Python

Fundamentals

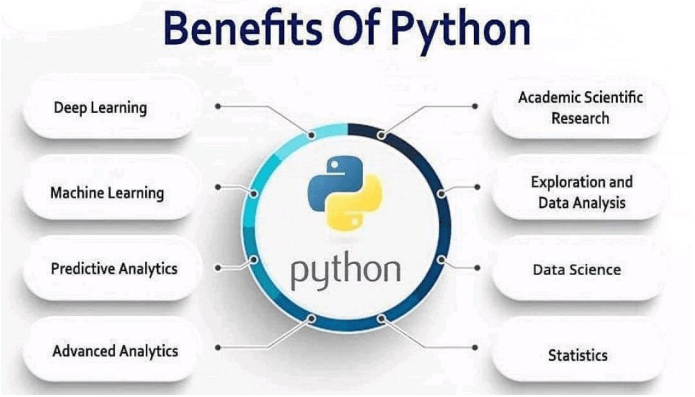
Content

What is Python?

Python is a high-level, general-purpose programming language that is widely known for its readability and ease of use. It features a simple syntax that is similar to everyday spoken language, making it especially beginner-friendly. Python is considered one of the most popular and versatile programming languages, and it is commonly used in automation, data science, machine learning, web development, and more.

Key Benefits and Advantages of using Python

1. ***Easy to learn and use***
   - Python has a simple and readable syntax, which makes it an excellent choice for beginners.
2. ***Versatile and General-Purpose***
   - Can be used for web development, automation, data analysis, machine learning, artificial intelligence, scripting, game development, and more.
3. ***Large Standard Library***
   - Comes with a rich set of built-in modules and functions for handling file operations, regular expressions, databases, web services, and more. Saves time and effort during development.
4. ***Massive Community Support***
   - Abundant resources, tutorials, forums, and libraries are available for learners and professionals alike.
5. ***Cross-Platform Compatibility***
   - Python runs on Windows, macOS, Linux, and many other platforms without requiring code changes.
6. ***Extensive Libraries and Frameworks***
   - Popular libraries like NumPy, Pandas, TensorFlow, PyTorch, Flask, Django, etc., support development in data science, AI, web apps, and more.
7. ***Strong Support for Automation and Scripting***
   - Excellent for automating repetitive tasks such as file manipulation, web scraping, or system monitoring.
8. ***Integration Capabilities***
   - Easily integrates with other languages like C/C++, Java, and with web services, databases, and APIs.
9. ***Great for Prototyping and Rapid Development***
   - Quick to write and test code, making it ideal for startups, research, and experimental projects.



Where is Python used?

- Python is used in a **wide range of industries and applications** due to its simplicity, versatility, and powerful libraries.

A quick overview of where **python** is commonly used:

- Web development
- Data Science & Data Analysis
- Machine Learning & Artificial Intelligence
- Scientific and Numeric Computing
- Automation & Scripting
- Game Development
- Software Development & Prototyping
- Cybersecurity and Ethical Hacking
- E-commerce and FinTech
- Entertainment and Media

Types of Program Flow in Python

In programming, a program flow (or control flow) refers to the order in which statements, instructions, or function calls are executed in a program. The flow determines how a program progresses from one instruction to another based on conditions, loops, and function calls. Python primarily uses a **sequential control flow** by default, but supports several types of control flow to manage the execution of code.

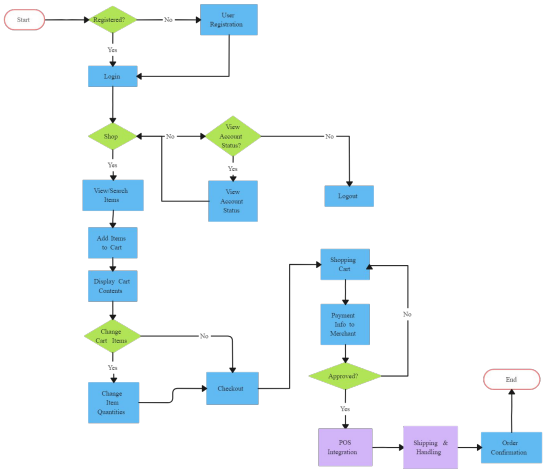
1. **Sequential Flow** - Statements are executed one after another in the order they appear.
2. **Conditional Flow** (Branching) - Controls execution based on conditions using statements like **if**, **elif**, **else** and **match case**.
3. **Iterative Flow** (Loops) - Repeats a block of code multiple times using loops like **for** and **while**. Can be altered by continue, else and break loops.
4. **Function Calls** (Subroutines) - Transfer control to a function and returns after execution.
5. **Exception Handling** (Jump Flow) - Alters flow when errors occur using **try**, **except** and **finally**.

```python
# Sequential Flow Example
print(1)
print(2)

# Conditional Flow Example
animal = "dog"
if animal == "dog":   # This if statement checks if the animal is a dog.
    print("This is a dog.")
elif animal == "cat":     # This elif statement gets executed if the animal is a cat.
    print("This is a cat.")
else:       # This else statement gets executed if the IF condition and the ELIF condition are not met.
    print("This is neither a dog nor a cat.")

# Iterative Flow Example
for number in range(5):  # This for loop iterates over a range of numbers from 0 to 4.
    print(f"Iteration {number + 1}")  # This line prints the current iteration number.

number = 0
while number < 5:  # This while loop continues as long as the number is less than 5.
    number += 1  # This line increments the number by 1 in each iteration.
    if number == 2:
        print("Reached number 2")  # This way we altered the normal flow by adding a condition.
        continue # This line continues to the next iteration when the number reaches 2 WITHOUT executing the rest of the loop body.
    if number == 4:
        print("Reached number 4, breaking loop")
        break # This line breaks the loop when the number reaches 4. This way we altered the normal flow by breaking the loop.
    print(f"While loop iteration {number}")  # This line prints the current iteration number.


# Function Calls Example
def greet(name):
    print(f"Hello, {name}!")  # This function prints a greeting message.

# In order to activate the function, we need to call it with a name. Otherwise it won't execute.
greet("Alice")  # This line calls the greet function with "Alice" as an argument.

# Exception Handling Example
try:
    result = 10 / 0  # This line attempts to divide by zero, which will raise an exception.
except ZeroDivisionError as e: # This block catches the ZeroDivisionError error.
    print(f"An error occurred: {e}")
finally:
    print("This block always executes, regardless of whether an exception occurred or not.")
```

Why is Programming Flow important?

Programming flow is critically important because it defines **how a program executes its logic**, and whether it behaves **correctly, efficiently, and predictably**.

1. **Controls the Logic of the Program** - *Without proper control flow, your program might produce incorrect results or behave unpredictably.*
   - You wouldn't want your login system to run "access granted" code **before** checking the password!
2. **Enables Decision-Making** - *Real-world applications must react to user input, environment, or data conditions.*
   - In an ATM program, flow logic determines whether to dispense cash, show an error or print a receipt based on account balance and user input.
3. **Supports Repetition and Automation** - *Loops let you repeat tasks without writing the same code multiple times*
   - Processing every item in a shopping cart or checking each row in a database
4. **Improves Readability and Maintainability** - *Well-structured flow makes the program easier to read, debug and update. Others (and your future self) can understand your intent and logic.*
   - Breaking logic into functions or using clear if-else blocks helps teams collaborate better.
5. **Prevents Errors and Bugs** - *Proper flow control avoids infinite loops, skipped conditions and unwanted behavior.*
   - Catching a division by zero before it crashes the program.
6. **Drives Program Efficiency** - *Control structures optimize when and how often code runs, avoiding unnecessary computations. This leads to faster and more resource-efficient software.*
   - Exiting a loop early using break when a result is already found.



Python Data Types

1. **Numeric Types**
   - **int** | Whole numbers
   - **float** | Decimal (floating-point) numbers
   - **complex** | Complex numbers (real + imaginary)
2. **Text Type**
   - **str** | Sequence of characters
3. **Boolean Type**
   - **bool** | Truth values (0 or 1) <- True or False
4. **None Type**
   - **NoneType** | Represents “no value”
5. **Collection (Container) Types**
   - **list** | An **ordered**, **changeable** (mutable) collection that **allows duplicates**. You can add, remove, or modify elements. Ideal for storing a sequence of related items.
   - **tuple** | An **ordered**, **unchangeable** (immutable) collection. Used for data that shouldn’t be modified after creation. Can also contain duplicates.
   - **set** | An **unordered**, **changeable** collection that **does not allow duplicates**. Automatically removes repeated items. Useful for mathematical operations like union, intersection, etc.
   - **dict** | A **key-value pair** collection. Keys are unique, values can change. Great for looking up values by names, IDs, etc. Similar to real-world dictionaries (word → definition).

- We can also check the **type** of a *variable* by calling the type() built-in method.
- We can **convert** data types.

```python
# Numeric Types
int = 1
float = 1.0
complex = 1 + 1j

# Text Types
str = "Hello, World!"

# Boolean Type
bool = True

# None Type
NoneType = None

# Collection Types
list = [3, 1, 3]
set = {1, 2, 3}
tuple = (1, 2, 3)
dict = {"key": "value"}


# Checking the type of a variable
print(type(int))

# We can also convert between types
str(5) # "5" Convert int to str
int("5") # 5 Convert str to int
list("abc") # ['a', 'b', 'c'] Convert str to list
```

What are Variables?

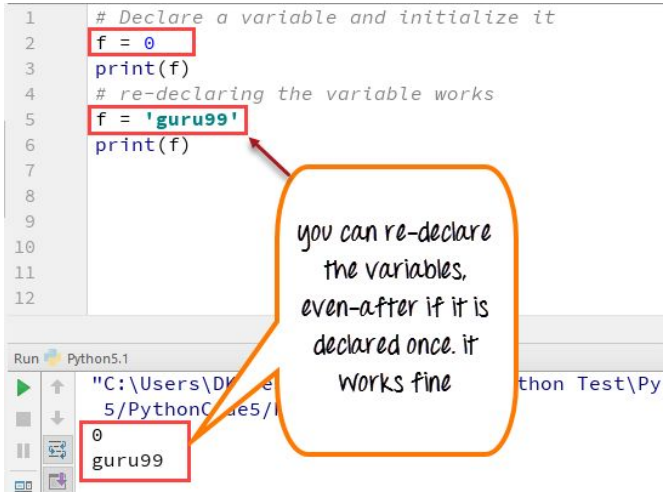
*In Python, a **variable** is a name that stores a value. Think of it as a label you attach to data, so you can refer to it later.*

1. **Key Features of Python Variables**
   a. *No need to declare type* - Python is dynamically typed | the type is inferred.
   b. *Case-Sensitive* - Name, name and NAME are different variables.
   c. *Can be reassigned* - You can assign a new value or even a different type.
   d. Variables exist when you first assign them a value.

2. **Variable Naming Rules**

   a. Can contain letters(a-z, A-Z), digits and underscores (_).
   b. **MUST** start with a letter or underscore(_).
   c. Cannot contain spaces - Use underscores _ instead of spaces.
   d. Case-Sensitive - Python treats uppercase and lowercase differently.
   e. **Cannot** be a Python *keyword* - You cannot use reserved words (like if, class, while, def etc.).

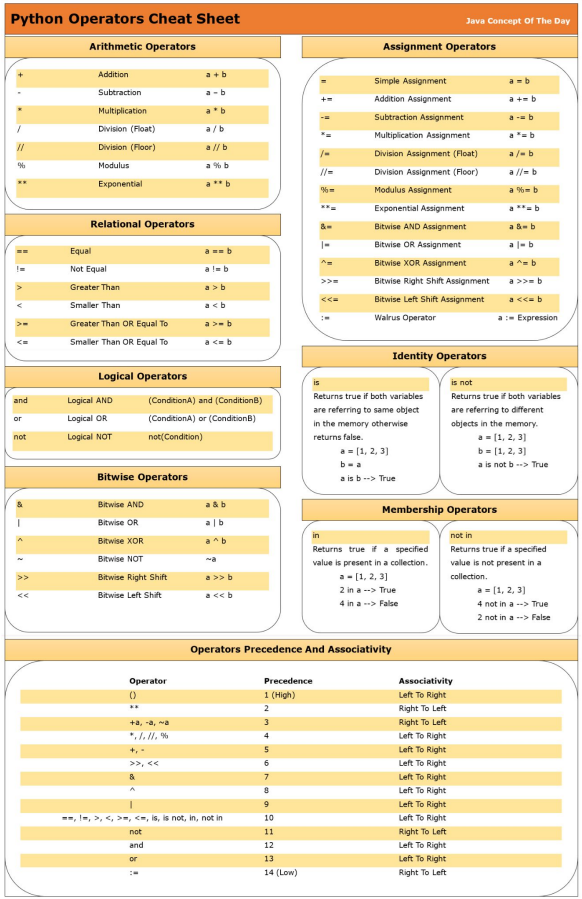
- You can assign **multiple** variables at **once**: x, y, z = 1, 2, 3.



Operators

Operators are special symbols or keywords used to perform operations on variables and values.

1. **Arithmetic** Operators - Used to perform basic math.
2. **Assignment** Operators - Used to assign values to variables.
3. **Comparison** Operators - Used to compare values.
4. **Logical** Operators - Used for combining conditions.
5. **Identity** Operators - Used to check if two variables point to the same object.
6. **Membership** Operators - Used to check if a value is in a sequence (like a list or string).
7. **Bitwise** Operators (Advanced - works at binary level)



Python Operators Cheat Sheet — Java Concept Of The Day

Arithmetic Operators

+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division (Float)	a / b
//	Division (Floor)	a // b
%	Modulus	a % b
**	Exponential	a ** b

Relational Operators

==	Equal	a == b
!=	Not Equal	a != b
>	Greater Than	a > b
<	Smaller Than	a < b
>=	Greater Than OR Equal To	a >= b
<=	Smaller Than OR Equal To	a <= b

Logical Operators

and	Logical AND	(ConditionA) and (ConditionB)
or	Logical OR	(ConditionA) or (ConditionB)
not	Logical NOT	not(Condition)

Bitwise Operators

&	Bitwise AND	a & b
|	Bitwise OR	a | b
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
>>	Bitwise Right Shift	a >> b
<<	Bitwise Left Shift	a << b

Assignment Operators

=	Simple Assignment	a = b
+=	Addition Assignment	a += b
-=	Subtraction Assignment	a -= b
*=	Multiplication Assignment	a *= b
/=	Division Assignment (Float)	a /= b
//=	Division Assignment (Floor)	a //= b
%=	Modulus Assignment	a %= b
**=	Exponential Assignment	a **= b
&=	Bitwise AND Assignment	a &= b
|=	Bitwise OR Assignment	a |= b
^=	Bitwise XOR Assignment	a ^= b
>>=	Bitwise Right Shift Assignment	a >>= b
<<=	Bitwise Left Shift Assignment	a <<= b
:=	Walrus Operator	a := Expression

Identity Operators

is
Returns true if both variables are referring to same object in the memory otherwise returns false.
a = [1, 2, 3]
b = a
a is b --> True

is not
Returns true if both variables are referring to different objects in the memory.
a = [1, 2, 3]
b = [1, 2, 3]
a is not b --> True

Membership Operators

in
Returns true if a specified value is present in a collection.
a = [1, 2, 3]
2 in a --> True
4 in a --> False

not in
Returns true if a specified value is not present in a collection.
a = [1, 2, 3]
4 not in a --> True
2 not in a --> False

Operators Precedence And Associativity

Operator	Precedence	Associativity
()	1 (High)	Left To Right
**	2	Right To Left
+a, -a, ~a	3	Right To Left
*, /, //, %	4	Left To Right
+, -	5	Left To Right
>>, <<	6	Left To Right
&	7	Left To Right
^	8	Left To Right
|	9	Left To Right
==, !=, >, <, >=, <=, is, is not, in, not in	10	Left To Right
not	11	Right To Left
and	12	Left To Right
or	13	Left To Right
:=	14 (Low)	Right To Left

Functions

A function is a block of reusable code that performs a specific task. Instead of repeating the same code multiple times, you define it once in a function and call it whenever needed.

1. **Why Use Functions?**
   - **Organize** your code.
   - **Avoid repetition** (DRY = Don't repeat yourself).
   - Make code easier to **understand**, **test** and **maintain**.
   - Allow **reuse** of code
2. **Key Functions Concepts**
   - *def* - Keyword to define a function.
   - Function Name - Must follow variable *naming rules*.
   - Parameters - Variables passed into the function (optional).
   - *return* - Sends a value back to the caller (optional).
3. **Function Types**
   a. Built-in Functions - print(), len(), type(), sum(), input()
   b. User-defined Functions - Created by using *def*.
   c. Lambda (Anonymous) Functions - Short, inline functions.
4. **Best practices**
   - Use descriptive names.
   - Keep functions **short** and **focused**.
   - Write **docstrings** for clarity (optional but very important).
5. **Return vs Print**
   a. print - Displays output to the screen.
   b. return - Sends value back to the caller.

```python
# How to define a function in Python
def function_name(parameter: str) -> str:
    # block of code
    return parameter  # optional
function_name("example") # calling the function.
# To be noted that we have to encapsulate the function call in a variable or print() to see the result

# Lambda function
lambda_function = lambda x: x * 2  # This is a simple lambda function that doubles the input
print(lambda_function(5))  # Output: 10

# Function with default parameters
def greet(name: str = "World") -> str:
    return f"Hello, {name}!"

# Function with Multiple Parameters
def add_numbers(a: int, b: int) -> int:
    return a + b

# Adding descriptive docstrings to the functions
def function_with_docstring(param1: int, param2: int) -> int:
    """
    This function adds two integers and returns the result.

    Args:
        param1 (int): The first integer to add.
        param2 (int): The second integer to add.

    Returns:
        int: The sum of the two integers.
    """
    return param1 + param2

# Testing the functions
print(greet())  # Output: Hello, World!
```

Classes - OOP (Object-Oriented Programming)

A class is a blueprint for creating objects.

***Objects** are instances of **classes** and can have:*

- **Attributes** (data)
- **Methods** (functions)

1. **Key Concepts**
   a. class - A keyword to define a class
   b. Object - An instance of a class
   c. Attribute - A variable tied to the object (like name, age)
   d. Method - A function inside a class that operates on object data (self is used)
   e. Inheritance - Let's a class (child) reuse code from another class (parent).
      - Base class or Parent class: *provides* behavior (e.g., Animal).
      - Derived Class or Child class: *inherits* and *can overrid*e behavior (e.g., Dog)
   f. Encapsulation - **Hiding internal details** and exposing only what's necessary. Helps protect object state and keep code clean.
      - _single_underscore: internal use (weakly private)
      - __double_underscore: name mangling (stronger privacy)
   g. Polymorphism (Many Forms) - Same method name, different behaviors. Lets different classes respond to the same method call in their own way.
   h. Abstraction (Essential Features Only) - Showing only the necessary features, hiding internal details. Helps reduce complexity and increases clarity.
2. **What is self?**
   - Refers to the current object instance.
   - You must include self as the first parameter of every method in a class, but *you don't pass it manually* when calling methods – Python does it for you.
3. **Special Methods (Magic or Dunder Methods)**
   a. __init__ – Constructor (runs on creation)
   b. __str__ – String representation
   c. __len__ – Used by len()
   d. __repr__ – Official representation
4. **Best Practices**
   - Use **CamelCase** for class names (MyClass)
   - Use **snake_case** for methods and variable names.
   - Keep class **single-purpose**
   - Use **docstrings** to describe the class and methods.

Python Collections - Key Concepts

1. **Indexing**
   - You can access elements using an index: collection[index].
   - It doesn't work with set or dict (unordered or key-based).
2. **Slicing**
   - Get a portion of a sequence using: collection[start:stop:step].
   - It doesn't work with set or dict (unordered or key-based).
3. **Iteration**
   - Looping through collections.
4. **Membership Test (in)**
   - Check if an item exists in the collection.
5. **Length**
   - Use len() to get the *size* of the collection.
6. **Nested Collections**
   - Collections can contain other collections.
7. **Built-in Functions with Collections**
   a. len() – Number of items.
   b. sorted() – Return a sorted copy.
   c. sum() – Add all numbers in a collection.
   d. min() , max() – Return smallest/largest item.
   e. list(), set() – Convert types.

	Tuple	List	Dictionary	Set
Eample	('Book 1', 12.99)	['apple', 'banana', 'orange']	{'name': 'Joe', 'age': 10}	{10, 20, 12}
Mutable?	Immutable	Mutable	Mutable	Mutable
Ordered?	Ordered	Ordered	Preserves order since Python 3.7	Unordered
Iterable?	Yes (takes linear time)	Yes (takes linear time)	Yes (constant time)	Yes (constant time)
Use case	Immutable data	Data that needs to change	Key/Value pairs	Unique items

```
''' Indexing example '''
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']

# Accessing elements by index
print(fruits[1])  # Output: banana
# Negative indexing example
print(fruits[-1])  # Output: elderberry

''' Slicing example'''
print(fruits[1:4])  # Output: ['banana', 'cherry', 'date']

''' Slicing with step example'''
print(fruits[::2])  # Output: ['apple', 'cherry', 'elderberry']

''' Slicing with negative step example'''
print(fruits[::-1])  # Output: ['elderberry', 'date', 'cherry', 'banana', 'apple']

''' Slicing with start, stop, and step example'''
print(fruits[1:5:2])  # Output: ['banana', 'date']

'''Iteration example'''
for fruit in fruits:
    print(fruit) # Output: apple, banana, cherry, date, elderberry (each on a new line)

''' Membership test example'''
print('banana' in fruits)  # Output: True
print('fig' in fruits)     # Output: False

''' Lenghth example'''
print(len(fruits))  # Output: 5

'''Nested collection example'''
nested_fruits = [['apple', 'banana'], ['cherry', 'date'], ['elderberry']]
# Accessing nested elements
print(nested_fruits[0][1])  # Output: banana
print(nested_fruits[0]) # Output: ['apple', 'banana']


''' List comprehension example '''
squared_numbers = [x**2 for x in range(10)]
print(squared_numbers)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

String Manipulation

String manipulation *is the process of* ***modifying***, ***analyzing*** *or* ***formatting*** *text in a program. Since strings are just sequences of characters (like "Hello" or 123), Python gives us a lot of tools to work with them easily and powerfully.*

1. **String manipulation is everywhere:**
   - Formatting user input.
   - Cleaning up data (e.g., removing spaces)
   - Searching and replacing words
   - Parsing text (like CSV files or logs)
   - Generating dynamic messages or code
2. **Common string manipulation operations**
   a. *Changing Case* - To control how text appears.
   b. *Stripping Whitespace* – To clean up input.
   c. *Finding and Replacing* – To locate and modify parts of a string.
   d. *Splitting and Joining* – To break apart and combine strings.
   e. *Checking Content* – To validate if a string is a number, alphabetic, etc.
3. **Key Concept: Strings are *IMMUTABLE***
   - Once a string is created, it can't be changed – any operation like replace() or strip() returns a **new string**, not a modified version of the original.

```
# Changing case
"hello".upper()  # 'HELLO'
"HELLO".lower()  # 'hello'
"Hello World".title()  # 'Hello World'

# Stripping whitespace
"   Hello World   ".strip()  # 'Hello World'

# Finding and Replacing
"text".find("x") # Returns the index of 'x' in 'text', or -1 if not found
"dog dog".replace("dog", "cat")  # 'cat cat'

# Splitting and Joining
"one,two,three".split(",")  # ['one', 'two', 'three']
"-".join(["one", "two", "three"])  # 'one-two-three'

# Checking Content
"hello".isalpha()  # True, all characters are alphabetic
"123".isdigit()  # True, all characters are digits
"hello123".isalnum()  # True, all characters are alphanumeric
```

Python Collections Manipulation

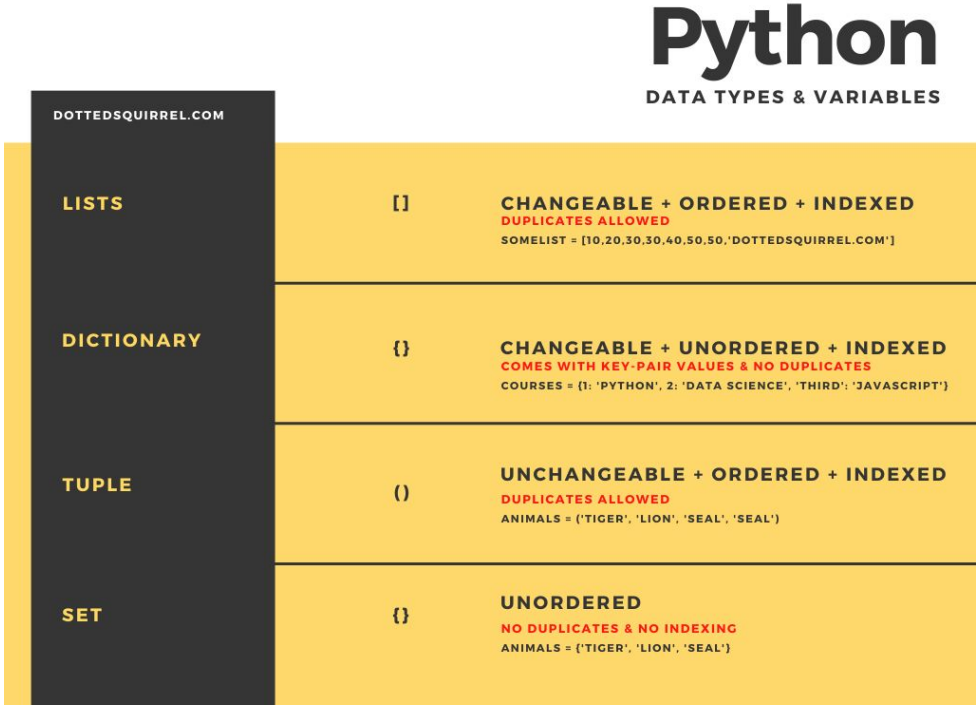
Python collections are data structures used to store groups of items. They come in different flavors depending on your needs.

1. **Common Collection Manipulations**
   a. Adding Elements
      - list – .append(x), .insert(i, x), .extend([....])
      - set – .add(x)
      - dict – d[key] = value
      - tuple – **Immutable** (rebuild if needed)

   b. Removing Elements
      - list – .remove(x), .pop(i), del
      - set – .remove(x), .discard(x)
      - dict – del d[key], .pop(key)

   c. Updating Items
      - list – By index: mylist[i] = x
      - dict – By Key: mydict[key] = value
      - set – Remove then add
      - tuple – Must recreate

   d. Searching & Testing
      - Check if item exists – “apple” in mylist
      - Get index (list/tuple) – mylist.index(“apple)
      - Count Items – mylist.count(“apple)
      - Get keys/values (dict) mydict.keys(), mydict.values()
2. **Copying (Lists, Tuples, Strings)**
   - Make a shallow copy – copy = a[:]
3. **Practical Uses**
   - Lookup by key (dict)
   - Maintain insertion order (list, dict)
   - Ensure uniqueness (set)
   - Store unchanging values (tuple)
4. **Common Pitfalls**
   - *Modifying* while iterating -> can cause bugs
   - Mutable defaults in functions -> use None and assign later.
   - Confusing deep vs shallow copies -> use copy.deepcopy() if needed.



Errors Handling

*Error handling is how you **detect**, **catch** and **respond** to exceptions (errors) that occur during the execution of your code. For example dividing by zero, accessing a missing file or using an undefined variable.*

1. **Types of Errors in Python**
   a. Syntax Errors – Mistakes in the code structure – Python can't even run it.
   b. Runtime Errors (Exceptions) – Errors that happen while the code is running.
2. **The Try-Except Block**
   - Use try to attempt code and except to catch and handle errors.
3. **Optional Blocks**
   - try – Code that might raise an exception.
   - except – What to do if an exception occurs.
   - else – Code to run if no exception occurs.
   - finally – Code that always runs, error or not.
4. **Catching Specific vs Generic Exceptions**
   - **Specific** (best practice)
     - except FileNotFoundError, except ValueError, etc.
   - **Generic** (less safe, but useful sometimes)
     - except Exception as e: print("Error:", e)
5. **Raising Your Own Exceptions**
   - You can create your own errors using raise
     - age = 1 if age < 0 raise ValueError("Age can't be negative!")
6. **Why is Error Handling Important?**
   - *Prevents crashes and bad user experiences.*
   - *Makes debugging easier.*
   - *Enables graceful failure and fallback options.*
   - *Essential for **production-quality** software.*


```python
# TO demonstrate exception handling of multiple except blocks

try:
    num1 = int(input("enter value of number1: "))
    num2 = int(input("enter value number2: "))
    result = num1/num2
    print(result)
except ValueError:
    print("Not valid number")
except ZeroDivisionError:
    print("Number Cannot be Divided by Zero")
except:
    print("This is the Generic Error")
```

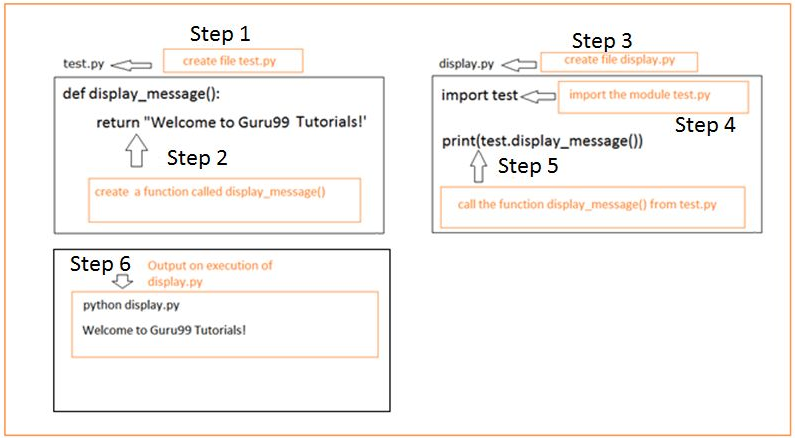
```
enter value of number1: 34g
Not valid number
```

Importing & Libraries

A library in Python is a collection of pre-written code (functions, classes and tools) that you can use so you don't have to write everything from scratch.

1. **What Does import Mean**
   - The import statement brings in external code (modules/libraries) so you can use it in you program.
2. **Ways to Import Libraries**
   a. Basic import – import math
   b. Import with alias – import numpy as np
   c. Import specific function – from math import sqrt
   d. Import multiple items – from math import sin, cos, pi
3. **Standard vs External Libraries**
   a. Standard Library
      - Comes with Python.
      - No installation needed.
      - Examples: math, os, sys, datetime.

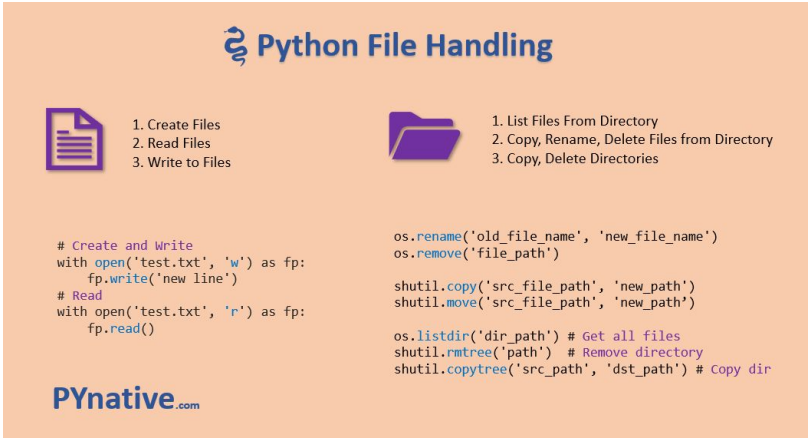
   b. External Libraries
      - Installed separately (via pip)
      - Examples: requests, pandas, flask.
4. **Importing Your Own Code (Modules)**
   - You can also import your own modules. For example if you are working on a project and you already have a function defined in another Python file, you can use import myexample (for the whole file) or from myexample import function (for a specific function).
     - Note that this direct import only works if both files are in the same directory, if they are not you will have to specify the path relative to the project's root, such as: from mynewfolder.myexample import function
5. **Why Imports Matter**
   - DRY Principle: Don't Repeat Yourself – reuse code.
   - Clean code: Organize logic into files/modules.
   - Powerful tools: Leverage thousands of libraries from the Python ecosystem.



Working with Files

Python provides built-in support for file operations like reading, writing and appending using the open() function.

1. **Basic Syntax**
   - 1. file = open("filename.txt", "mode") 2. # Do something 3. file.close()
2. **Common File Modes**
   a. 'r' – Read (default)
   b. 'w' – Write (overwrites file)
   c. 'a' – Append (adds to end)
   d. 'x' – Create (error if exists)
   e. 'b' – Binary mode (e.g., 'rb')
   f. 't' – Text mode (default)
3. **Reading Files**
   a. *Read entire file:* with open("file.txt", "r") as f: content = f.read() print(content)
   b. *Read line by line*: with open("file.txt", "r") as f: for line in f: print(line.strip())
4. **Writing to Files**
   a. *Overwrite existing file*: with open("file.txt", "w") as f: f.write("Hello, World")
   b. *Append to a file*: with open("file.txt", "a") as f: f.write("\nNew line added.")
5. **Why with open()**
   - Using with is recommended because it automatically closes the file, even if errors occur
6. **File Handling Extras**
   a. *Check if file exists*: import os if os.path.exists("file.txt"): print("File exists.")
   b. *Remove a file*: os.remove("file.txt")
7. **Real-world Use Cases**
   - *Logging system events.*
   - *Loading configuration files.*
   - *Reading CSV or text datasets.*
   - *Exporting results from a script.*
   - *Writing reports or summaries to disl.*



Glossary

Python
A high-level, versatile programming language known for its readability and wide range of applications.
Variable
A named container used to store data in a Python program.
Variable Naming Rules
Rules for naming variables: must begin with a letter or underscore, can't use Python keywords, and should be descriptive.
Data Types
Categories of data: int, float, str, bool, list, tuple, dict, set, etc.
Collections
Data structures that hold multiple values: list, tuple, dict, and set.
Indexing
Accessing an item in a collection using its position.
Slicing
Extracting a portion of a collection using start:stop:step syntax.
Iteration
Looping through a collection using for or while.
Membership
Checking if a value exists in a collection using in or not in.
Nested Collections
Collections inside other collections (e.g., list of lists).
Length (len)
A function that returns the number of items in a collection.
String Manipulation
Methods to modify or analyze strings: .lower(), .upper(), .replace(), .split(), etc.
Operators
Symbols to perform operations: arithmetic (+, -), comparison (==, !=), logical (and, or).
Control Flow
The order in which code executes, controlled with if, elif, else, for, while, etc.
Function
A reusable block of code defined with def that performs a task.
Arguments/Parameters
Inputs to a function. Arguments are passed, parameters are defined.
Return Statement
Sends a result back from a function.

Class
A blueprint for creating objects in Object-Oriented Programming.
Object
An instance of a class.
Attribute
A variable attached to an object or class.
Method
A function defined inside a class.
Inheritance
A way to create a new class that inherits attributes and methods from another class.
Encapsulation
Hiding internal object details; restricting access using _ or __.
Polymorphism
The ability to use the same method name with different implementations.
Abstraction
Hiding complex implementation details, exposing only the essentials.
Exception
An error detected during program execution.
Error Handling
Using try, except, else, finally to handle exceptions safely.
Raise
A statement to manually throw an exception.
Import
A keyword used to include external modules or files.
Library
A collection of pre-written code (functions, classes, etc.) that you can import and use.
Module
A single Python file containing code that can be imported.
Package
A directory of modules, often with an __init__.py file.
File Handling
Reading from or writing to files using open(), read(), write(), etc.
with Statement
Used to handle files safely, automatically closing them after use.
Standard Library
Built-in Python modules like math, datetime, os, etc.
External Library
Modules you install separately with pip, like pandas or requests.

Dive deeper into any subject!

Official Python Resources

https://docs.python.org/3/

https://wiki.python.org/moin/BeginnersGuide

Interactive and Tutorial-Based Learning

https://www.learnpython.org/ - Offers a free, interactive Python tutorial suitable for beginners and intermediate learners, allowing hands-on coding directly in the browser.

Comprehensive Learning Platform

https://www.geeksforgeeks.org/python-programming-language-tutorial/ – Provides extensive articles and examples on Python programming, including data structures, algorithms and more.

Academic and Structured Courses

https://python101.pythonlibrary.org – A free book that covers Python basics, including data types, control flow, functions, and classes, suitable for beginners.