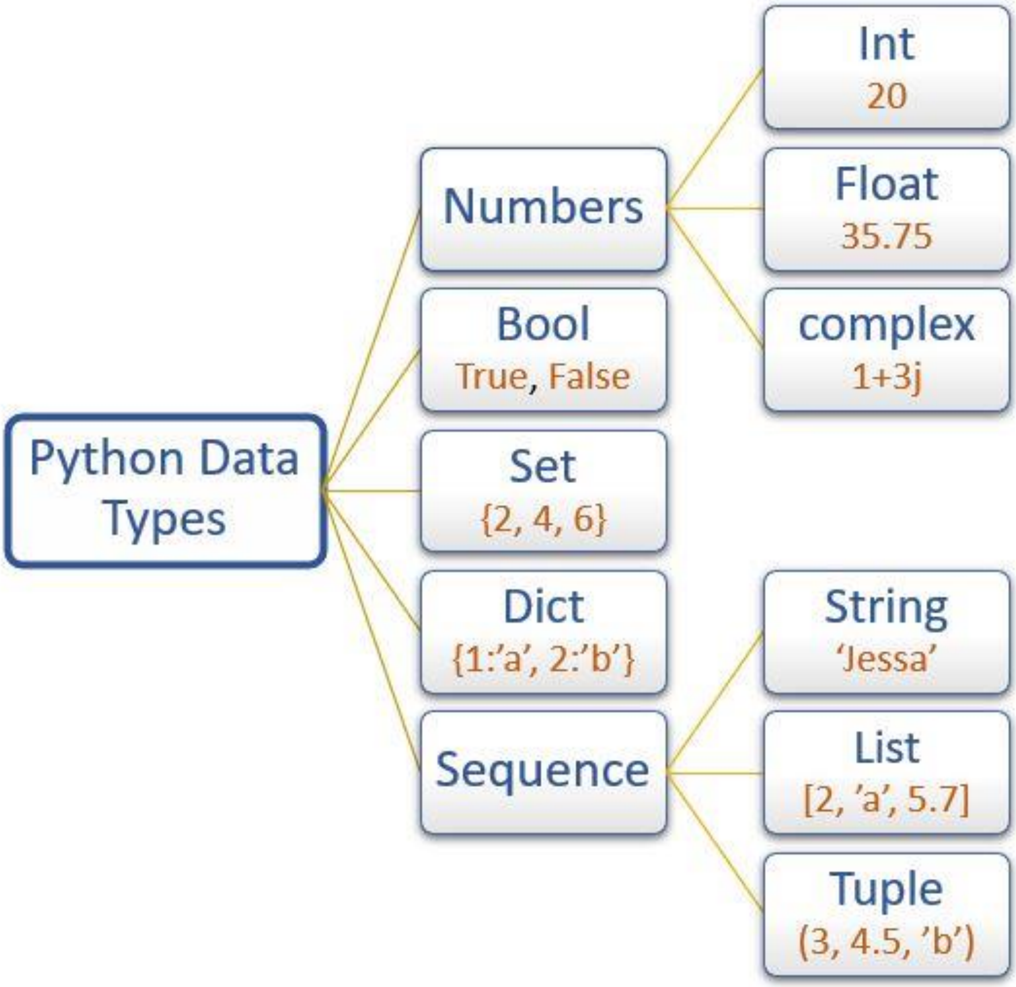
Bash

```
# In your terminal or command prompt

python --version

# Expected output (or similar): Python 3.10.0
```

Beneficial Visuals:

- **Screenshots:** Step-by-step screenshots of the installation process, especially highlighting the "Add to PATH" option.
- **Flowchart:** A simple flowchart illustrating the download -> install -> verify process.





4. Data Types

Data types classify the type of values that a variable can hold. Python is dynamically typed, meaning you don't need to explicitly declare the type of a variable.

Common Data Types:

- **int (Integers):** Whole numbers (positive, negative, or zero).

Python

age = 30

print(type(age)) # <class 'int'>

- **float (Floating-point numbers):** Numbers with a decimal point.

Python

price = 19.99

print(type(price)) # <class 'float'>

- **str (Strings):** Sequences of characters, enclosed in single or double quotes.

Python

```
name = "Alice"
message = 'Hello, world!'
print(type(name)) # <class 'str'>
```

- **bool (Booleans):** Represent truth values, either True or False.

Python

```
is_active = True
is_admin = False
print(type(is_active)) # <class 'bool'>
```

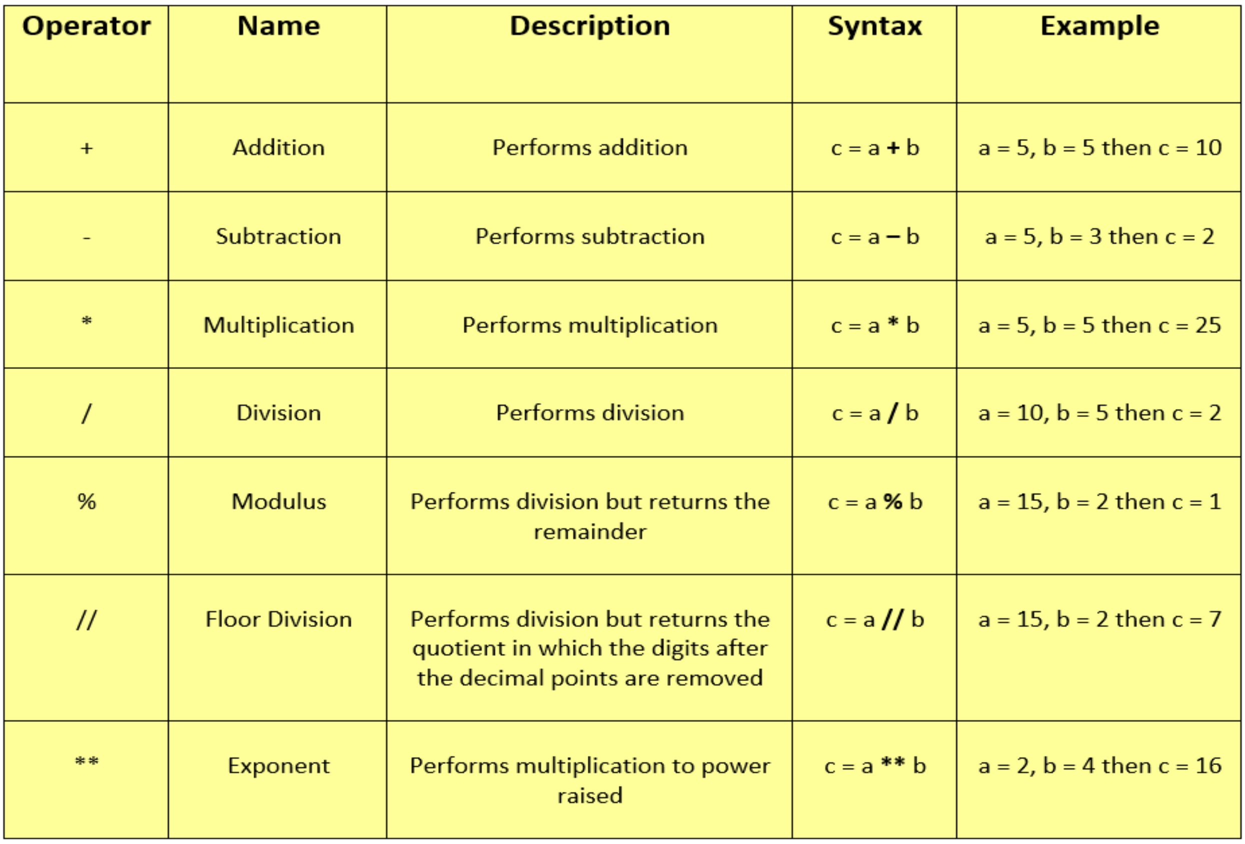
- **NoneType:** Represents the absence of a value.

Python

```
result = None
print(type(result)) # <class 'NoneType'>
```

Beneficial Visuals:

- **Table:** A table summarizing each data type with its name, description, and an example.
- **Diagram:** A conceptual diagram showing how different data types categorize various kinds of information.



Operator	Name	Description	Syntax	Example
+	Addition	Performs addition	c = a **+** b	a = 5, b = 5 then c = 10
-	Subtraction	Performs subtraction	c = a **–** b	a = 5, b = 3 then c = 2
*	Multiplication	Performs multiplication	c = a ***** b	a = 5, b = 5 then c = 25
/	Division	Performs division	c = a **/** b	a = 10, b = 5 then c = 2
%	Modulus	Performs division but returns the remainder	c = a **%** b	a = 15, b = 2 then c = 1
//	Floor Division	Performs division but returns the quotient in which the digits after the decimal points are removed	c = a **//** b	a = 15, b = 2 then c = 7
**	Exponent	Performs multiplication to power raised	c = a ****** b	a = 2, b = 4 then c = 16

5. Operators

Operators are special symbols that perform operations on one or more operands.

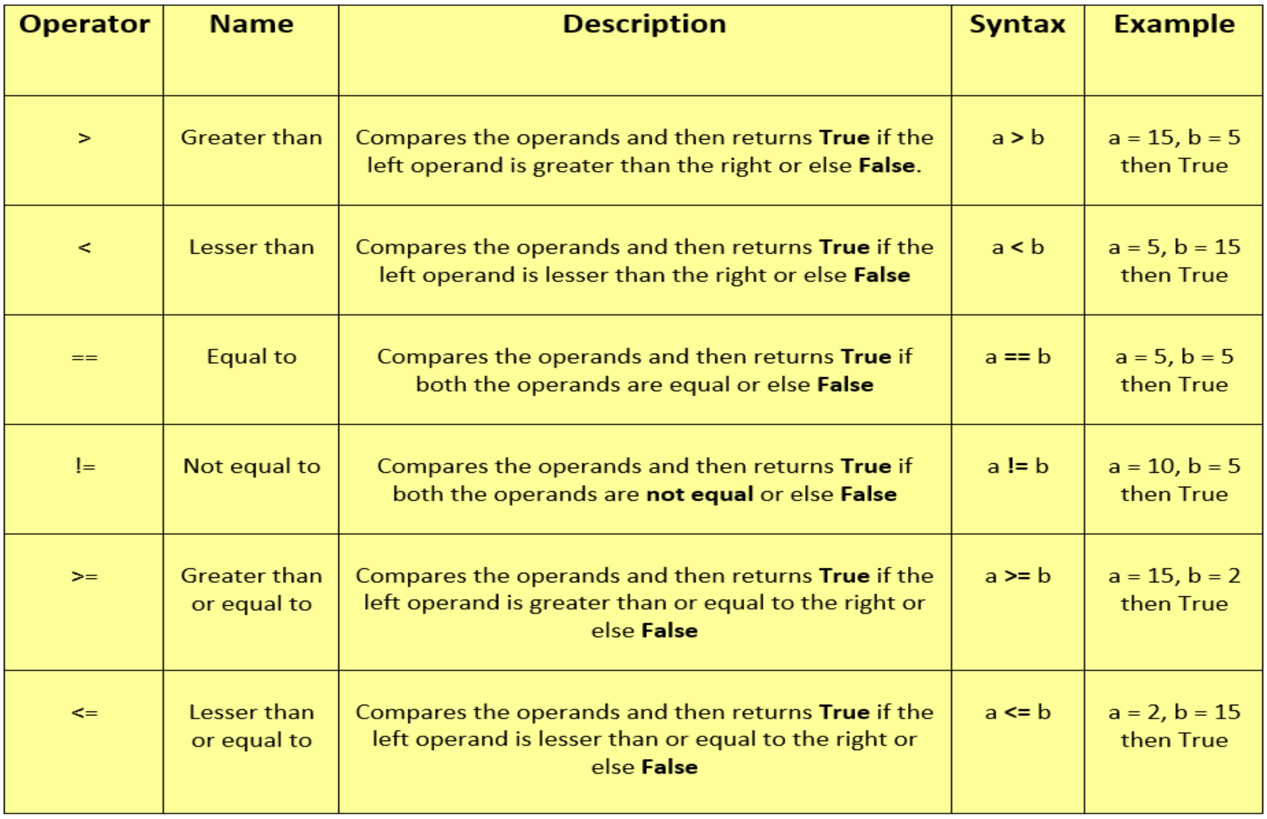
Types of Operators:

- **Arithmetic Operators:** Perform mathematical calculations.
    - + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulo), ** (Exponentiation), // (Floor Division)

<!-- end list -->

Python

```
a = 10

b = 3

print(f"Addition: {a + b}")    # 13

print(f"Division: {a / b}")    # 3.333...

print(f"Modulo: {a % b}")      # 1

print(f"Exponentiation: {a ** b}") # 1000

print(f"Exponentiation: {a ** b}")  # 1000 (10 raised to the power 3)

print(f"Floor Division: {a // b}") # 3
```



Operator	Name	Description	Syntax	Example
>	Greater than	Compares the operands and then returns **True** if the left operand is greater than the right or else **False**.	a > b	a = 15, b = 5 then True
<	Lesser than	Compares the operands and then returns **True** if the left operand is lesser than the right or else **False**	a < b	a = 5, b = 15 then True
==	Equal to	Compares the operands and then returns **True** if both the operands are equal or else **False**	a == b	a = 5, b = 5 then True
!=	Not equal to	Compares the operands and then returns **True** if both the operands are **not equal** or else **False**	a != b	a = 10, b = 5 then True
>=	Greater than or equal to	Compares the operands and then returns **True** if the left operand is greater than or equal to the right or else **False**	a >= b	a = 15, b = 2 then True
<=	Lesser than or equal to	Compares the operands and then returns **True** if the left operand is lesser than or equal to the right or else **False**	a <= b	a = 2, b = 15 then True

- **Comparison Operators:** Compare two values and return True or False.
    - == (Equal to), != (Not equal to), > (Greater than), < (Less than), >= (Greater than or equal to), <= (Less than or equal to)

<!-- end list -->

Python

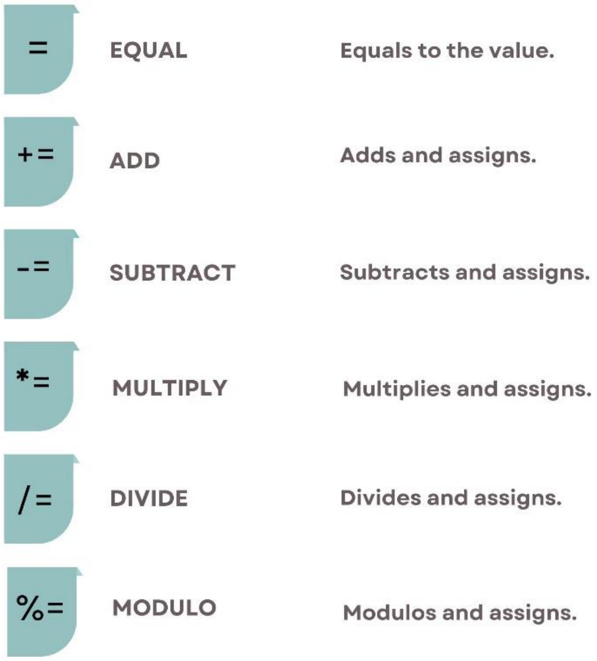
```
x = 5

y = 10

print(f"Is x equal to y? {x == y}") # False

print(f"Is x less than y? {x < y}") # True
```

- **Assignment Operators:** Assign values to variables.
    - = (Assign), += (Add and assign), -= (Subtract and assign), etc.

<!-- end list -->

Python

```
count = 0

count += 1  # count is now 1 (count = count + 1)

print(f"Count: {count}") # 1
```



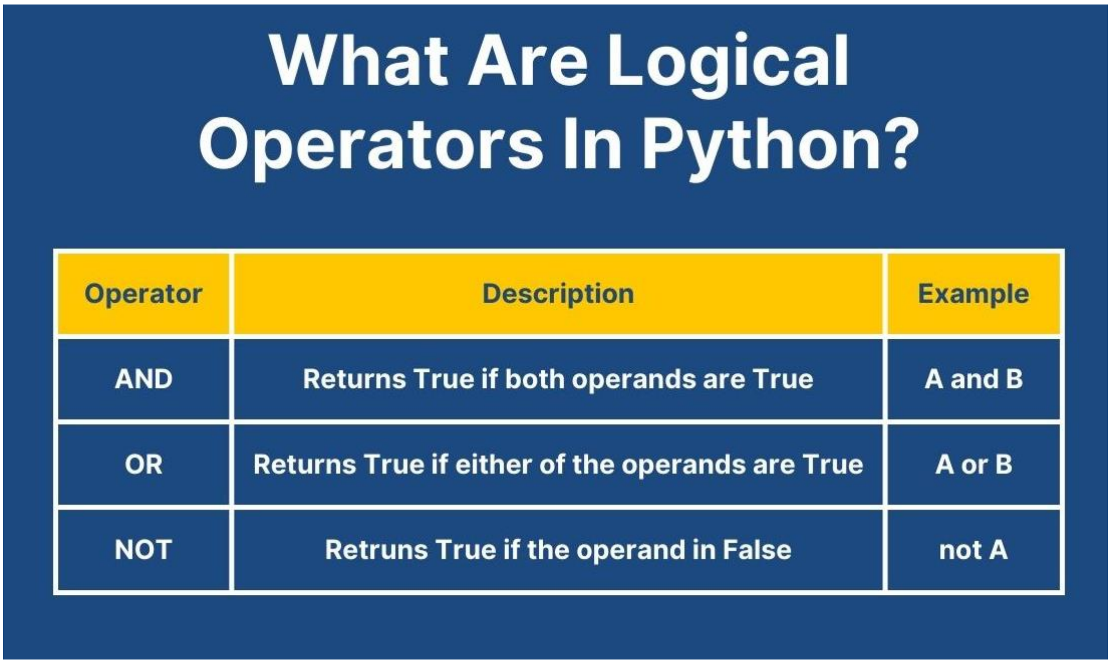
=	EQUAL	Equals to the value.
+=	ADD	Adds and assigns.
-=	SUBTRACT	Subtracts and assigns.
*=	MULTIPLY	Multiplies and assigns.
/=	DIVIDE	Divides and assigns.
%=	MODULO	Modulos and assigns.

- **Logical Operators:** Combine conditional statements (and, or, not).

Python

```
is_sunny = True

is_warm = False
```

```
print(f"Sunny AND Warm: {is_sunny and is_warm}") # False
print(f"Sunny OR Warm: {is_sunny or is_warm}")   # True
print(f"NOT Sunny: {not is_sunny}")             # False
```



What Are Logical Operators In Python?

Operator	Description	Example
AND	Returns True if both operands are True	A and B
OR	Returns True if either of the operands are True	A or B
NOT	Retruns True if the operand in False	not A

Operator	Name	Description	Syntax	Example
is	Is	Compares two or more operands and returns **True** if they have the same id or pointing to the same memory location or else returns **False**	a **is** b	a = 5, b = 5 returns **True**
is not	Is Not	Reverse of **is.** Returns **True** if both the objects have different id's or not pointing to the same memory location or else returns **False**.	a **is not** b	a = 6, b = 5 returns **True**

- **Identity Operators:** Check if two variables refer to the same object (is, is not).

Python

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1
print(f"list1 is list2: {list1 is list2}")   # False (different objects, same content)
print(f"list1 is list3: {list1 is list3}")   # True (same object)
```

- **Membership Operators:** Check if a sequence contains a specific value (in, not in).

Python

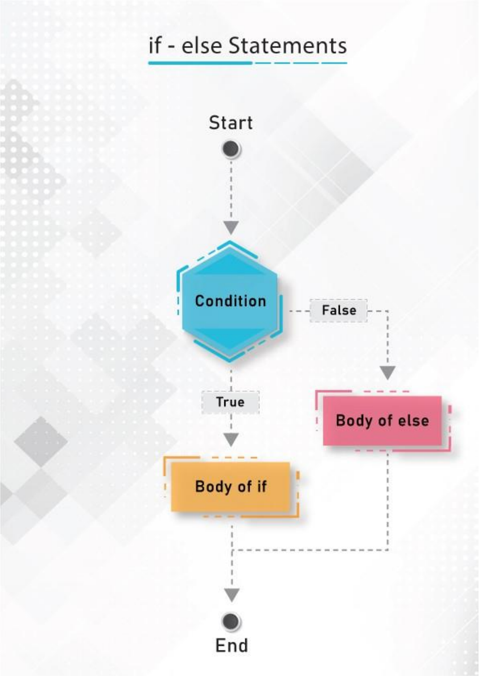
```
fruits = ["apple", "banana", "cherry"]
print(f"'banana' in fruits: {'banana' in fruits}")     # True
print(f"'grape' not in fruits: {'grape' not in fruits}") # True
```

Beneficial Visuals:

- **Table:** A table for each operator type, listing the operator, its description, and a small example.
- **Mind Map:** A central "Operators" node with branches for each type, and sub-branches for specific operators.

6. Conditional Statements: if, elif, else, nested if

Conditional statements allow your program to make decisions and execute different blocks of code based on certain conditions.



if statement: Executes a block of code if a condition is True.

Python

```
temperature = 25
if temperature > 20:
   print("It's a warm day!")
```

if-else statement: Executes one block if the condition is True and another if it's False.

Python

```
age = 17
```

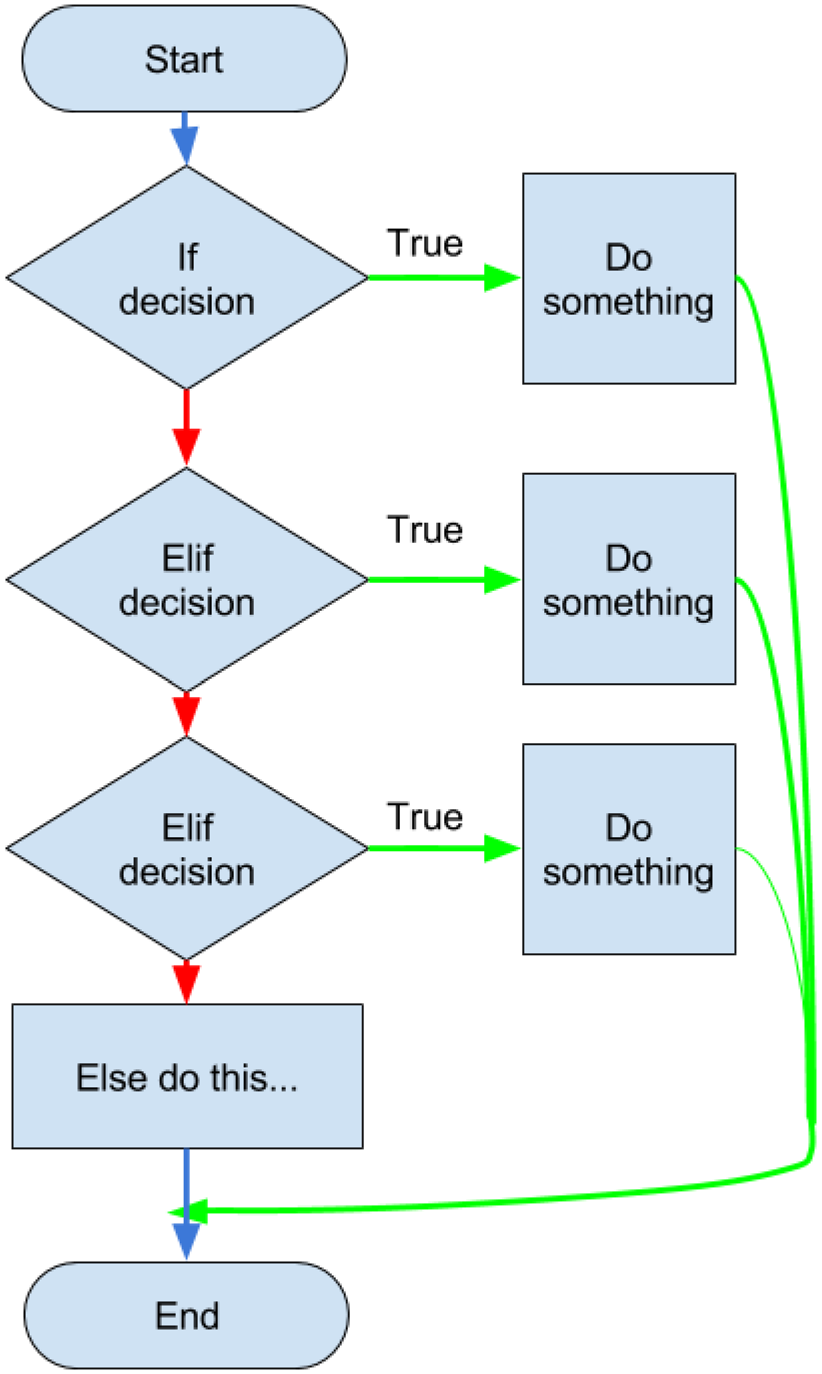
```
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

if-elif-else statement: Allows checking multiple conditions sequentially.

Python

```
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

```
# Conditional Statements Example
x = 20
if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is equal to 10")
else:
    print("x is less than 10")
```



Nested if statement: An if statement inside another if statement.

Python

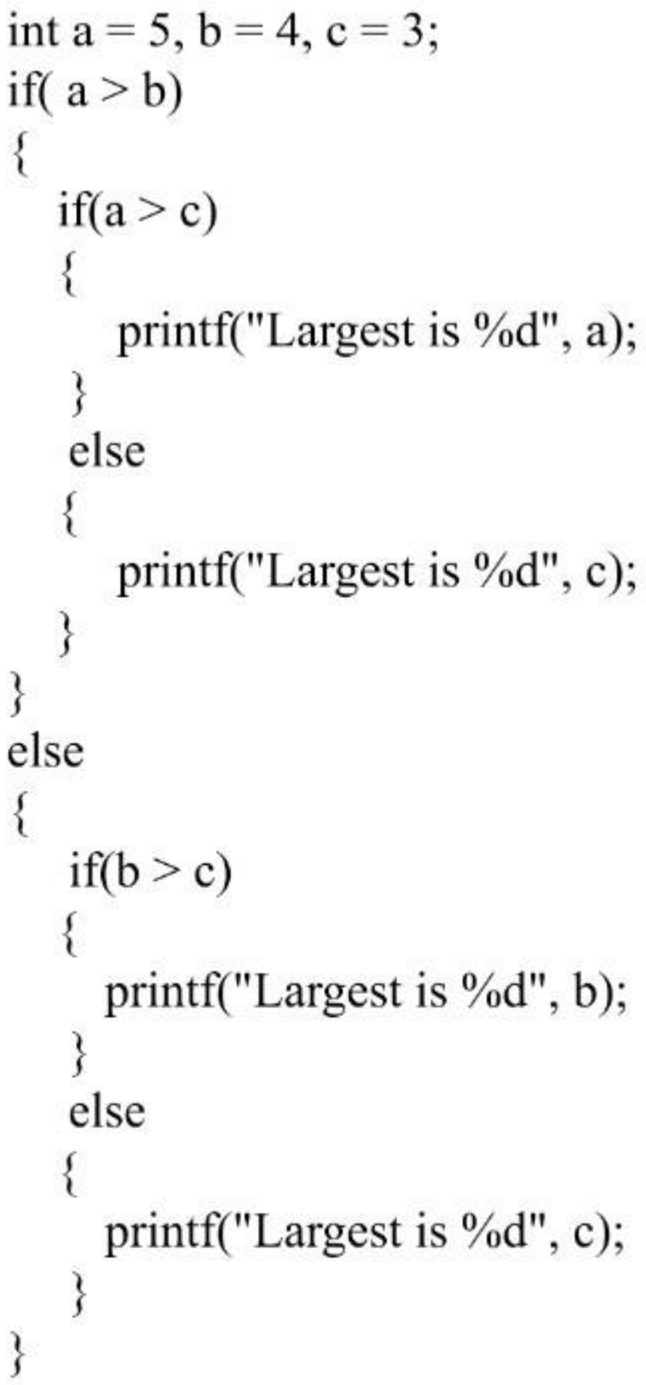
is_logged_in = True

has_permission = False

```
if is_logged_in:
    print("User is logged in.")
    if has_permission:
        print("User has access to privileged features.")
    else:
        print("User does not have sufficient permissions.")
else:
    print("Please log in to continue.")

# Nested if Example
age = 18
if age >= 18:
    if age < 60:
        print("You are an adult.")
    else:
        print("You are a senior citizen.")
else:
    print("You are a minor.")
```

Beneficial Visuals:

- **Flowcharts:** Flowcharts for if, if-else, and if-elif-else clearly showing the decision points and execution paths.
- **Indentation Diagram:** A visual representation highlighting the importance of indentation in Python to define code blocks within conditional statements.

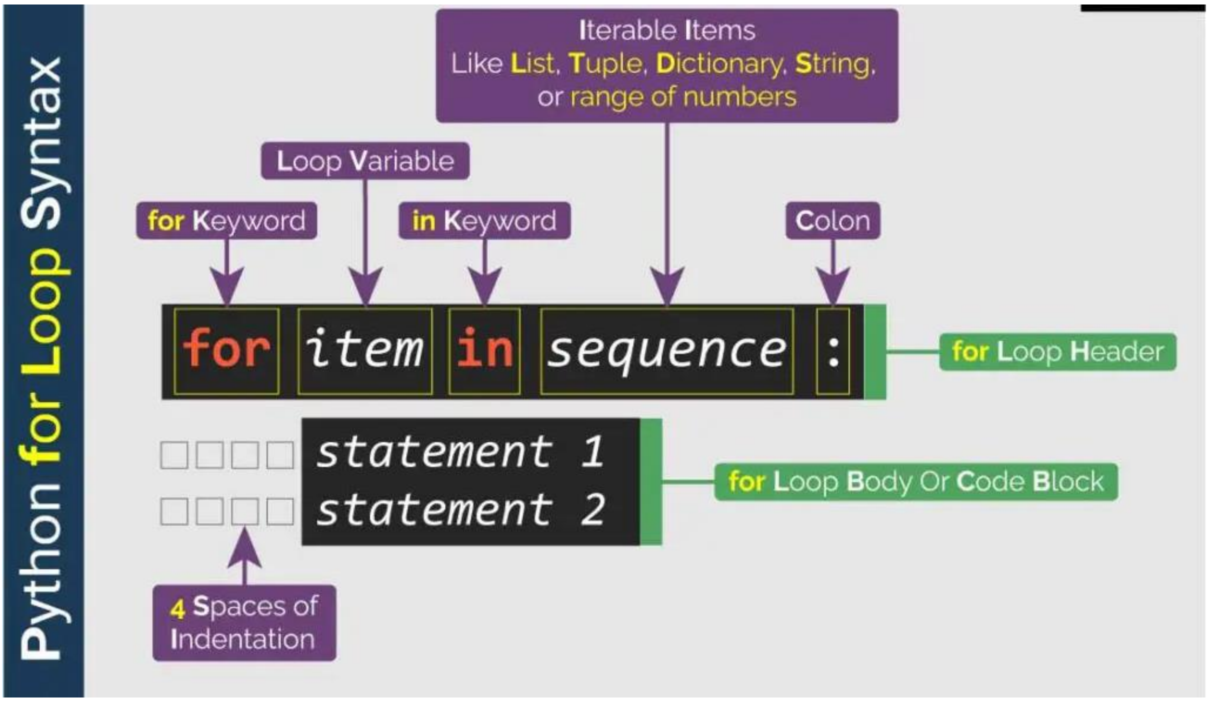


```
int a = 5, b = 4, c = 3;
if( a > b)
{
   if(a > c)
   {
      printf("Largest is %d", a);
    }
    else
    {
      printf("Largest is %d", c);
    }
}
else
{
    if(b > c)
    {
      printf("Largest is %d", b);
    }
    else
    {
      printf("Largest is %d", c);
    }
}
```

7. Loops: for, while

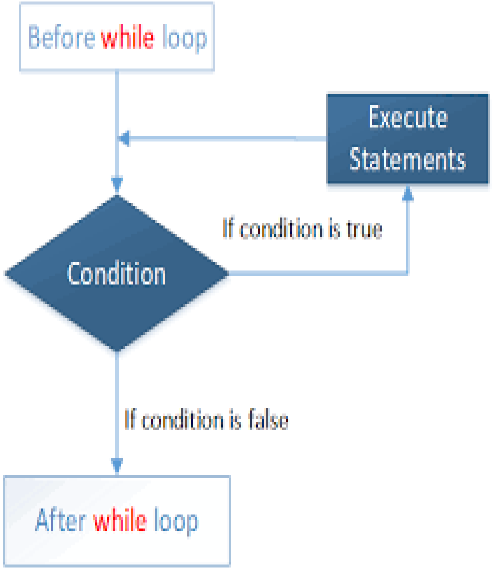
Loops are used to repeatedly execute a block of code.

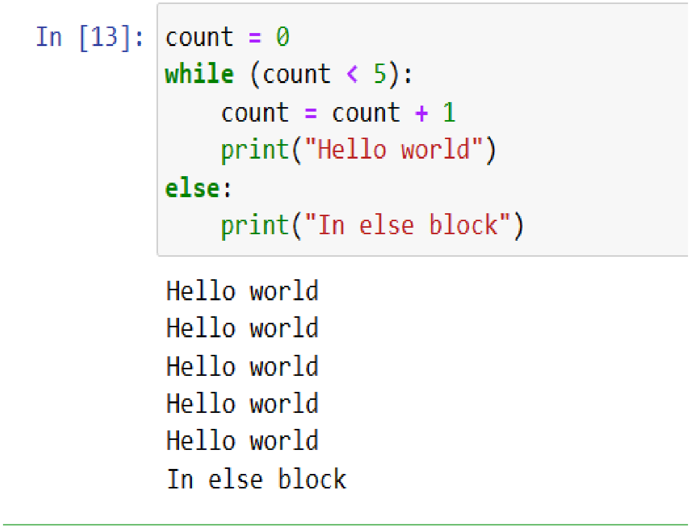
for loop: Used for iterating over a sequence (like a list, tuple, string, or range).



Python

```
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
print("Iterating through fruits:")
for fruit in fruits:
    print(fruit)

# Iterating using range()
print("\nNumbers from 0 to 4:")
for i in range(5): # range(5) generates 0, 1, 2, 3, 4
    print(i)

# Iterating with index
print("\nFruits with index:")
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```



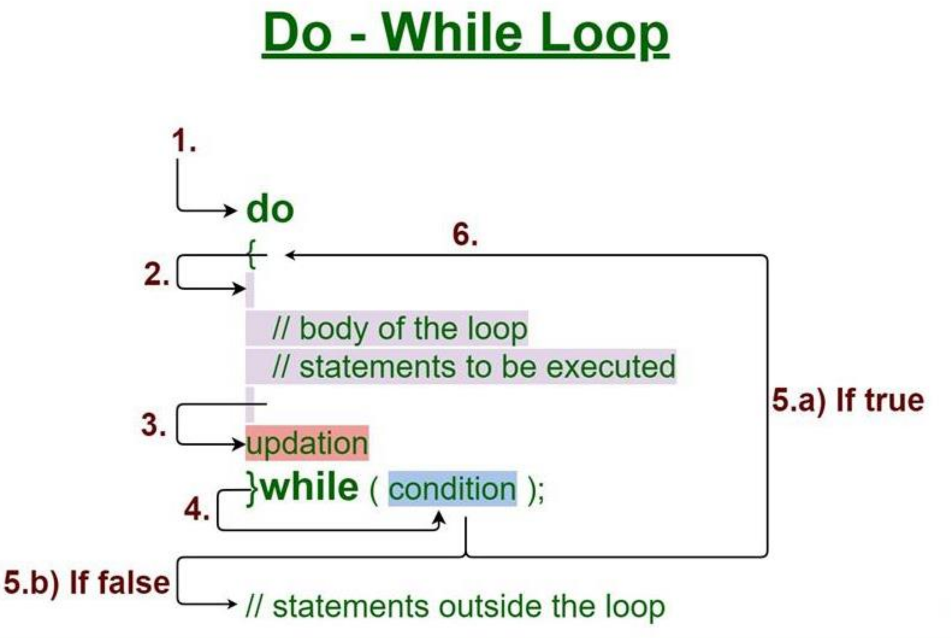


```
In [13]: count = 0
         while (count < 5):
             count = count + 1
             print("Hello world")
         else:
             print("In else block")

         Hello world
         Hello world
         Hello world
         Hello world
         Hello world
         In else block
```

while loop: Executes a block of code as long as a condition is True.

Python

```
count = 0

print("\nCounting up to 3:")

while count < 3:

  print(count)

  count += 1 # Increment count to eventually make the condition False
```



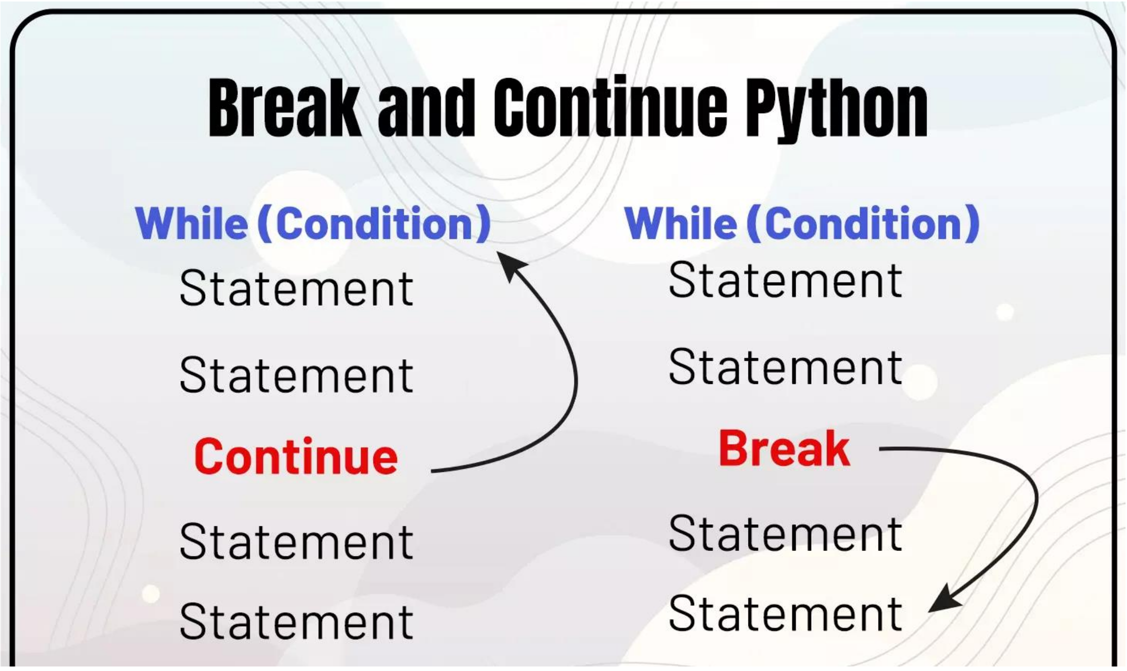
do-while (Emulation in Python): Python does not have a direct do-while loop. It can be emulated using a while True loop with a break statement.

Python

```
print("\nDo-while emulation:")

secret_number = 7

guess = 0

while True:

  try:

    guess = int(input("Guess the secret number (1-10): "))

    if guess == secret_number:

      print("Congratulations! You guessed it.")

      break # Exit the loop

    else:

      print("Try again!")

  except ValueError:

    print("Invalid input. Please enter a number.")
```

Beneficial Visuals:

- **Flowcharts:** Flowcharts for for and while loops, illustrating the iteration process and condition checking.
- **Diagram:** A visual representation of range() and how it generates numbers for for loops.



8. Jumping Statements: break, continue

Jumping statements alter the flow of execution within loops.

break statement: Terminates the loop entirely and transfers control to the statement immediately following the loop.

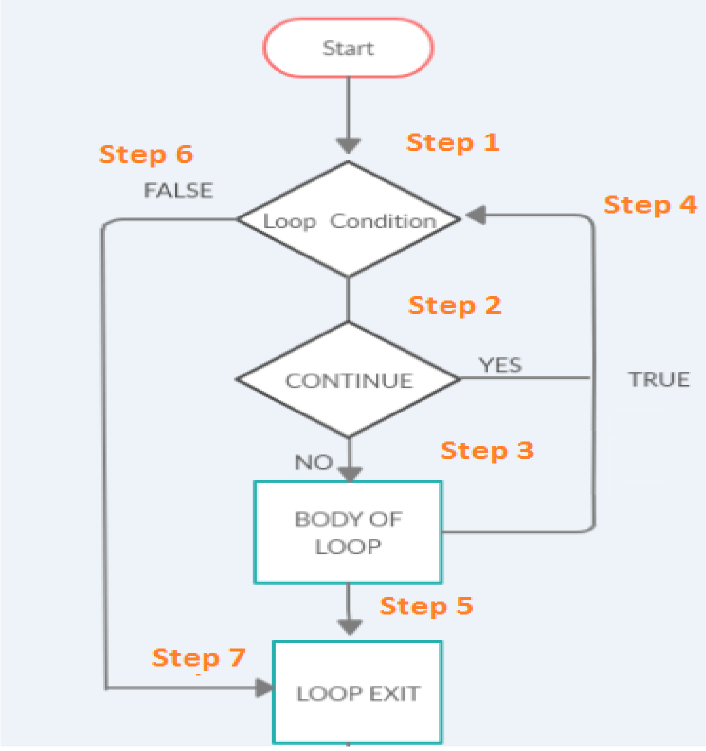
Python

```
print("Using 'break':")
for i in range(10):
   if i == 5:
       break # Loop will stop when i is 5
   print(i) # Prints 0, 1, 2, 3, 4
```

continue statement: Skips the rest of the current iteration of the loop and moves to the next iteration.

Python

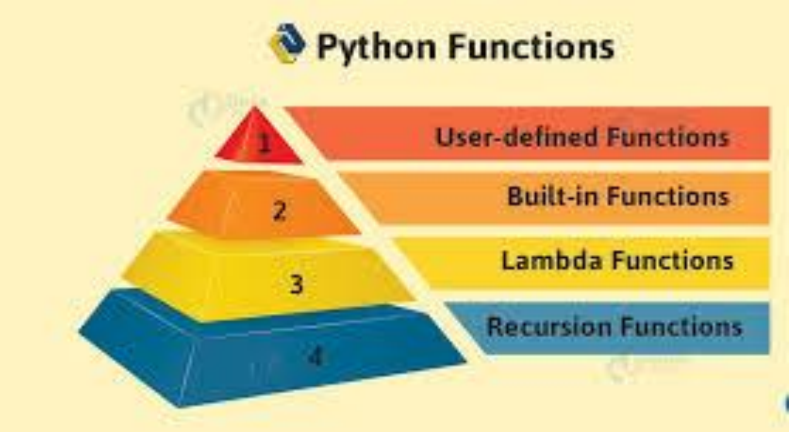
```
print("\nUsing 'continue':")
for i in range(10):
   if i % 2 == 0: # If i is even
       continue # Skip the print statement for even numbers
   print(i) # Prints 1, 3, 5, 7, 9
```

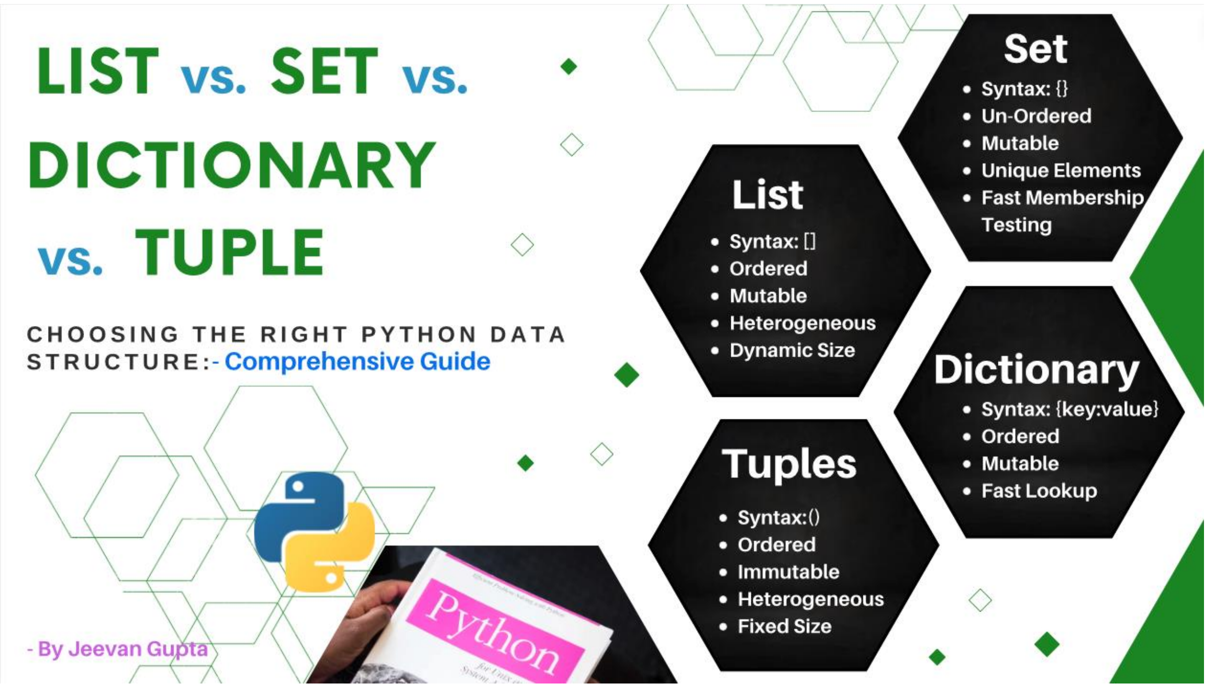


Beneficial Visuals:

- **Flowcharts:** Flowcharts showing how break and continue change the flow within a loop.
- **Trace Table:** A simple table showing the value of a loop variable and the effect of break/continue at each step.

9. Data Structures: list, tuple, set, dictionary





Python provides several built-in data structures to organize and store collections of data.

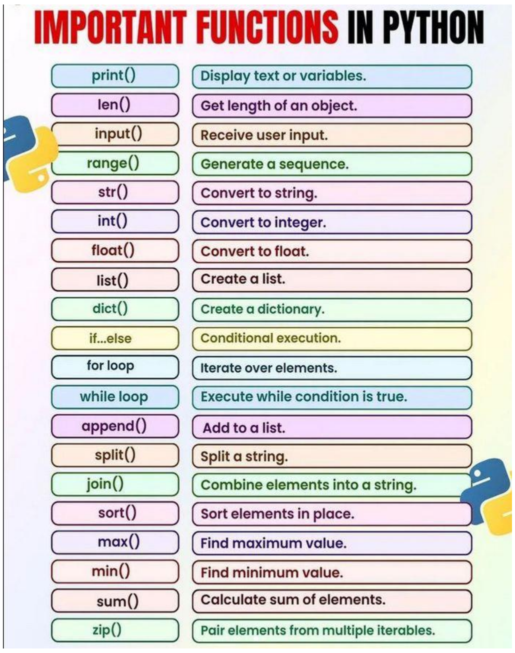
list

- **Ordered:** Elements have a defined order.
- **Mutable:** Elements can be changed (added, removed, modified).
- **Allows duplicates:** Can contain multiple identical elements.
- Defined using square brackets [].

<!-- end list -->

Python

```
# Creating a list

my_list = [1, 2, 3, "apple", True]

print(f"Original list: {my_list}")


# Accessing elements (by index)

print(f"First element: {my_list[0]}") # 1


# Modifying an element

my_list[1] = 20

print(f"Modified list: {my_list}") # [1, 20, 3, 'apple', True]


# Adding elements

my_list.append("banana")

print(f"List after append: {my_list}") # [1, 20, 3, 'apple', True, 'banana']


# Removing elements

my_list.remove("apple")

print(f"List after remove: {my_list}") # [1, 20, 3, True, 'banana']
```



IMPORTANT FUNCTIONS IN PYTHON

print()	Display text or variables.
len()	Get length of an object.
input()	Receive user input.
range()	Generate a sequence.
str()	Convert to string.
int()	Convert to integer.
float()	Convert to float.
list()	Create a list.
dict()	Create a dictionary.
if...else	Conditional execution.
for loop	Iterate over elements.
while loop	Execute while condition is true.
append()	Add to a list.
split()	Split a string.
join()	Combine elements into a string.
sort()	Sort elements in place.
max()	Find maximum value.
min()	Find minimum value.
sum()	Calculate sum of elements.
zip()	Pair elements from multiple iterables.

tuple

- **Ordered:** Elements have a defined order.
- **Immutable:** Elements cannot be changed after creation.
- **Allows duplicates:** Can contain multiple identical elements.
- Defined using parentheses ().

<!-- end list -->

Python

```
# Creating a tuple
my_tuple = (10, 20, 30, "orange", False)
print(f"Original tuple: {my_tuple}")

# Accessing elements
print(f"Second element: {my_tuple[1]}") # 20

# Attempting to modify (will raise an error)
# my_tuple[0] = 100 # TypeError: 'tuple' object does not support item assignment
```

set

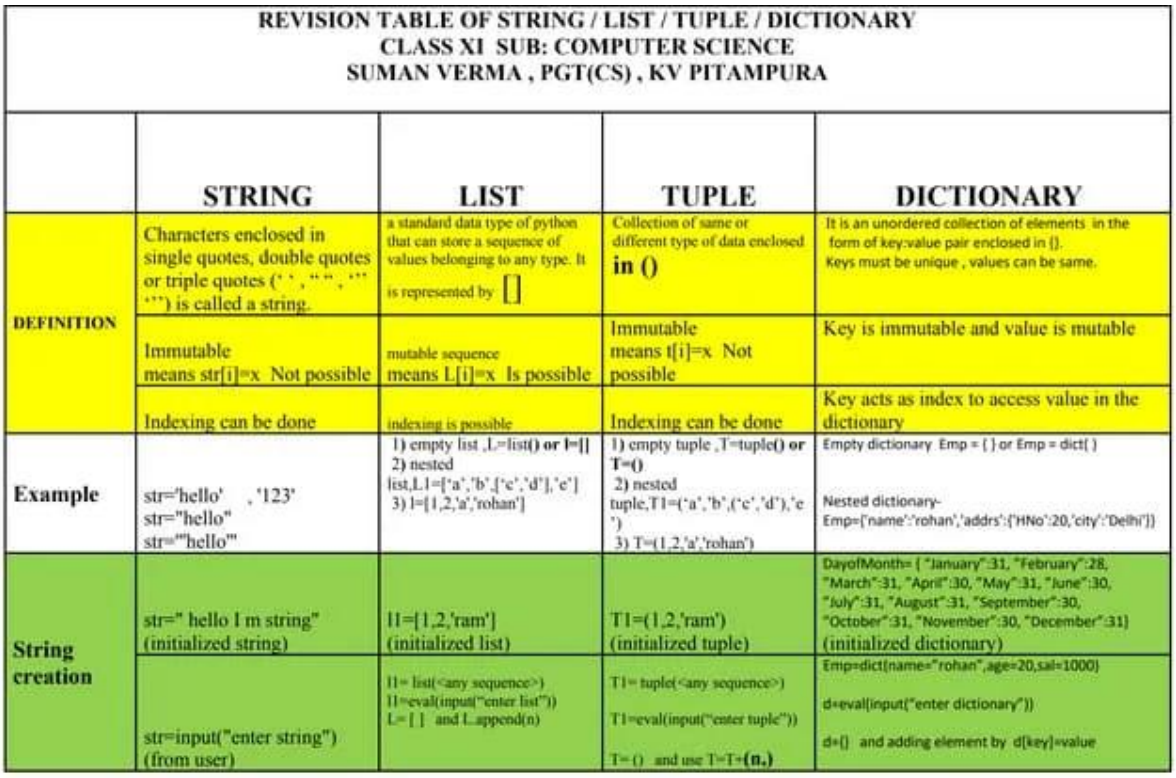
- **Unordered:** Elements do not have a defined order (cannot be accessed by index).
- **Mutable:** Elements can be added or removed, but individual elements within the set are immutable (e.g., you can't have a list inside a set directly, but you can add/remove elements).
- **Does NOT allow duplicates:** Automatically removes duplicate elements.
- Defined using curly braces {} (or set() for an empty set).

<!-- end list -->

Python

```
# Creating a set
my_set = {1, 2, 3, 2, 4, "hello"}
print(f"Original set (duplicates removed): {my_set}") # {1, 2, 3, 4, 'hello'} (order might vary)

# Adding elements
my_set.add(5)
print(f"Set after adding 5: {my_set}")
```

```
# Removing elements
my_set.remove(2)
print(f"Set after removing 2: {my_set}")

# Set operations
set_a = {1, 2, 3}
set_b = {3, 4, 5}
print(f"Union: {set_a.union(set_b)}")        # {1, 2, 3, 4, 5}
print(f"Intersection: {set_a.intersection(set_b)}") # {3}
```



REVISION TABLE OF STRING / LIST / TUPLE / DICTIONARY
CLASS XI SUB: COMPUTER SCIENCE
SUMAN VERMA , PGT(CS) , KV PITAMPURA

	STRING	LIST	TUPLE	DICTIONARY
DEFINITION	Characters enclosed in single quotes, double quotes or triple quotes (' ' , " " , ''' ''') is called a string.	a standard data type of python that can store a sequence of values belonging to any type. It is represented by []	Collection of same or different type of data enclosed **in ()**	It is an unordered collection of elements in the form of key:value pair enclosed in {}. Keys must be unique , values can be same.
	Immutable means str[i]=x Not possible	mutable sequence means L[i]=x Is possible	Immutable means t[i]=x Not possible	Key is immutable and value is mutable
	Indexing can be done	indexing is possible	Indexing can be done	Key acts as index to access value in the dictionary

Example	str='hello' , '123'
	str="hello"
	str='''hello'''
		1) empty list ,L=list() **or** l=[]
		2) nested list,L1=['a','b',['c','d'],'e']
		3) l=[1,2,'a','rohan']
			1) empty tuple ,T=tuple() **or** T=()
			2) nested tuple,T1=('a','b',('c','d'),'e')
			3) T=(1,2,'a','rohan')
				Empty dictionary Emp = { } or Emp = dict( )
				Nested dictionary-Emp={'name':'rohan','addrs':{'HNo':20,'city':'Delhi'}}

String creation	str=" hello I m string" (initialized string)	l1=[1,2,'ram'] (initialized list)	T1=(1,2,'ram') (initialized tuple)	DayofMonth= { "January":31, "February":28, "March":31, "April":30, "May":31, "June":30, "July":31, "August":31, "September":30, "October":31, "November":30, "December":31} (initialized dictionary)

	str=input("enter string")
	(from user)
		l1= list(<any sequence>)
		l1=eval(input("enter list"))
		L= [ ] and L.append(n)
			T1= tuple(<any sequence>)
			T1=eval(input("enter tuple"))
			T= () and use T=T+**(n,)**
				Emp=dict(name="rohan",age=20,sal=1000)
				d=eval(input("enter dictionary"))
				d={} and adding element by d[key]=value

dictionary

- **Unordered (in Python 3.7+ they maintain insertion order):** Elements are stored as key-value pairs.
- **Mutable:** Key-value pairs can be added, removed, or modified.
- **Keys must be unique and immutable:** Values can be of any type.
- Defined using curly braces {} with key-value pairs separated by colons :.

<!-- end list -->

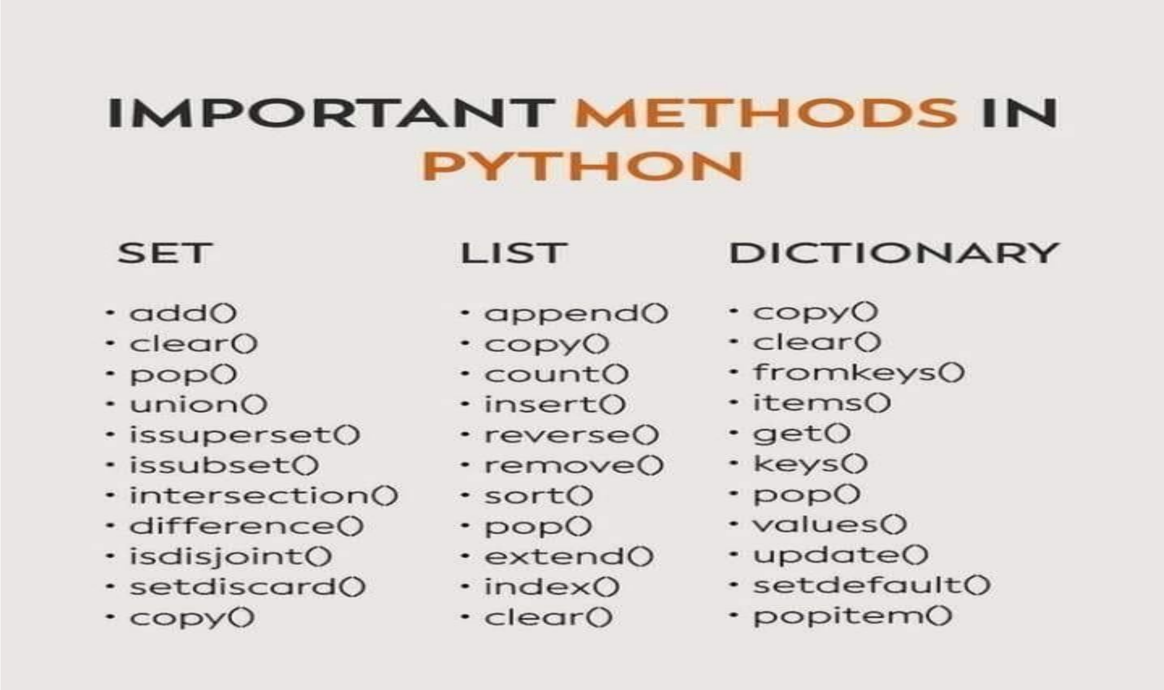
Python

```
# Creating a dictionary
person = {
   "name": "Bob",
   "age": 25,
   "city": "New York"
}
print(f"Original dictionary: {person}")

# Accessing values (by key)
print(f"Person's name: {person['name']}") # Bob

# Modifying a value
person["age"] = 26
print(f"Dictionary after age change: {person}") # {'name': 'Bob', 'age': 26, 'city': 'New York'}

# Adding a new key-value pair
person["occupation"] = "Engineer"
print(f"Dictionary after adding occupation: {person}")

# Removing a key-value pair
del person["city"]
print(f"Dictionary after removing city: {person}")

# Iterating through a dictionary
print("\nIterating through dictionary:")
for key, value in person.items():
   print(f"{key}: {value}")
```

Beneficial Visuals:

Table: A comparison table highlighting the key differences (ordered/unordered, mutable/immutable, duplicates allowed/not allowed) between lists, tuples, sets, and dictionaries.

- **Diagrams:** Visual representations of each data structure:
  - **List:** A linear sequence of boxes with indices.
  - **Tuple:** Similar to a list, but with a padlock icon to signify immutability.
  - **Set:** A Venn diagram or a collection of distinct elements with no order.
  - **Dictionary:** Key-value pairs organized like a lookup table.



IMPORTANT METHODS IN PYTHON

SET

- add()
- clear()
- pop()
- union()
- issuperset()
- issubset()
- intersection()
- difference()
- isdisjoint()
- setdiscard()
- copy()

LIST

- append()
- copy()
- count()
- insert()
- reverse()
- remove()
- sort()
- pop()
- extend()
- index()
- clear()

DICTIONARY

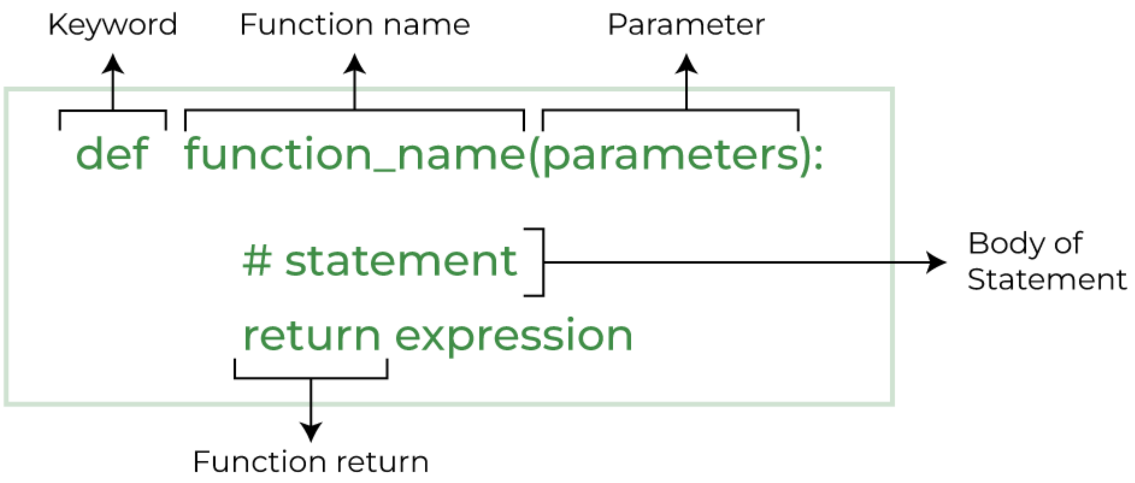
- copy()
- clear()
- fromkeys()
- items()
- get()
- keys()
- pop()
- values()
- update()
- setdefault()
- popitem()

Feature	List	Tuple	Set	Dictionary
Ordered	Yes	Yes	No	Yes (Python 3.7+)
Mutable	Yes	No	Yes	Yes
Duplicates	Allowed	Allowed	Not allowed	Keys unique
Syntax	[]	()	{}	{key: value}

Feature	**List**	**Tuple**	**Set**	**Dictionary**
Ordered	☑ Yes	☑ Yes	✗ No	☑ (Python 3.7+)
Mutable	☑ Yes	✗ No	☑ Yes	☑ Yes
Duplicates	☑ Allowed	☑ Allowed	✗ Not allowed	☑ Keys unique
Syntax	[]	()	{}	{key: value}

10. Functions

A function is a block of organized, reusable code that performs a single, related action. Functions provide better modularity for your application and a high degree of code reusing.



Defining a Function: Use the def keyword, followed by the function name, parentheses (), and a colon :.

Python

```
# Simple function
def greet():
    print("Hello from a function!")

# Calling the function
greet()
```

Parameters and Arguments:

- **Parameters:** Variables listed inside the parentheses in the function definition.
- **Arguments:** The actual values passed to the function when it is called.

<!-- end list -->

Python

```
def greet_person(name): # 'name' is a parameter
    print(f"Hello, {name}!")

greet_person("Alice") # "Alice" is an argument
greet_person("Bob")
```

Return Values: The return statement is used to send a value back to the caller of the function.

Python

```
def add_numbers(a, b):
  result = a + b
  return result # Returns the sum


sum_result = add_numbers(5, 3)
print(f"The sum is: {sum_result}") # 8


def calculate_area(length, width):
  if length <= 0 or width <= 0:
    return "Invalid dimensions" # Can return different types
  return length * width


area1 = calculate_area(10, 5)
print(f"Area 1: {area1}") # 50


area2 = calculate_area(-2, 5)
print(f"Area 2: {area2}") # Invalid dimensions
```

Default Parameters: Assign a default value to a parameter, making it optional.

Python

```
def print_message(message="Default message"):
  print(message)


print_message("This is a custom message.")
print_message() # Uses the default message
```

Arbitrary Arguments (*args and **kwargs):

- *args: Allows a function to accept an arbitrary number of non-keyword arguments (as a tuple).
- **kwargs: Allows a function to accept an arbitrary number of keyword arguments (as a dictionary).

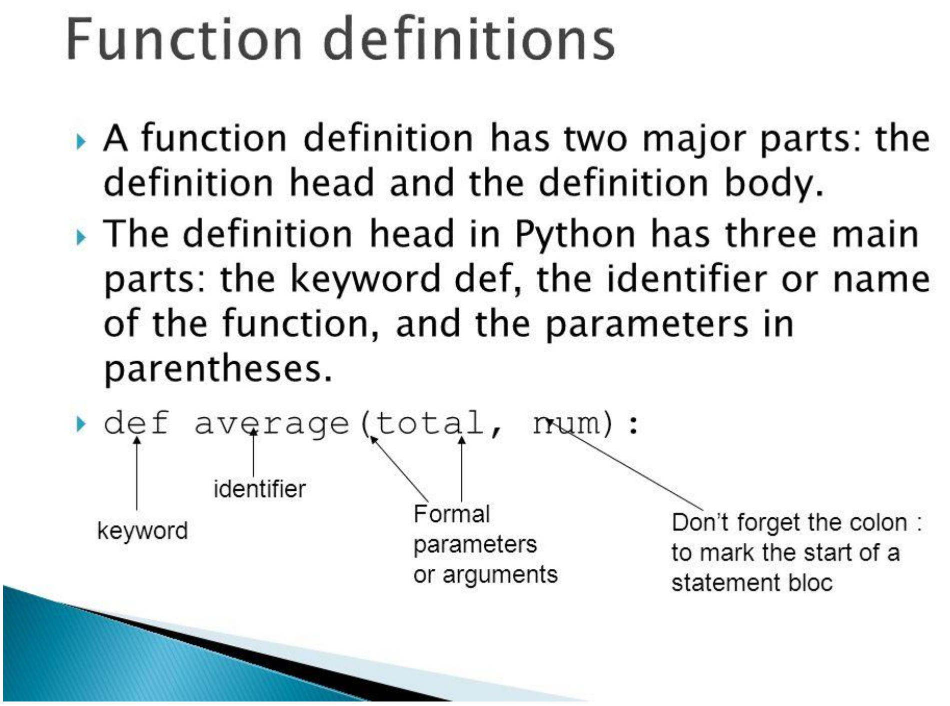
<!-- end list -->

Python

```
def print_numbers(*args):
   for num in args:
       print(num)


print_numbers(1, 2, 3, 4)


def print_info(**kwargs):
   for key, value in kwargs.items():
       print(f"{key}: {value}")


print_info(name="Charlie", age=30, city="London")
```

Beneficial Visuals:

- **Function Call Stack Diagram:** A diagram illustrating how functions are called, how arguments are passed, and how return values are sent back.
- **Input/Output Diagram:** A simple box diagram for a function showing inputs (parameters) and outputs (return values).



Function definitions

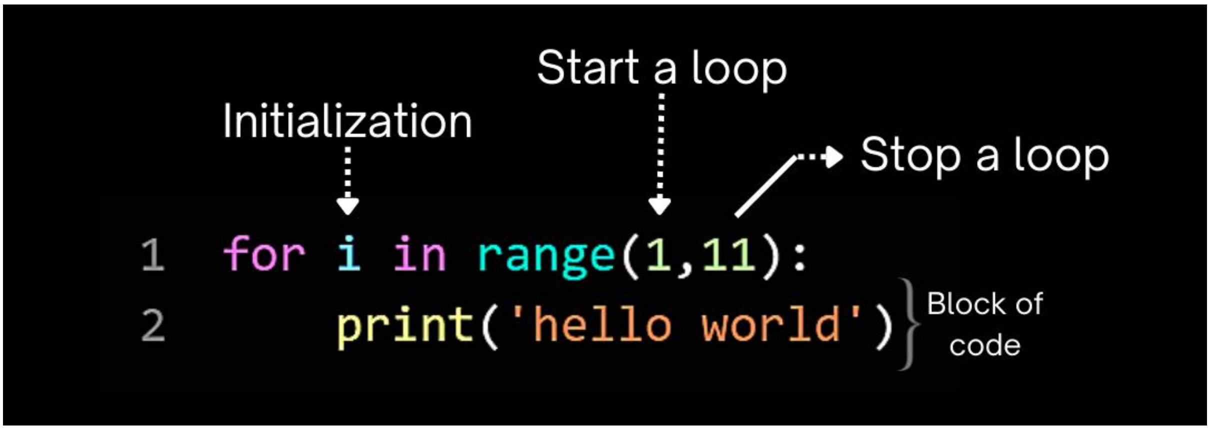
- A function definition has two major parts: the definition head and the definition body.
- The definition head in Python has three main parts: the keyword def, the identifier or name of the function, and the parameters in parentheses.
- `def average(total, num):`

keyword

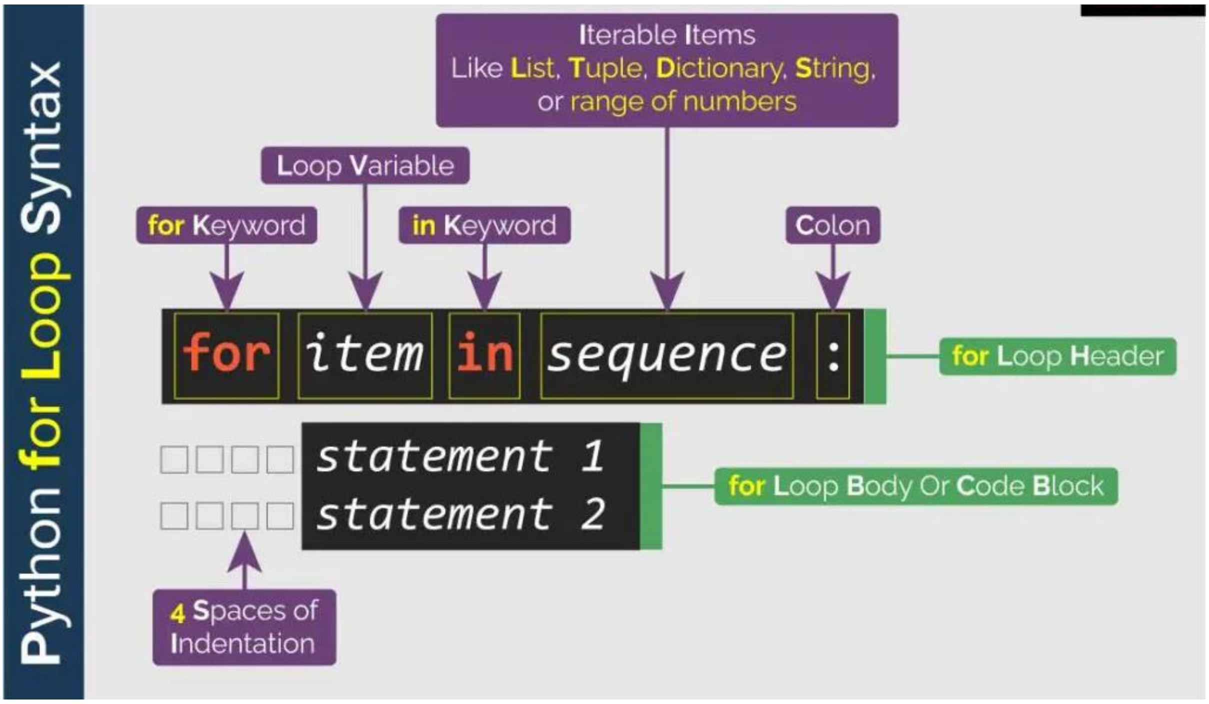
identifier

Formal parameters or arguments

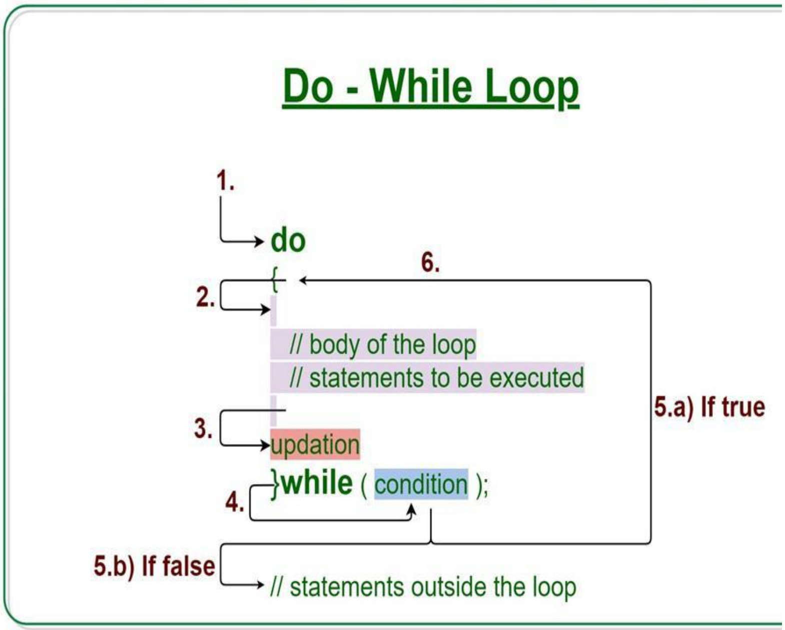
Don't forget the colon : to mark the start of a statement bloc



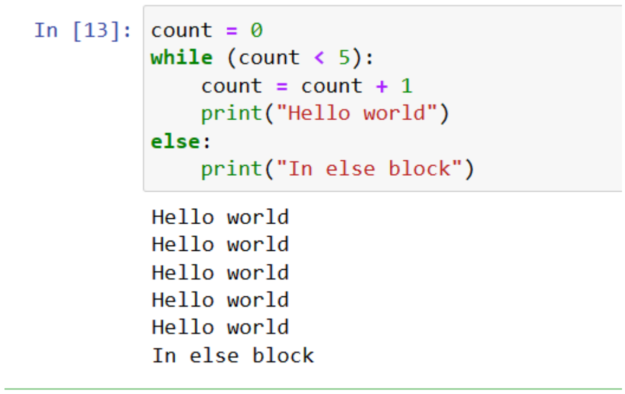
```
for i in range(1,11):
    print('hello world')
```



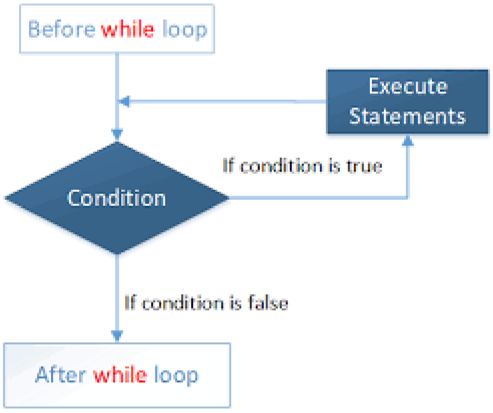
```
for item in sequence :
    statement 1
    statement 2
```



```
do
{
// body of the loop
// statements to be executed

updation
}while ( condition );
// statements outside the loop
```



```
In [13]: count = 0
         while (count < 5):
             count = count + 1
             print("Hello world")
         else:
             print("In else block")

         Hello world
         Hello world
         Hello world
         Hello world
         Hello world
         In else block
```



Recursion

Recursion is a process where a function calls itself directly or indirectly to solve a smaller instance of the same problem.

A recursive function has two main parts:

- **Base case:** The condition that stops the recursion.
- **Recursive case:** The part where the function calls itself with modified arguments.

Example: Calculating Factorial Using Recursion

The factorial of a number n (denoted n!) is the product of all positive integers less than or equal to n.

- Base case: factorial(0) = 1
- Recursive case: factorial(n) = n * factorial(n-1)

python

CopyEdit

```
def factorial(n):
  if n == 0:
    return 1  # Base case
  else:
    return n * factorial(n - 1)  # Recursive call


print(factorial(5))  # Output: 120
```

How Recursion Works (Flow)

1. factorial(5) calls factorial(4)
2. factorial(4) calls factorial(3)

3. factorial(3) calls factorial(2)
4. factorial(2) calls factorial(1)
5. factorial(1) calls factorial(0)
6. factorial(0) returns 1 (base case reached)
7. Then each call returns, multiplying the results back up:
    - factorial(1) returns 1 * 1 = 1
    - factorial(2) returns 2 * 1 = 2
    - factorial(3) returns 3 * 2 = 6
    - factorial(4) returns 4 * 6 = 24
    - factorial(5) returns 5 * 24 = 120

Recursion Tips

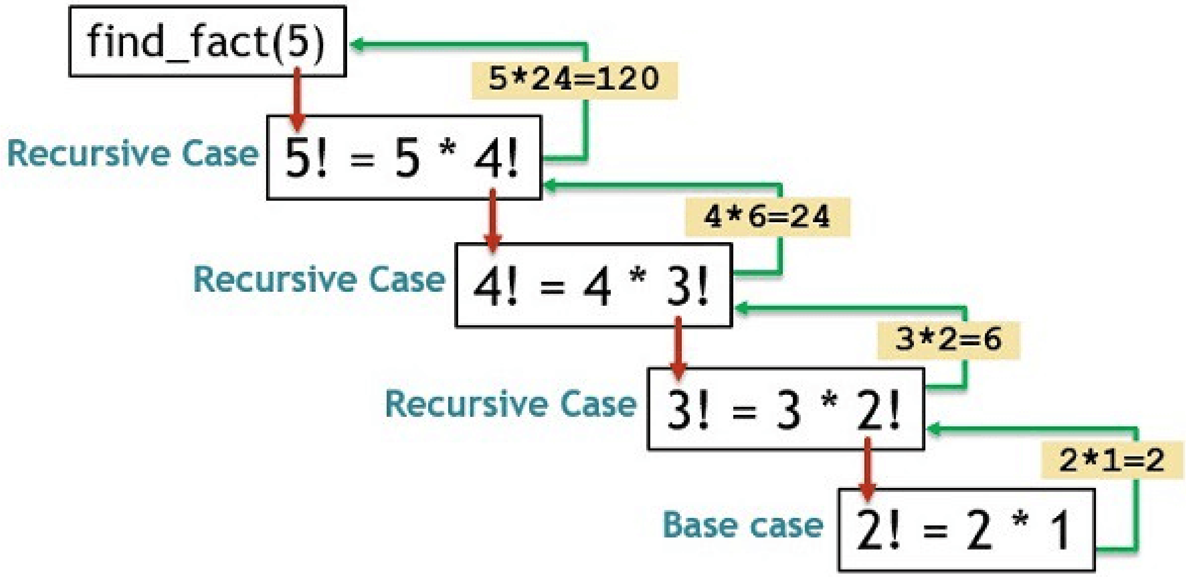
- Always ensure the base case is defined, otherwise the recursion will go infinite and cause a stack overflow.
- Recursion is useful for problems like factorial, Fibonacci sequence, tree traversals, and divide-and-conquer algorithms.

Optional: Fibonacci Series (Recursive)

python

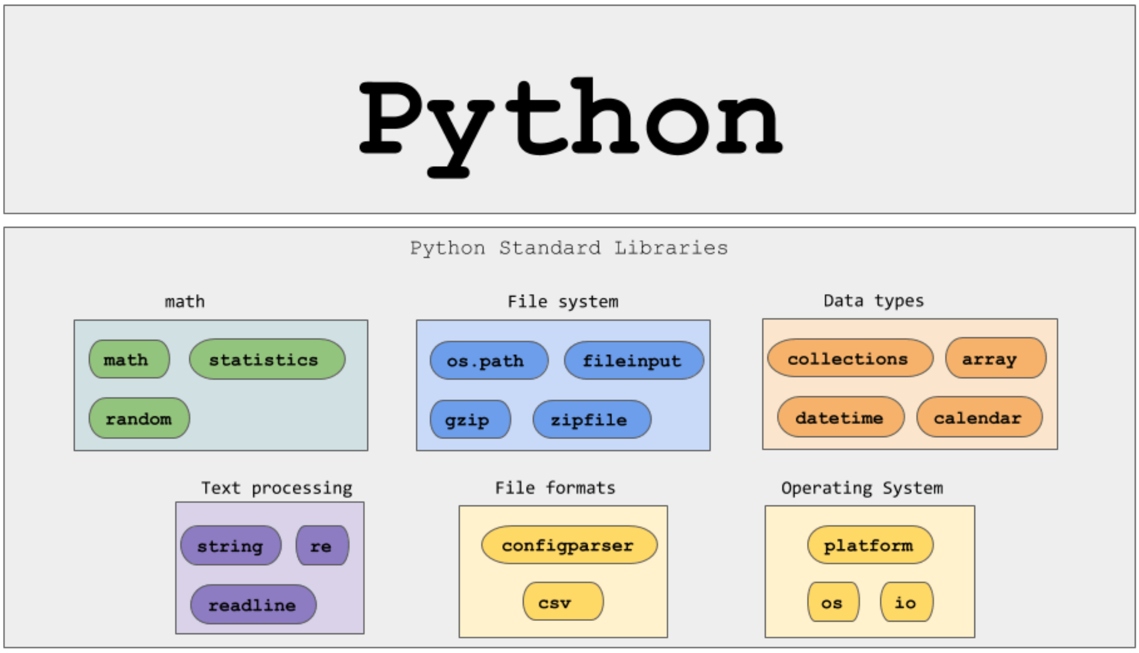
CopyEdit

```
def fibonacci(n):
    if n <= 1:
        return n  # Base case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive calls


print(fibonacci(7))  # Output: 13
```



11. Modules and Packages

Modules: A module is a Python file (.py) containing Python definitions and statements. Modules allow you to logically organize your Python code.



Creating a Module: Let's say you create a file named my_math.py:

Python

```
# my_math.py

PI = 3.14159
```

```
def add(a, b):
   return a + b


def subtract(a, b):
   return a - b


def circle_area(radius):
   return PI * radius * radius
```

Importing a Module: You can use functions and variables from other modules using the import statement.

Python

```
# main_program.py
import my_math # Imports the entire module


print(f"Pi from module: {my_math.PI}")
print(f"Sum: {my_math.add(10, 5)}")
print(f"Area of circle with radius 5: {my_math.circle_area(5)}")


# Importing specific items
from my_math import subtract, PI
print(f"Difference: {subtract(20, 7)}")
print(f"PI directly: {PI}")


# Importing with an alias
import my_math as mm
print(f"Sum using alias: {mm.add(2, 8)}")
```

Packages: A package is a way of organizing related modules into a directory hierarchy. A package must contain a special file named __init__.py (which can be empty) to be recognized as a package.

Example Package Structure:

```
my_package/
   __init__.py
   module_a.py
```

```
  module_b.py
  sub_package/
    __init__.py
    module_c.py
```

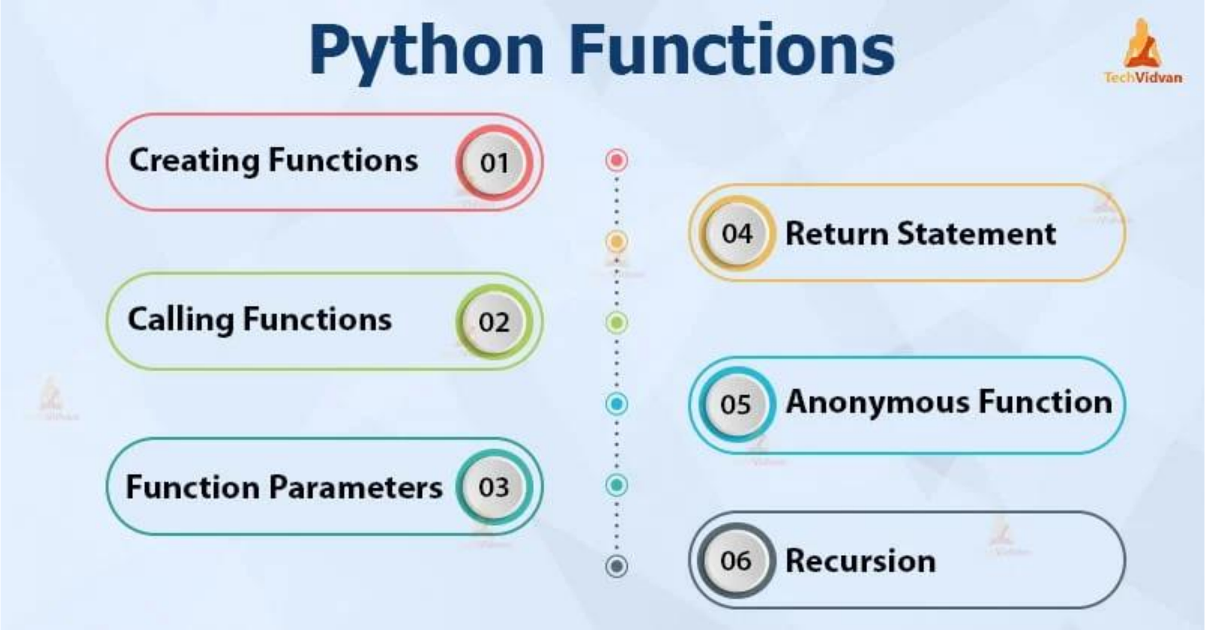
Importing from a Package:

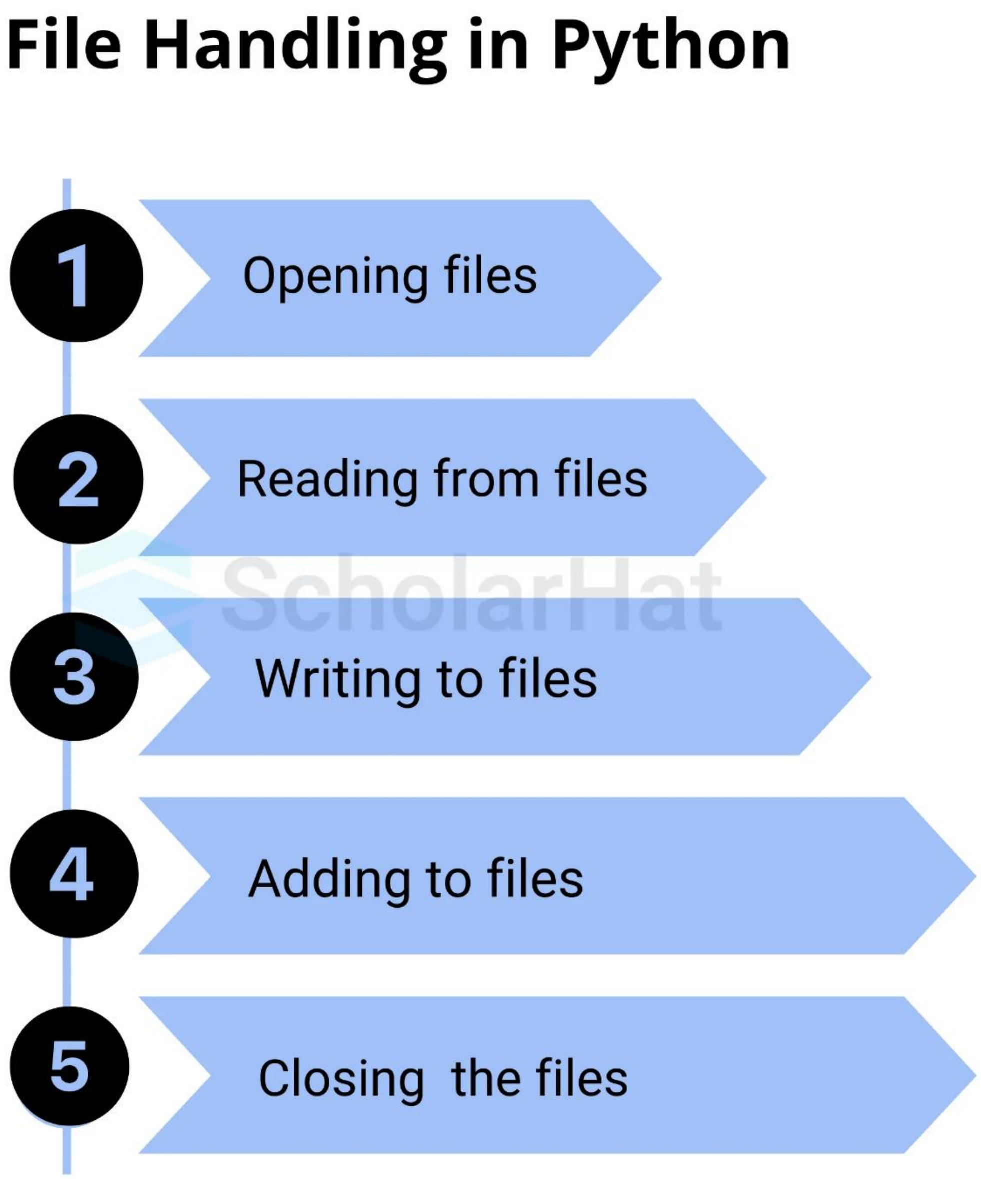
Python

```
# Assuming you are in the directory containing 'my_package'
from my_package import module_a
# Or
from my_package.sub_package import module_c


# To use functions:
# module_a.some_function()
# module_c.another_function()
```

Beneficial Visuals:

- **Module Diagram:** A simple box representing a .py file with functions/variables inside, and an arrow indicating import.
- **Package Directory Tree:** A clear directory structure diagram showing how packages and sub-packages organize modules.





12. File Handling

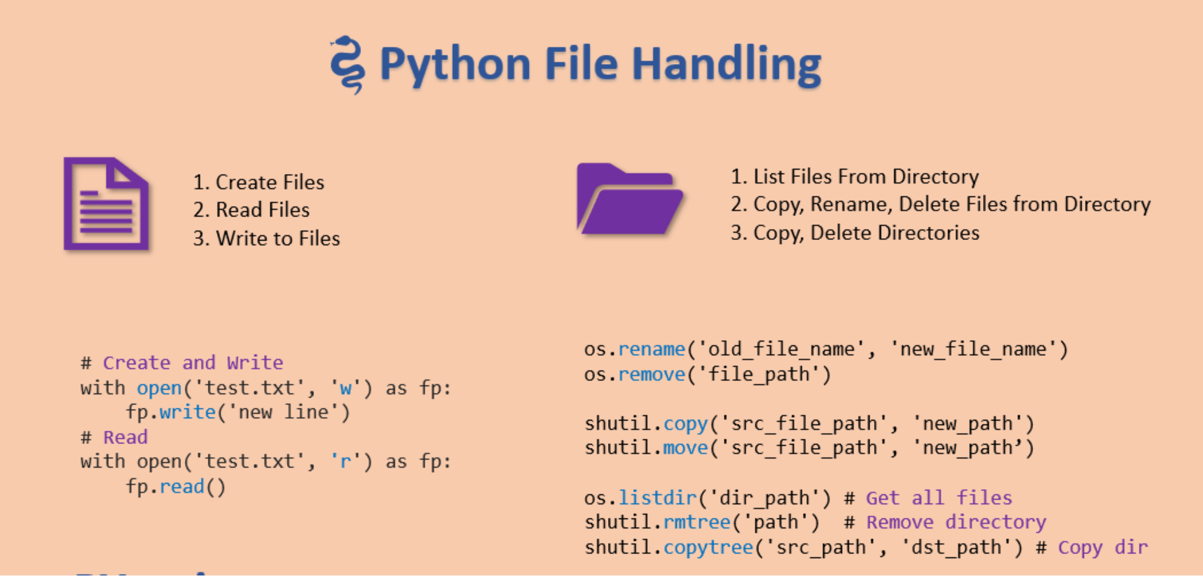
File handling allows your Python programs to interact with files on your computer's file system.

Opening a File: The open() function is used to open a file. It takes the file path and mode as arguments.

Modes:

- 'r' (read): Default mode. Opens for reading.
- 'w' (write): Opens for writing. Creates a new file if it doesn't exist, overwrites if it does.

- 'a' (append): Opens for appending. Creates a new file if it doesn't exist, appends to the end if it does.
- 'x' (create): Creates a new file. Raises an error if the file exists.
- 't' (text): Default. Opens in text mode.
- 'b' (binary): Opens in binary mode.



Python File Handling

1. Create Files
2. Read Files
3. Write to Files

1. List Files From Directory
2. Copy, Rename, Delete Files from Directory
3. Copy, Delete Directories

```
# Create and Write
with open('test.txt', 'w') as fp:
    fp.write('new line')
# Read
with open('test.txt', 'r') as fp:
    fp.read()
```

```
os.rename('old_file_name', 'new_file_name')
os.remove('file_path')

shutil.copy('src_file_path', 'new_path')
shutil.move('src_file_path', 'new_path')

os.listdir('dir_path') # Get all files
shutil.rmtree('path')  # Remove directory
shutil.copytree('src_path', 'dst_path') # Copy dir
```

Reading from a File:

Python

```
# Create a sample file first
with open("sample.txt", "w") as f:
  f.write("Hello, world!\n")
  f.write("This is a test file.\n")


# Read entire content
try:
  with open("sample.txt", "r") as file:
    content = file.read()
    print("File content (read()):")
    print(content)
except FileNotFoundError:
  print("Error: sample.txt not found.")
```

```
# Read line by line
try:
    with open("sample.txt", "r") as file:
        print("\nFile content (readlines()):")
        for line in file: # Iterates line by line efficiently
            print(line.strip()) # .strip() removes newline characters
except FileNotFoundError:
    print("Error: sample.txt not found.")
```

Writing to a File:

Python

```
# Writing (overwrites existing content)
with open("output.txt", "w") as file:
    file.write("First line.\n")
    file.write("Second line.")
print("\n'output.txt' created/overwritten.")


# Appending (adds to existing content)
with open("output.txt", "a") as file:
    file.write("\nThird line (appended).")
print("'Third line (appended)' added to 'output.txt'.")
```

Closing a File: It's crucial to close files after use to release resources. The with statement (as shown above) is the recommended way, as it automatically handles closing the file even if errors occur.

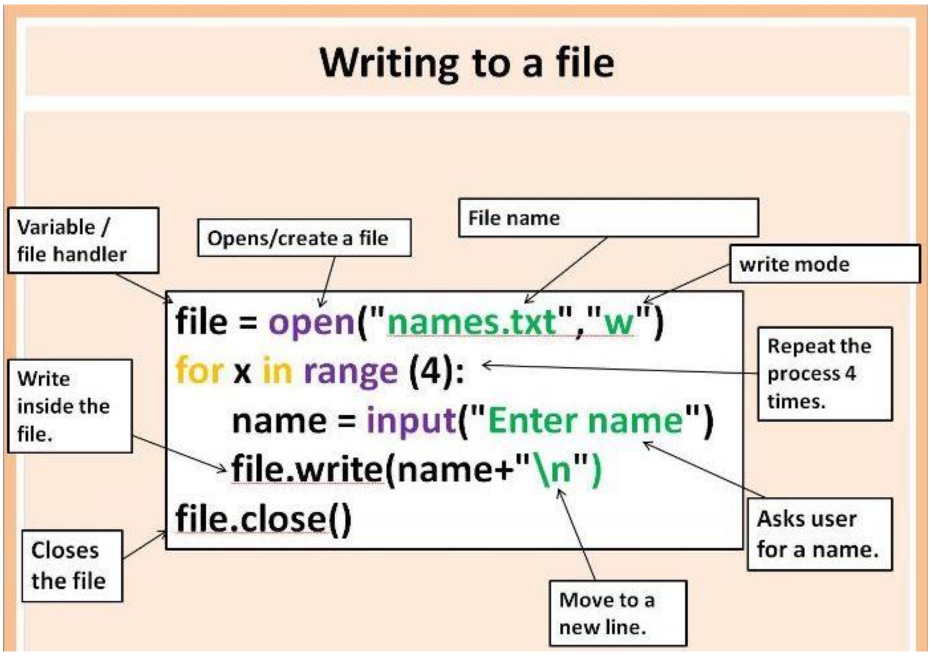
Python

```
# Manual file handling (less recommended)
file_obj = open("another.txt", "w")
file_obj.write("This is manually handled.")
file_obj.close()
print("\n'another.txt' created.")
```

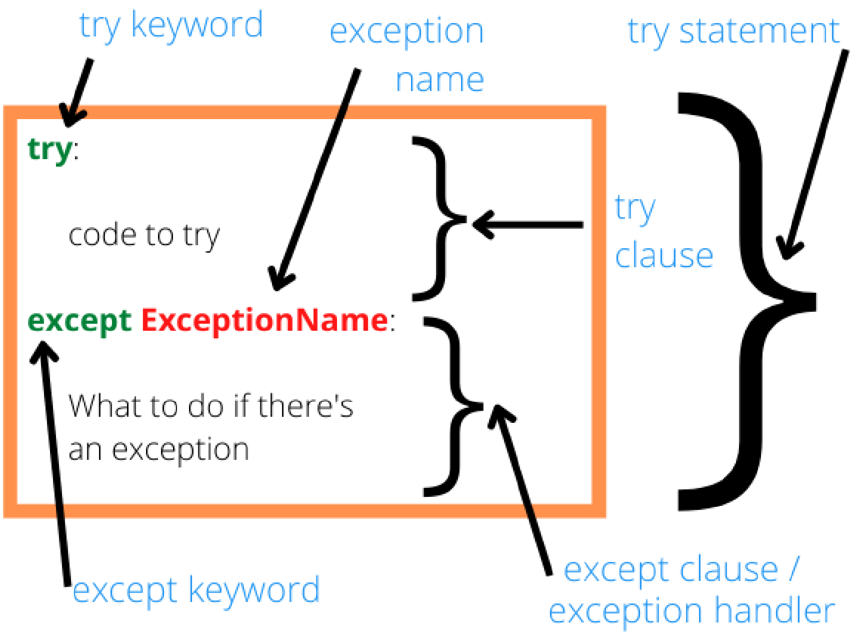
Beneficial Visuals:

- **File I/O Flowchart:** A flowchart showing the steps of opening, reading/writing, and closing a file.

- **Diagram:** A visual representation of a file on disk, with arrows showing data flow into and out of the Python program.



```
file = open("names.txt","w")
for x in range (4):
    name = input("Enter name")
    file.write(name+"\n")
file.close()
```



```
try:
    code to try
except ExceptionName:
    What to do if there's
    an exception
```

13. Exception Handling

Exception handling allows you to gracefully manage errors that occur during program execution, preventing your program from crashing.

try-except block:

- The try block contains code that might raise an exception.
- The except block catches and handles specific exceptions.

<!-- end list -->

Python

```
# Handling a ZeroDivisionError
try:
    result = 10 / 0
    print(result)
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")

# Handling a ValueError
try:
    num = int("abc")
    print(num)
except ValueError:
    print("Error: Invalid input for integer conversion!")

# Handling multiple exceptions
try:
    value = int(input("Enter a number: "))
    divisor = int(input("Enter a divisor: "))
    result = value / divisor
    print(f"Result: {result}")
except ValueError:
    print("Error: Please enter valid numbers.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except Exception as e: # Catch-all for other unexpected errors
```

```
    print(f"An unexpected error occurred: {e}")
```

else block (with try-except): The else block executes if no exception occurs in the try block.

Python

```
try:
    number = int(input("Enter an even number: "))
    if number % 2 != 0:
        raise ValueError("Number is not even.") # Manually raise an exception
except ValueError as e:
    print(f"Input error: {e}")
else:
    print(f"You entered an even number: {number}")
```

finally block: The finally block always executes, regardless of whether an exception occurred or not. It's often used for cleanup operations (e.g., closing files).

Python

```
file = None # Initialize to None
try:
    file = open("non_existent_file.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found! (handled in except)")
finally:
    if file: # Check if file was opened before attempting to close
        file.close()
        print("File closed (from finally block).")
```

Beneficial Visuals:

- **Exception Handling Flowchart:** A flowchart showing the flow of execution through try, except, else, and finally blocks.
- **Error Hierarchy Diagram:** A simple hierarchy of common Python exceptions.