



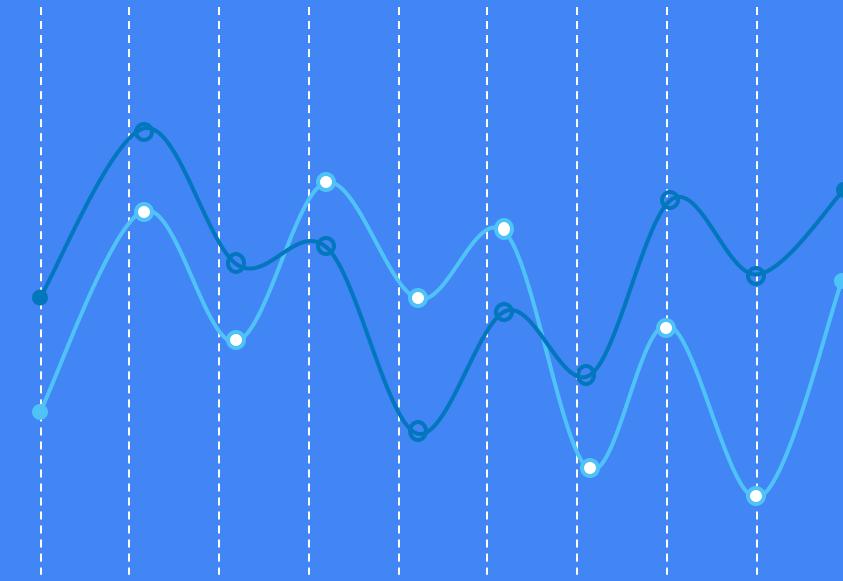
Certification Training Courses for Graduates and Professionals

[www.signivetech.com](http://www.signivetech.com)

# Python

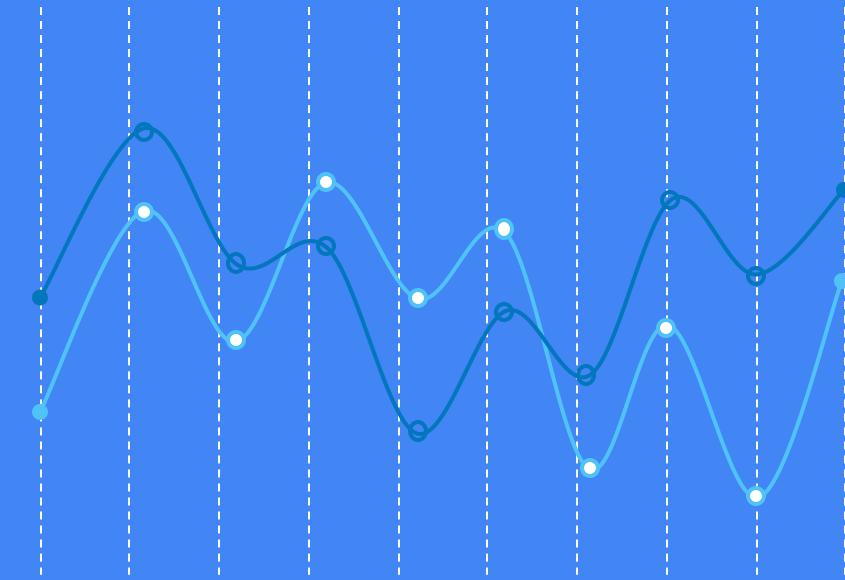
# Agenda

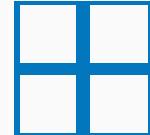
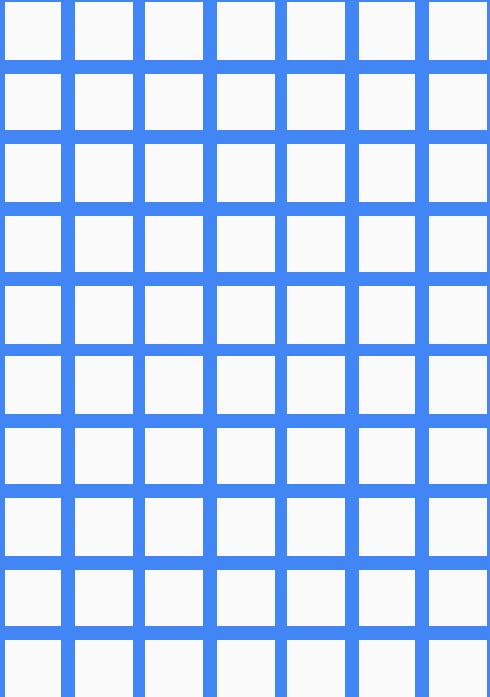
- ❖ Introduction
- ❖ Python Basics
- ❖ Numbers
- ❖ Operators
- ❖ Strings
- ❖ Decision Making
- ❖ Loops
- ❖ List
- ❖ Tuple
- ❖ Dictionary



# Agenda

- ❖ Set
- ❖ Functions
- ❖ Modules
- ❖ Files & I/O
- ❖ Date & Time
- ❖ Exceptions
- ❖ Object Oriented
- ❖ Regular Expressions





# Introduction

Python is a:

- High-Level
- Interpreted
- Interactive
- Object-Oriented scripting language.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

### Python:

- **Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **High-Level:** Python is a High-Level Language meaning code writes in English language & can converted into low-level or assembly language.

# History of Python

- Python is developed by **Guido van Rossum**
- Python named after a TV show called as “Monty Python’s Flying Circus”
- van Rossum is principal author and BDFL(Benevolent Dictator for Life)
- Python 1.0 released on January 1994
- Python 2.0 released on October 2000
- Python 3.0 released on December 2008
- Both Python 2.0 and 3.0 versions are available today and in the development modes.

# Features of Python

- Easy-to-learn, Easy-to-read and Easy-to-maintain
- Interactive
- Portable
- Extendable
- Supports databases
- Scalable
- Supports automatic garbage collection
- GUI Programming

# Features of Python

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **Interactive:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Supports databases:** Python provides interfaces to all major commercial databases.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

# Features of Python

- Python supports functional and structured programming methods as well as Object Oriented Programmings.
- Python provides very high-level dynamic data types and supports dynamic type checking.
- Python supports automatic garbage collection.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems.
- **Free and open source:** One can freely distribute copies of this software, modify it etc.
- **Programming Paradigm:** Python Supports both Procedure Oriented (Dividing code into small modules or functions & re-use them) and Object Oriented (user-defined classes & objects) Paradigms.

# Applications of Python

- **Web Application:** Python can be used to develop scalable and secure web applications. Frameworks like **Django**, **Flask**, **Pyramid** etc are amazing to design and develop web based applications.
- **Computer Software or Desktop Applications:** As python can be used to develop GUI too, hence it is a great choice for developing desktop applications. **Tk** is an open source widget toolkit which can be used to develop desktop applications with python. **Kivy** is another such platform.
- **Scientific Computing Applications:** For its amazing computational power and simple syntax, python is used for scientific computing applications. Python libraries like **SciPy** and **NumPy** are best suited for scientific computations.

# Applications of Python

- **AI (Artificial Intelligence) and ML (Machine Learning):** Python is at the forefront of the paradigm shift towards Artificial Intelligence and Machine Learning.
- **Image Processing:** Python is known for its image processing capabilities, which includes traversing and analysing any image pixel by pixel. There are numerous python libraries available for image processing, for example: Pillow, scikit-image etc.
- **Websites like Instagram, Reddit, Mozilla,** have been developed using Python. There are various web frameworks like **Django** (most popular one) and **Pyramid**, based on Python, which can be used to develop modern web applications.

# Modes of Python

- Modes of Python:
- Batch Mode:
  - A complete program is written in a file, saving the file with *.py* extension and run the complete file as program giving required output.
- Interpreter Mode:
  - A program can be executed line by line displaying output.

## Installation

- Download Python from official website.
- Install Python on the machine by following simple steps.
- Run IDLE which is a python shell to execute the python code. This comes with Python and it is a default environment.
- We will use **Jupyter Notebook** which comes with **Anaconda Community**.  
The Jupyter Notebook is a powerful tool to progress with Python programming along with Data Science and Machine Learning modules.

## Python 2 vs Python 3

- Python 2.X: To display output, use `print`
  - In Python 2.X use of parentheses in Print function is optional.
- Python 3.X: To display output, use `print()`
  - In Python 3.X use of parentheses in Print function is mandatory.

Python 2.X:

```
print "Hello Python 2" → Output: Hello Python 2
```

```
print("Hello Python 2") → Output: Hello Python 2
```

Python 3.X:

```
print("Hello Python 3") → Output: Hello Python 3
```

## Python 2 vs Python 3

- Division Operator:

Python 2.X:

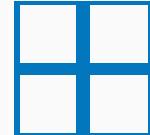
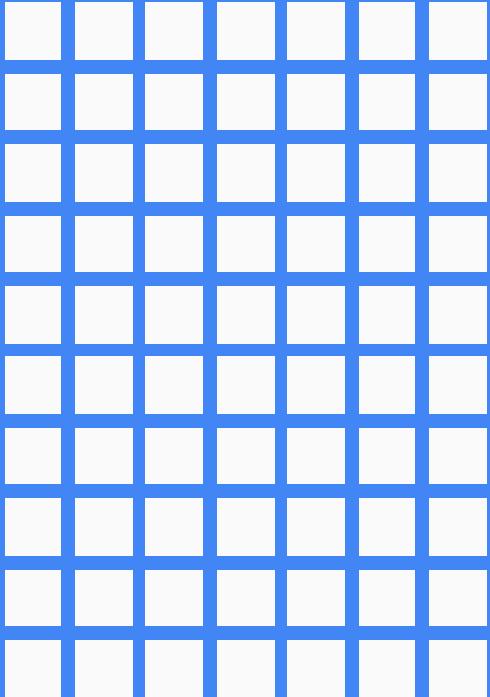
`print 7 / 5` → Output: 1

`print 7.0 / 5` → Output: 1.4

Python 3.X:

`print(7 / 5)` → Output: 1.4

`print(7.0 / 5)` → Output: 1.4



# Python Basics

# Identifiers

- Identifier is a name used to identify a variable, function, class, module or any other objects.
- They should start with a letter A-Z or a-z or an underscore(\_) followed by zero or more letters, underscores and digits(0-9).
- Identifiers start with Special Symbols or Digits are invalid.
- Punctuation characters such as @, \$, % are not allowed.
- Python is a case sensitive.
- Class names should start with uppercase letter and followed by lowercase letters. All other identifiers should start with lowercase letter.
- \_identifiername: It makes Protected Identifier.
- \_\_identifiername: It makes Private Identifier.
- \_\_identifiername\_\_: Special methods/Magic methods.

# Keywords

- Reserved words that cannot be used as constants or variables or identifiers are called as keywords.
- Keywords contain lowercase letters only.

and	assert	break	class	continue	def
del	elif	else	except	finally	for
from	global	if	import	in	is
lambda	not	or	pass	raise	return
	try	while	with	yield	

## Comments

- A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.
- Example:

```
# this is comment 1
```

## Lines and Indentations

- Statement typically ends with 'New Line' (Enter Key).
- **There is no command terminator**, which means no semicolon ; or anything.
- Indentated blocks start with ':'
- Example:

```
x = 32
```

```
y = 34.56; print("Hi")
```

```
if x = 23:
```

```
    print("Hello")
```

## Lines and Indentations

- **Line Continuation:** To write a code in multiline without confusing the python interpreter, is by using a backslash \ at the end of each line to explicitly denote line continuation.
- Expressions enclosed in ( ), [ ] or { } brackets don't need a backward slash for line continuation.
- Example:

```
sum = 23 + \
```

```
34
```

```
vowels = ['a', 'e', 'i',  
          'o', 'u']
```

## Quotations

- Python accepts ('), (""), and ("") quotes to denote strings.
- Example:

X = 'hello'

Y = "Hello Python"

Z = ""hey, this is Python,  
and this is a great language""

## Print() Function

- To display Output or to show any message, the print() function is used. Pass the string message in the print() function.

```
print("Hello Python") → Output: Hello Python
```

- By default print() function ends with a newline.
- However, print() function comes with a parameter called 'end'. Here, one can end a print statement with any character/string using this parameter.

```
print("Hello Python", end = " ... ")
```

```
print("Welcome to the World")
```

Output: Hello Python ... Welcome to the World

## Input() Function

- To get Input or Message from user, the `input()` function is used. By default, the `input()` function takes input in the String format.

```
name = input("Enter Name: ")
```

```
print("Your Name:", name)
```

Output:

Enter Name: *D'costa*

Your Name: D'costa

# Variables

- A variable(symbolic name or label) is a reference to some data stored at a specific memory location.
- Variable is an important element of programming world, visually, can consider variable as a *box*, that is capable of storing some value within it.
- They also provide a way of labelling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in the memory. This data can then be used throughout your program.
- Python is a dynamically typed/loosely typed language.

# Variables

- Example:

X = 2 → Integer

Y = 5.6 → Floating Point

Z = "Python" → String

W = 'A' → String

P = True → Boolean

R = None → None or Null

- Example: (Multiple Assignments are Possible)

X, Y, Z = 2, 6, 5.6

print(X, Y, Z) → Output: 2 6 5.6

A = B = C = 1

print(A, B, C) → Output: 1 1 1

## Variables

- `type()` method can be used to check the datatype of the variable.
- Example:

`type(X) → Output: int`

`type(Y) → Output: float`

`type(Z) → Output: str`

`type(W) → Output: str`

`type(P) → Output: bool`

`type(R) → Output: NoneType`

# Data Types

- **Numbers:** Number data types store numeric values. Number objects are created when you assign a value to them. Eg: `a = 10; b = 2.5`
- **Strings:** Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Eg: `x = "python"`
- **Boolean:** A boolean value of either True or False. Eg: `t = True; f = False`
- **List:** Collection of elements having different or same data types.  
Eg: `y = ["python", "java", 2]; u = [1, 2, [3, 4, 5], 6, 7]`
- **Tuple:** Collection of elements having different or same data types.  
Eg: `z = ("python", "java", 2); c = 9, 8, 'A'`

# Data Types

- **Dictionary:** Dictionaries are kind of hash table type. They work like associative arrays and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Eg: `p = {'A': "Apple", 1: "101"}; q = {1: ['A', 'B'], 'C': ("Ram", 23)}`

- **Set:** Set is an unordered collection of unique values.

Eg: `a = {1, 2, 3, 'a', 'b', 4, 5}`

- **Constants:** Python has built-in Constants.
  - **True:** The true value of the built-in type `bool`.
  - **False:** The false value of the built-in type `bool`.
  - **None:** It is used to signal that a value is absent.

# Quiz

Which of the following is an invalid variable?

- A. my\_String
- B. str1
- C. 1st\_string
- D. myString

# Quiz

All keywords in python are in

- A. Lower Case
- B. Upper Case
- C. Capitalized
- D. Titled Case

# Quiz

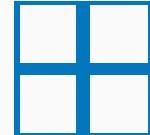
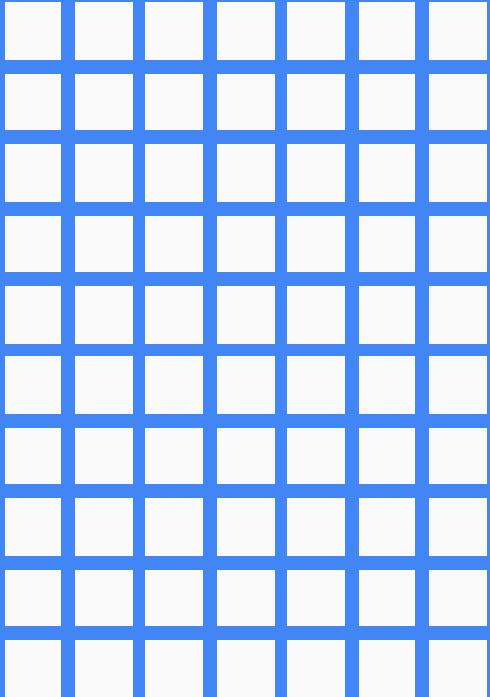
Which of the following is an invalid statement?

- A.  $abc = 1000$
- B.  $x \cdot y = 100 \cdot 200$
- C.  $a, b, c = 100, 200, 300$
- D.  $x \cdot y = 3000$

# Quiz

Which of the following cannot be a variable?

- A. if
- B. IF
- C. print
- D. IF\_Else



# Numbers

# Numbers

- Number data types store numeric values. Number objects are created when you assign a value to them.
- They are immutable data types, means that changing the value of a number data type results in a newly allocated object.
- You can also delete the reference to a number object by using the **del** statement. Example:

```
x = 23  
del x
```

- Python supports following number types:
- Integers (int): They are positive or negative whole numbers with no decimal point. Eg: **x = 23; y = 67; z = -89**

# Numbers

- Float: They represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 ( $2.5\text{e}2 = 2.5 \times 10^2 = 250$ )  
Eg: `x = 2.5; y = -67.45`
- Complex: They are of the form  $a + bJ$ , where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Eg: `x = 10 + 2j`
- Number Type Conversion:
  - `x = 23.34 → int(x) → Output: 23`
  - `y = 34 → float(y) → Output: 34.0`
  - `Z = 56 → complex(z) → Output: (56 + 0j)`

# Mathematical Functions

- These are built-in functions:
- $\text{abs}(x)$ : Returns the absolute (positive) value of  $x$ .
  - Eg:  $\text{abs}(-34)$  → Output: 34
  - Eg:  $\text{abs}(34)$  → Output: 34
- $\text{max}(x_1, x_2\dots)$ : Returns the maximum value from the tuple.
  - Eg:  $\text{max}(-34, 34)$  → Output: 34
- $\text{min}(x_1, x_2\dots)$ : Returns the minimum value from the tuple.
  - Eg:  $\text{min}(-34, 34)$  → Output: -34
- $\text{round}(x [, n])$ : Returns the rounded value of decimal numbers.  $n$  stands for how many digits to round-off.
  - Eg:  $\text{round}(-34.33)$  → Output: -34
  - Eg:  $\text{round}(34.45567, 3)$  → Output: 34.456

## Mathematical Functions

Before we proceed to get insight in further Mathematical Functions, we will see what are the Modules.

**Modules:** This is just a simple python file saved with `.py` extension in which many functions are readily defined. To use these functions we have to import these modules. There are following ways to import Modules and use pre-defined functions from these modules:

We will use `math` module and will use functions defined in this module.

## Mathematical Functions

```
import math # this is First way to import math module
```

```
math.sqrt(4) → Output: 2.0 # to use the function, we will code  
# module_name.function_name()
```

```
import math as m # this is Second way to import math module, here we  
# created m called as alice for math module.
```

```
m.sqrt(4) → Output: 2.0 # to use the function, we will code  
# alice_name.function_name()
```

## Mathematical Functions

```
from math import * # this is Third way to import math module, here we  
# can import all functions using * or the specified  
# function which only will be useful for our projects.
```

```
sqrt(4) → Output: 2.0 # to use the function, we will code  
# function_name()
```

```
from math import sqrt  
sqrt(4) → Output: 2.0
```

# Mathematical Functions

To use these methods: Import ***math*** module

- `ceil(x)`: Returns the ceiling of x. The smallest integer not less than x.
- `floor(x)`: Returns the floor of x. The largest integer not greater than x.
- `exp(x)`: Returns the exponential of x:  $e^x$
- `log(x)`: Returns the natural logarithm of x for  $x > 0$
- `log10(x)`: Returns the base-10 logarithm of x for  $x > 0$
- `pow(x, y)`: Returns the value of  $x^{**}y$
- `sqrt(x)`: Returns the square root of x
- `factorial(x)`: Returns the factorial value of x

# Mathematical Functions

```
import math
```

```
math.ceil(3.33) → Output: 4
```

```
math.ceil(-3.33) → Output: -3
```

```
math.floor(5.55) → Output: 5
```

```
math.floor(-5.55) → Output: -6
```

```
math.exp(3) → Output: 20.085536923187668
```

```
math.log(2) → Output: 0.6931471805599453
```

```
math.log10(2) → Output: 0.3010299956639812
```

```
math.pow(2, 3) → Output: 8.0
```

```
math.sqrt(55) → Output: 2.23606797749979
```

```
math.factorial(3) → Output: 6
```

# Trigonometric Functions

To use these methods: Import **math** module

- `sin(x)`: Returns the sine of x in radians
- `cos(x)`: Returns the cosine of x in radians
- `tan(x)`: Returns the tangent of x in radians
- `hypot(x, y)`: Calculates the length of the hypotenuse
- `sinh(x)`: Returns the hyperbolic sine of x in radians
- `cosh(x)`: Returns the hyperbolic cosine of x in radians
- `tanh(x)`: Returns the hyperbolic tangent of x in radians

# Inverse Trigonometric Functions

To use these methods: Import **math** module

- `asin(x)`: Returns the arc sine of x in radians
- `acos(x)`: Returns the arc cosine of x in radians
- `atan(x)`: Returns the arc tangent of x in radians
- `asinh(x)`: Returns the hyperbolic arcsine of x in radians
- `acosh(x)`: Returns the hyperbolic arccosine of x in radians
- `atanh(x)`: Returns the hyperbolic arctangent of x in radians

## Other Functions

- `degrees(x)`: Converts angle x in radians to degrees
- `radians(x)`: Converts angle x in degrees to radians
- `Pi → (math.pi)`: Returns the pi value
- `E → (math.e)`: Returns the e value

To define constants:

- `math.inf → To define infinity constant.`  
→ Eg: `a = math.inf → a → Output: inf`
- `math.nan → To define not-a-number constant.`  
→ Eg: `b = math.nan → b → Output: nan`

# Random Number Functions

To use these methods: Import **random** module

- `choice(sequence)`: Select a random item from a sequence like list, tuple or string
  - **Syntax:** `choice((tuple)), choice([list]), choice("string")`
- `randrange(start, stop, step)`: Returns a randomly selected element from range. {start: included, stop: excluded, step: Steps between the numbers}
- `random()`: Returns the random number such that **0 < number < 1**
- `seed()`: Sets the integer starting value used in generating the random numbers i.e it will create a fixed state.
- `shuffle(list)`: Randomizes items of the list in place
- `uniform(x, y)`: Returns the random number between lower-bound x and upper-bound y such that **x < number < y**

# Random Number Functions

To use these methods: Import **random** module

- `sample(sequence, k)`: Select a random elements from a sequence like list, tuple or string and we can specify how many items we want, here is *k*
  - Syntax: `sample((tuple), k)`, `sample([list], k)`, `sample("string", k)`
- `randint(start, stop)`: Returns a randomly selected element from range.  
{start: included, stop: included}

## Random Number Functions

```
import random
```

```
random.choice((2, 3, 4)) → Output: 3
```

```
random.choice([2, 3, 4]) → Output: 4
```

```
random.choice("HELLO") → Output: 'E'
```

```
random.sample((2, 3, 4, 5, 6), 3) → Output: [3, 4, 2]
```

```
random.sample([2, 3, 4, 5, 6, 7], 2) → Output: [4, 6]
```

```
random.sample("HELLO", 3) → Output: ['O', 'E', 'L']
```

```
random.randrange(1, 45, 4) → Output: 21
```

```
random.randint(1, 45) → Output: 34
```

## Random Number Functions

```
import random
```

```
random.random() → Output: 0.8088230111268553
```

```
random.seed(21)
```

```
random.uniform(1, 7) → Output: 2.3921312675038964
```

```
random.uniform(1, 7) → Output: 2.3921312675038964
```

```
a = [23, 56, 78, 89]
```

```
print(a) → Output: [23, 56, 78, 89]
```

```
random.shuffle(a)
```

```
print(a) → Output: [78, 56, 23, 89]
```

# Quiz

To use 'random' module, what should we do from following?

- A. import random
- B. random.h
- C. import.random
- D. random.random

# Quiz

What will be the output?

```
random.choice(2, 3, 4)
```

- A. An integer other than 2, 3, 4
- B. Either 2, 3 or 4
- C. Error
- D. None of the Above

# Quiz

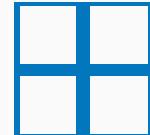
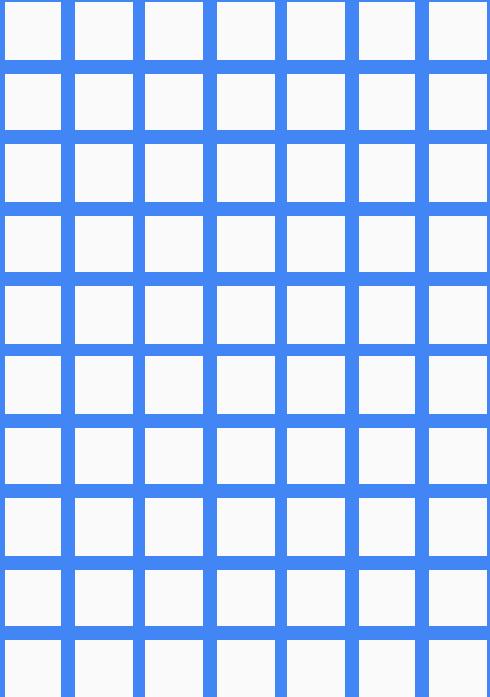
What will be the **possible** output?  
`random.randrange(1, 100, 10)`

- A. 32
- B. 67
- C. 91
- D. 80

# Quiz

What will be the output?  
`round(0.5) - round(-0.5)`

- A. 1
- B. 2
- C. 0
- D. -1



# Operators

# Operators in Python

- Operator: A symbol or function which performs a specific task/operation and yields results
- Types of Operators:
- Arithmetic
- Comparison
- Assignment
- Logical
- Bitwise
- Membership
- Identity

## Arithmetic Operators

- Example:  $a = 10; b = 20$
- Addition:  $a + b \rightarrow \text{Output: } 30$
- Subtraction:  $a - b \rightarrow \text{Output: } -10$
- Multiplication:  $a * b \rightarrow \text{Output: } 200$
- Division:  $a / b \rightarrow \text{Output: } 0.5$
- Modulus:  $a \% b \rightarrow \text{Returns remainder} \rightarrow a \% b \rightarrow \text{Output: } 10$
- Floor Division:  $a // b \rightarrow \text{Returns quotient/result of division in Integer value.}$   
But if one of the operands is negative, the result is floored, i.e. rounded away from zero  $\rightarrow a // b \rightarrow \text{Output: } 0$
- Exponent:  $a ** b \rightarrow 10**20 \rightarrow \text{Output: } 10000000000000000000000000000000$

# Comparison Operators

- Example: `a = 10; b = 20`
- Equal to: `a == b` → Output: False
- Not equal to: `a != b` → Output: True
- Greater than: `a > b` → Output: False
- Greater than or equal to: `a >= b` → Output: False
- Less than: `a < b` → Output: True
- Less than or equal to: `a <= b` → Output: True

# Assignment Operators

- Example:  $a = 10; b = 20$
- $c = a + b \rightarrow$  Result of  $a + b$  will assign to  $c \rightarrow c \rightarrow 30$
- $c += a \rightarrow$  Similar to  $c = c + a$
- $c -= a \rightarrow$  Similar to  $c = c - a$
- $c *= a \rightarrow$  Similar to  $c = c * a$
- $c /= a \rightarrow$  Similar to  $c = c / a$
- $c %= a \rightarrow$  Similar to  $c = c \% a$
- $c // a \rightarrow$  Similar to  $c = c // a$
- $c **= a \rightarrow$  Similar to  $c = c ** a$

# Bitwise Operators

- Example:  $a = 10$ ;  $b = 20$
- Binary representation:  $a \rightarrow 0000\ 1010$     $b \rightarrow 0001\ 0100$
- & binary AND:  $a \& b \rightarrow 0000\ 1010 \& 0001\ 0100 \rightarrow$  Output: 0
- | binary OR:  $a | b \rightarrow 0000\ 1010 | 0001\ 0100 \rightarrow 0001\ 1110$   
→ Output: 30
- ^ binary XOR:  $a ^ b \rightarrow 0000\ 1010 ^ 0001\ 0100 \rightarrow 0001\ 1110$   
→ Output: 30
- $\sim$  Binary Two's Complement:  $\sim a \rightarrow \sim 0000\ 1010 \rightarrow$  Output: -11
- << Binary Left Shift:  $a << 2 \rightarrow 0000\ 1010 << 2 \rightarrow 0010\ 1000$   
→ Output: 40
- >> Binary Right Shift:  $a >> 2 \rightarrow 0000\ 1010 >> 2 \rightarrow 0000\ 0010$   
→ Output: 2

## Logical Operators

- **and** Logical AND: Evaluates to the second argument if and only if both of the arguments are truthy. Otherwise evaluates to the first falsey argument.

Eg: `a = True; b = True → a and b → Output: True`

Eg: `a = True; b = False → a and b → Output: False`

Eg: `a = False; b = True → a and b → Output: False`

Eg: `a = False; b = False → a and b → Output: False`

# Logical Operators

- **or** Logical OR: Evaluates to the first truthy argument if either one of the arguments is truthy. If both arguments are false, evaluates to the second argument.

Eg: a = True; b = True → a or b → Output: True

Eg: a = True; b = False → a or b → Output: True

Eg: a = False; b = True → a or b → Output: True

Eg: a = False; b = False → a or b → Output: False

## Membership Operators

- Membership operators test for membership of an element in the sequence like string, list or tuple.
- in: Evaluates **True** if element belongs to sequence.
- not in: Evaluates **True** if element does not belong to sequence

Example: **a = [23, 45, 67]**

**23 in a → Output: True**

**44 not in a → Output: True**

## Identity Operators

- Identity operators compare the memory locations of two objects.
- `is`: Evaluates **True** if two elements point to the same object or memory location.
- `is not` : Evaluates **True** if two elements point to the different objects or memory locations.
- Example: `a = 10; b = 20; c = 12; d = 12`
- `id()` → Returns the identity or memory location of variable.
  - `id(a)` → Output: 1900008640
  - `id(b)` → Output: 1900008800
  - `id(c)` → Output: 1843489888
  - `id(d)` → Output: 1843489888
  - `a is b` → Output: False
  - `c is d` → Output: True

# Quiz

What is the value of X if:

$$X = \text{int}(43.55 + 2 / 2)$$

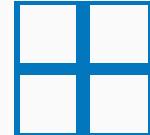
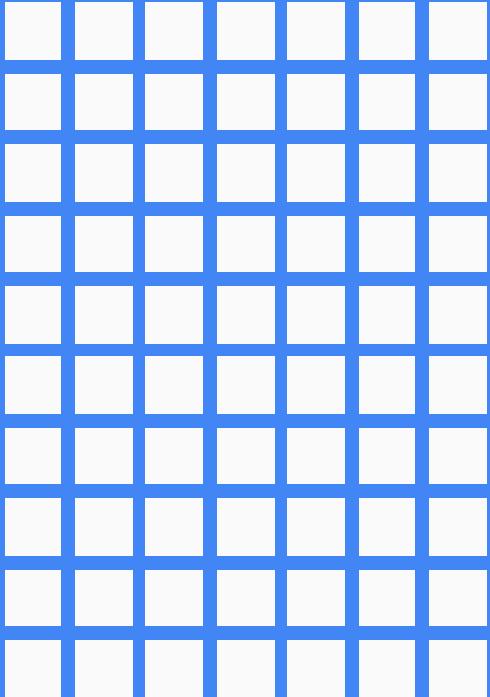
- A. 44
- B. 43
- C. 44.55
- D. 22

# Quiz

What is the value of Y if:

$$Y = \text{float}(22/3+3/3)$$

- A. 8
- B. 8.0
- C. 8.3
- D. 8.33



# Strings

# Strings

- Strings are finite sequence of characters enclosing in single or double or triple quotes.
- Python does not support character type. Python considers as a string even if it contains only one character.
- To check ASCII Value of character use the method: `ord()`
  - Eg: `ord('A')` → Output: 65
- To check which ASCII Value refers to the character use the method: `chr()`
  - Eg: `chr(90)` → Output: 'Z'
- To get the input from the user use the method: `input()`
- Eg: `s = input("Enter the String: ")` → This will prompt the message which is given as **Enter the String** and ask interpreter to type the String which will be stored in the variable `s`

## Strings Special Operators

- Example: `a = "Hello"; b = "Python"`
- + Concatenation: Joins two or more strings.  
→ `a + b` → Output: 'HelloPython'
- \* Repetition: Repeat the same string and concatenate it.  
→ `a * 2` → Output: 'HelloHello'
- [] Slice: Returns the character in the string according to the index.  
→ `a[4]` → Output: 'o'    `a[-2]` → Output: 'l'
- [ :] Range Slice: Returns characters from the given range.  
→ `a[0 : 4]` → Output: 'Hell'    `a[0 : -4]` → Output: 'H'  
→ `a[0 :]` → Output: 'Hello'    `a[-3 :]` → Output: 'llo'  
→ `a[: 4]` → Output: 'Hell'    `a[: -1]` → Output: 'Hell'  
→ `a[:]` → Output: 'Hello'    `a[-4 : -1]` → Output: 'ell'

## Strings Special Operators

- Example: `a = 'Hello'`
- `in`: Membership: Returns **True** if character present in the string
  - `'h' in a` → Output: **False**
- `not in`: Membership: Returns **True** if character not present in the string
  - `'H' not in a` → Output: **False**
- Escape or non-printable characters: `\n` → new line, `\t` → tab
- Raw String `r/R`: Suppresses the actual meaning of escape characters
  - Example: `print(r"new line \n")` → Output: **new line \n**

## Strings Boolean Methods

- **isalnum()**: Returns **True** if string contains at least one character and all characters are alphanumeric
- **isalpha()**: Returns **True** if string contains at least one character and all characters are alphabetic
- **isdigit()**: Returns true if string contains only digits
- **isnumeric()**: Returns **True** if unicode string contains only numeric chars
- **isdecimal()**: Returns **True** if unicode string contains only decimal chars
- **islower()**: Returns **True** if string is in lowercase
- **isupper()**: Returns **True** if string is in uppercase
- **isspace()**: Returns **True** if string contains only whitespace characters
- **istitle()**: Returns **True** if string consists of “titledcase”

## Strings Other Methods

- Example: `a = 'hi'; b = 'hello PYTHON'`
- **capitalize()**: Capitalizes the first letter of string and lowercase the other letters if in uppercase.
  - `a.capitalize() → Output: 'Hi'`
  - `b.capitalize() → Output: 'Hello_python'`
- **center(width, fillchar)**: Returns filled characters string with the original string centered to a total of width characters
  - `a.center(10, '*') → Output: '****hi****'`
  - `a.center(15, '*') → Output: '*****hi*****'`
- **count(str, beg=0, end=len(string))**: Counts how many times *str* occurs in a string or in a substring if *beg* and *end* are given.
  - `a = "this is python and this is a great language"`
  - `a.count("this") → Output: 2`
  - `a.count('t', 0, len(a)) → Output: 4`

## Strings Other Methods

- **join(seq)**: Returns a string in which the string elements of sequence have been joined by str separator  
→ `s = '-'; seq = ['A', 'B', 'C']; s.join(seq) → Output: 'A-B-C'`
- **len(string)**: Returns the length of the string  
→ `s = "hello_python"; len(s) → Output: 12`
- **lower()**: Converts all uppercase letters in a string to the lowercase  
→ `s = "Hello_Python"; s.lower() → Output: hello_python`
- **upper()**: Converts all lowercase letters in a string to the uppercase  
→ `s = "Hello_Python"; s.upper() → Output: HELLO PYTHON`

## Strings Other Methods

- **lstrip()**: Returns a copy of the string in which all characters have been stripped from the beginning of the string (Default: whitespace)  
→ `str1 = " this is amazing"` → `str.lstrip()` → Output: 'this is amazing'
- **rstrip()**: Returns a copy of the string in which all characters have been stripped from the end of the string (Default: whitespace)  
→ `str1 = "this is amazing "` → `str.rstrip()` → Output: 'this is amazing'
- **strip()**: Returns a copy of the string in which all characters have been stripped from the beginning and end of the string (Default: whitespace)  
→ `str1 = "88 this is amazing 88"` → `str1.strip('8')`  
→ Output: 'this is amazing'

## Strings Other Methods

- **max()**: Returns the max alphabetical character from the string  
→ `max('hello')` → Output: 'o'
- **min()**: Returns the min alphabetical character from the string  
→ `max('hello')` → Output: 'e'
- **swapcase()**: Returns a copy of the string in which cases of characters will be swapped  
→ `s = "heLLo"` → `s.swapcase()` → Output: 'HEllo'
- **title()**: Returns a copy of the string in the titled cased  
→ `s = "hi! how are you?"` → `s.title()` → Output: 'Hi! How Are You?'

## Strings Other Methods

- **zfill(width)**: Pads string on the left with zeros to fill the width  
→ `a = "hello" → a.zfill(10) → Output: '00000hello'`
- **ljust(width [, n])**: Returns the string left justified in a string of length width  
→ `a = "hello" → a.ljust(10, '*') → Output: 'hello*****'`
- **rjust(width [, n])**: Returns the string right justified in a string of length width  
→ `a = "hello" → a.rjust(10, '*') → Output: '*****hello'`
- **replace(old, new, [, max])**: Replaces all occurrences of old characters in string with new characters or at most max occurrences if max is given
  - `a = "this is an apple" → a.replace('an', 'the')`
  - `Output: 'this is the apple'`
  - `a = "this is an banana" → a.replace('an', 'a', 1)`
  - `Output: 'this is a banana'`

## Strings Other Methods

- **maketrans()**: Returns a translation table that maps each character in the *intab* string into the character at the same position in the *outtab* string.  
Both *intab* and *outtab* must have the same length.
  - `intab = 'ae'; outtab = '12'`
  - `str1 = "this is an eye"`
  - `str1.translate(str1.maketrans(intab, outtab))`
  - `Output: 'this is 1n 2y2'`
- **translate()**: Translates the string according to the translation table using **maketrans()**

## Strings Other Methods

- **startswith(suffix, beg=0, end=len(string))**: Returns **True** if string starts with particular suffix or in between matching boundary if *beg* and *end* are given.
  - `str1 = "this is wow!" → str1.startswith('this')`
  - **Output: True**
- **endswith(suffix, beg=0, end=len(string))**: Returns **True** if string ends with particular suffix or in between matching boundary if *beg* and *end* are given
  - `str1 = "this is wow!" → str1.endswith('this')`
  - **Output: False**

## Strings Other Methods

- **find(str, beg=0, end=len(string)):** It determines if *str* occurs in a string or substring of string if starting index *beg* and ending index *end* are given. It returns the index if *str* found otherwise will return -1
  - `str1 = "this is wow!"; str2 = "is"` → `str1.find(str2)` → Output: 2
  - `str1.find('are')` → Output: -1
- **index(str, beg=0, end=len(string)):** It determines if *str* occurs in a string or substring of string if starting index *beg* and ending index *end* are given. It raises an exception if *str* is not found.
  - `str1 = "this is wow!"; str2 = "is"` → `str1.index(str2)` → Output: 2
  - `str1.index('are')` → Output: ValueError: substring not found

## Strings Other Methods

- **rfind(str, beg=0, end=len(string)):** Returns the last index where the substring str is found, or -1 if no such index exists.
  - `str1 = "this is wow!"; str2 = "is" → str1.rfind(str2) → Output: 5`
  - `str1.rfind('are') → Output: -1`
- **rindex(str, beg=0, end=len(string)):** Returns the last index where the substring str is found, or raises an exception if no such index exists.
  - `str1 = "this is wow!"; str2 = "is" → str1.rindex(str2) → Output: 5`
  - `str1.rindex('are') → Output: ValueError: substring not found`

## Strings Other Methods

- **split(str, num=string.count(str)):** Returns a list of all words in the string split using *str* as the separator (if not specified, default is space) or split into at most *num* (number of) substrings if given.
  - `str1 = "If there is a will, there will be a way!"`
  - `str1.split()`
  - Output: `['If', 'there', 'is', 'a', 'will,', 'there', 'will', 'be', 'a', 'way!']`
  - `str1.split(',')`
  - Output: `['If there is a will', ' there will be a way!']`

## Strings Other Methods

- **encode(encoding='utf-8', errors='strict')**: Returns an encoded version of the string. The default for errors is 'strict' meaning that encoding error will raise a UnicodeError or ValueError. Other possible values are 'ignore' and 'replace'

Example:

```
→ str1 = "If there is a will, there will be a way!"  
→ str1.encode()  
→ Output: b'If there is a will, there will be a way!'
```

## Strings Other Methods

- **partition(sep)**: Split the string at the first occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.
  - `str1 = "If there is a will, there will be a way!"`
  - `str1.partition("will")` → Output: ('If there is a ', 'will', ', there will be a way!')
- **rpartition(sep)**: Split the string at the last occurrence of sep, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.
  - `str1 = "If there is a will, there will be a way!"`
  - `str1.rpartition("will")` → Output: ('If there is a will, there ', 'will', ' be a way!')

## Strings Formatting

**Consider the Example:**

```
a = "I am Ram, 23 years old. My height is 5.5 feet and weight is 56 kg"  
print(a)
```

Output: I am Ram, 23 years old. My height is 5.5 feet and weight is 56 kg

Above String can be written using String Formatting Concepts when we occur to include Characters, Numbers or Decimal Values into the String.

```
print("I am %s, %d years old. My height is %.2f feet and weight is %d kg"  
% ("Ram", 23, 5.5, 56))
```

Output: I am Ram, 23 years old. My height is 5.5 feet and weight is 56 kg

## Strings Formatting

Use String Formatting Method whenever, there is a need to include numbers and strings into the String.

%d → It indicates to include Digits

%f → It indicates to include Float Values

%.2f → It indicates to include Float Values, but 2 indicates that how many digits you want after decimal point

%s → It indicates String values

Example:

a = 34; b = 56

c = a + b

print("Addition of %d and %d is %d" %(a, b, c))

Output: Addition of 34 and 56 is 90

# Quiz

What is the output of:

```
s = "goodday"  
s.find('d')  
s.rfind('d')
```

- A. 2 3
- B. 3 4
- C. -3 -4
- D. None

# Quiz

What is the output of:  
`print("hello"+1+2+3)`

- A. hello123
- B. hello
- C. Error
- D. hello6

# Quiz

What is the output of:

```
s = "xyyzxyzxzxxy"  
print(s.count('xyy', -10, -1))
```

- A. 2
- B. 0
- C. Error
- D. 1

# Quiz

What is the output of:

```
print('ab12'.isalpha())
print('ab12'.isalnum())
print('12 12'.isdigit())
```

- A. True True True
- B. False True True
- C. False True False

# Quiz

What is the output of:

```
s = "xyxxyyzxxyz"  
print(s.lstrip('xyy'))
```

- A. zxxyz
- B. xyxxyyzxxyz
- C. xyxzxxyz
- D. None

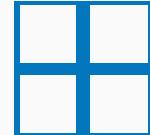
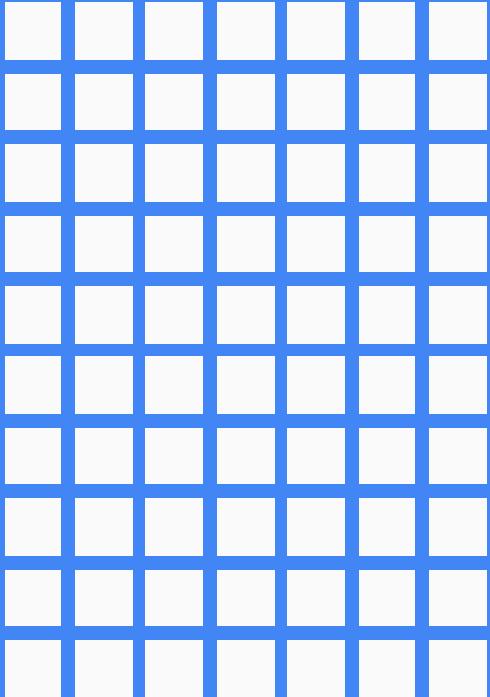
# Quiz

s = "Hello"

What will be the return type when:

print(type(s))

- A. int
- B. bool
- C. String
- D. str



## Decision Making

# Decision Making

- **Decision Making:** The if-elif-else statement is used to conditionally execute a statement or a block of statements. Conditions can be **TRUE** or **FALSE**, execute one thing when the condition is *true*, something else when the condition is *false*.
- **If Statement:** The **if** statement evaluates the condition followed by block of statements to execute when condition is **TRUE**.
- **If...else Statement:** The **if...else** statement evaluates the condition followed by block of statements to execute when condition is **TRUE**, otherwise **else** block of statements will execute when condition is **FALSE**.
- **If...elif...else Statement(Nested if):** Nested if is used when there is two or more conditions to be checked.

## Decision Making

- **Ternary Operator:** The ternary operator is used for inline conditional expressions.

Example:

n = 5

"Greater than 2" if n > 2 else "Smaller than or equal to 2"

Output: 'Greater than 2'

n = 5

"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"

Output: 'Good day'

# Decision Making

- If Statement:

```
if condition_is TRUE:  
    Statement1_will_execute
```

- If-Else Statement:

```
if condition_is TRUE:  
    Statement1_will_execute  
else:  
    Statement2_will_execute
```

# Decision Making

- If-elif Else Statement:

`if condition1_isTRUE:`

*Statement1\_will\_execute*

`elif condition2_isTRUE:`

*Statement2\_will\_execute*

`else:`

*Statement3\_will\_execute*

# Decision Making

- *and-or* operators in the if statement:

x = True

y = False

```
if x and y:  
    print("Both x & y are TRUE")  
elif x or y:  
    print("One of the x or y is TRUE")  
else:  
    print("This is the end of the conclusion")
```

Output: One of the x or y is TRUE

## Decision Making

- *in* operator in the if statement:

```
a = ["Python", 'C', "Java"]  
b = "C"
```

```
if b in a:  
    print(b, "Tutorial")  
else:  
    print("This is the end of the conclusion")
```

Output: C Tutorial

## Decision Making

- **Negative if:**

a = 20

```
if not a == 20:  
    print("Value of a is different a:", a)  
else:  
    print("Yes! value of a is", a)
```

Output: Yes! value of a is 20

# Decision Making

- Testing with Boolean and NoneType data types:

```
if True:  
    print("It is True")  
else:  
    print("It is False")
```

Output: It is True

```
if False:  
    print("It is False")  
else:  
    print("It is True")
```

Output: It is False

## Decision Making

- Testing with Boolean and NoneType data types:

a = None

```
if a is None:  
    print("It is NoneType")  
else:  
    print("It is not")
```

Output: It is NoneType

# Quiz

Guess the Output!

```
m = "ABC"  
if m.isupper():  
    print("Welcome")  
else:  
    print("Bye")
```

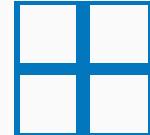
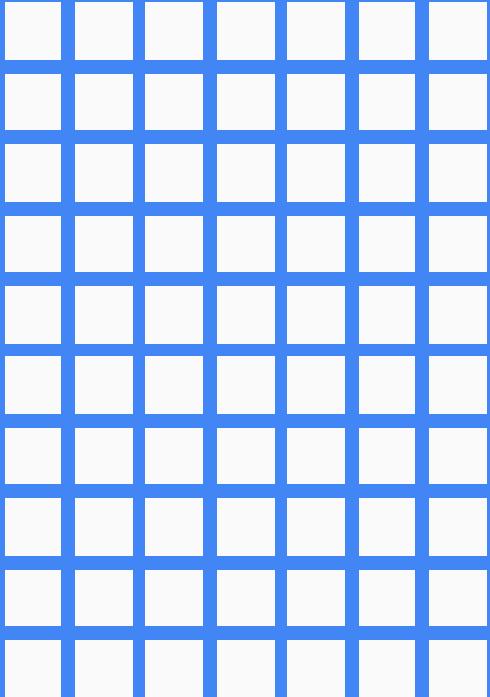
- A. Welcome
- B. Bye

# Quiz

Guess the Output!

```
m = "ABC"  
if 'b' in m:  
    print("Welcome")  
else:  
    print("Bye")
```

- A. Welcome
- B. Bye



# Loops

# Loops

- **Loops:** Loop is used to iterate over the items of any sequence including the Python list, string, tuple etc. The loop is also used to access elements from a container (for example list, string, tuple) using built-in function range(). A loop statement allows us to execute a statement or group of statements multiple times.
- **For loop:** For loop is used to execute the block of code/statements multiple times till the iteration count. For loop can be used most times when number of iterations is known.
- **While loop:** While loop repeats the block of statements while the given condition is True. While loop is used when number of iterations is unknown.

# Loops

- While Loop:

`while condition:`

*Statement1\_will\_execute*

- While Loop with Else Statement:

`while condition:`

*Statement1\_will\_execute*

`else:`

*Statement2\_will\_execute*

# Loops

- For Loop:

```
for iterator_variable in iterator_sequence(tuple/list):
    Statement1_will_execute
```

- Nested For Loop

```
for iterator_variable1 in iterator_sequence:
    for iterator_variable2 in iterator_sequence:
        Statement1_will_execute
    #1st Loop ends
#2nd Loop ends
```

# Loops

- For Loop with Else Statement:

```
for iterator_variable in iterator_sequence(tuple/list):
    Statement1_will_execute
else:
    Statement2_will_execute
```

# Loops

- For Loop:

```
list1 = ["orange", "apple", "garlic", "tomatoes", "mangoes"]
```

```
for i in list1:  
    print("Name of string:", i, "and", "Length of string:", len(i))
```

## Output:

```
Name of string: orange and Length of string: 6  
Name of string: apple and Length of string: 5  
Name of string: garlic and Length of string: 6  
Name of string: tomatoes and Length of string: 8  
Name of string: mangoes and Length of string: 7
```

# Loops

- While Loop:

```
list1 = ["orange", "apple", "pineapple"]
```

```
count = 0
while count < len(list1):
    print("Name of string:", list1[count])
    count = count + 1
```

Output:

```
Name of string: orange
Name of string: apple
Name of string: pineapple
```

# Loops

- **range() function:** The range() function returns a list of consecutive integers. The function has one, two or three parameters where last two parameters are optional. It is widely used in for loops. It generates arithmetic progressions.

Syntax:

`range(start, stop, step(increment))`

start: Start the sequence of numbers from, including

stop: Stop the sequence of numbers, excluding

step: Generate the numbers in the sequence incrementing by

`range(a):` Generates a sequence of numbers from 0 to a, excluding a, incrementing by 1.

# Loops

- range() function example:

```
range(5) ⇒ 0, 1, 2, 3, 4
```

```
range(5, 10) ⇒ 5, 6, 7, 8, 9
```

```
range(0, 10, 3) ⇒ 0, 3, 6, 9
```

```
range(-10, -100, -30) ⇒ -10, -40,  
-70
```

```
a = ['Mary', 'is' , 'clever']  
for i in range(len(a)):  
    print(i, a[i])
```

Output:

0 Mary

1 is

2 clever

# Loops

- Iterating over tuple:

```
tup1 = (1, 2, 3, 4, 5, 6, 7, 8, 9)
count_odd = 0
count_even = 0

for x in tup1:
    if x % 2 == 0:
        count_even = count_even + 1
    else:
        count_odd = count_odd + 1
print("Number of even numbers:", count_even)
print("Number of odd numbers:", count_odd)
```

Output:

Number of even numbers: 4

Number of odd numbers: 5

# Loops

- Iterating over dictionary:

```
dict1= {"id1": "nandini", "id2": "rahul", "id3": "rohan"}
```

```
for x in dict1:  
    print(x)
```

Output:  
id1  
id2  
id3

```
for x in dict1.values():  
    print(x)
```

Output:  
nandini  
rahul  
rohan

# Loops

- Iterating over dictionary:

```
dict1= {"id1": "nandini", "id2": "rahul", "id3": "rohan"}
```

```
for x in dict1.keys():
    print(x)
```

Output:  
id1  
id2  
id3

```
for x in dict1.items():
    print(x)
```

Output:  
('id1', 'nandini')  
('id2', 'rahul')  
('id3', 'rohan')

# Loop Control Statements

- **Break Statement:** The break statement is used to exit a for or a while loop. The purpose of this statement is to end the execution of the loop (for or while) immediately and the program control goes to the statement after the last statement of the loop. If there is an optional else statement in while or for loop it skips the optional clause also.
- **Continue Statement:** The continue statement is used in a while or for loop to take the control to the top of the loop without executing the rest statements inside the loop.
- **Pass Statement:** Pass statement is used when there is a requirement of declaring control statements, however no need of giving block of statements i.e. empty for/while loops of if..else statements.

## Loop Control Statements

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)
num_sum = 0
count = 0
for x in numbers:
    num_sum = num_sum + x
    count = count + 1
    if count == 5:
        break
print("Sum of first",count,"integers
is:", num_sum)
```

Output:

Sum of first 5 integers is: 15

```
num = (1, 2, 3, 4, 5, 6, 7, 8, 9)
num_sum = 0
count = 0
while count<len(num):
    num_sum = num_sum + num[count]
    count = count + 1
    if count== 5:
        break
print("Sum of first",count,"integers is:",
num_sum)
```

Output:

Sum of first 5 integers is: 15

# Loop Control Statements

```
for x in range(7):
    if x == 3 or x==6:
        continue
    print("Number:", x)
```

Output:

Number: 0  
Number: 1  
Number: 2  
Number: 4  
Number: 5

```
x = 0
for i in range(20):
    pass

if x == True:
    pass

if x < 0:
    pass

elif x > 100:
    pass
```

# Quiz

Guess the Output!

i= 1

while True:

    print(i)

- A. 1
- B. Error
- C. Infinite Loop

# Quiz

Guess the Output!

```
x = 2  
for i in range(x):  
    x = x + 1  
    print(i)
```

- A. 0 1
- B. 0 1 2 3
- C. Infinite Loop

# Quiz

Guess the Output!

```
x = "ABC23DE45"
```

```
for i in x:  
    if i.isdigit():  
        break  
    else:  
        print(i)
```

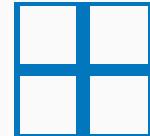
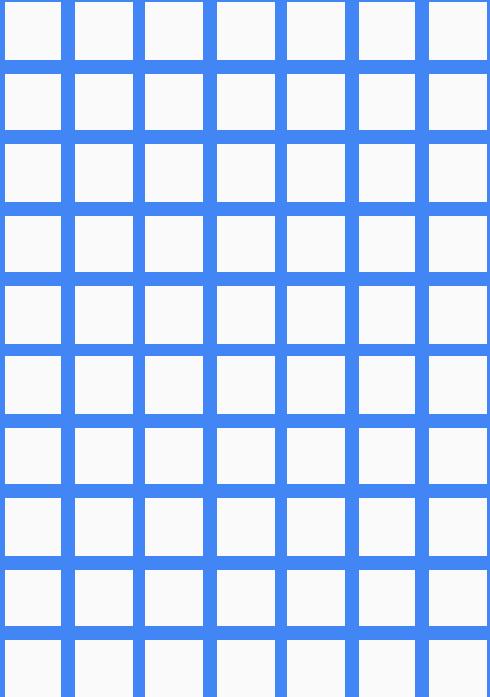
- A. A B C
- B. A B C D E
- C. 23

# Quiz

Guess the Output!

```
c = 0
x = "siri123siri"
for i in x:
    if i.isalpha():
        c = c + 1
    else:
        break
print(c)
```

A. 4  
B. 8



List

# List

- List: Collection of comma-separated values/elements/items enclosed in the square brackets where elements need not all have same data-types.  
→ Example: `list1 = ["python", "java", 12, 25.5]`

0	1	2	3
python	java	12	25.5
-4	-3	-2	-1

- Accessing the List: Using index or slicing
  - `list1[0]` → Output: 'python'
  - `list1[0: -1]` → Output: ['python', 'java', 12]
  - `list1[1: 3]` → Output: ['java', 12]
  - `list1[-2: ]` → Output: [12, 25.5]
  - `list1[1:]` → Output: ['java', 12, 25.5]
  - `list1[-4: -1]` → Output: ['python', 'java', 12]

# List

- Updating the List:

→ `list1[3] = "Hello"`

→ `list1 → Output: ['python', 'java', 12, 'Hello']`

0	1	2	3
python	java	12	Hello
-4	-3	-2	-1

→ `list1[1] = "C++"`

→ `list1 → Output: ['python', 'C++', 12, 'Hello']`

0	1	2	3
python	C++	12	Hello
-4	-3	-2	-1

# List

- Deleting the list:
  - `del list1[3]`
  - `list1` → Output: `['python', 'C++', 12]`

0	1	2
python	C++	12
-3	-2	-1

- `list1.clear()`: Removes all items from the list → `list` → Output: `[ ]`
- `list2 = []` → Empty List
- `del list1`: Whole list will be deleted

## List: Operations on List

→ `a = [1, 2, 3]`

- `len(a)` → Output: 3 → Returns the length of the list

→ `b = [4, 5, 6]`

- `a + b` → Output: [1, 2, 3, 4, 5, 6] → Concatenation of two lists

- `a * 3` → Output: [1, 2, 3, 1, 2, 3, 1, 2, 3] → Repetition of the list

- Membership Operator `in`: `3 in a` → Output: True → Returns True as 3 is present in the list

- Iteration of list:

```
for x in a:  
    print(x)
```

Output:

1

2

3

To access all the elements of list →

## List: Methods

- **list.append(item)**: Add the item to the end of the list.  
→ `|l1 = [1, 2, 3] → l1.append(4) → l1 → Output: [1, 2, 3, 4]`
- **list.count(item)**: Returns the count of how many times *item* occurs in list.  
→ `|l1 = [1, 2, 3, 3, 3, 4] → l1.count(3) → Output: 3`
- **list.extend(sequence)**: Extends the list with the another list or sequence.  
→ `|l1 = [1, 2]; l2 = [4, 5] → l1.extend(l2) → l1 → Output: [1, 2, 4, 5]`  
→ `|l1 = [1, 2, 3]; l1.extend((6, 7)) → l1 → Output: [1, 2, 3, 6, 7]`  
→ `|l1 = [1, 2]; l1.extend('BYE') → l1 → Output: [1, 2, 'B', 'Y', 'E']`
- **list.index(item)**: Returns the lowest index of the item in the list.  
→ `|l1 = [1, 2, 3, 3, 3, 4] → l1.index(3) → Output: 2`

## List: Methods

- **list.insert(index, item)**: Inserts the item at the specific index position in list.  
If we insert an index out of bound it will append the item.
  - `|l1 = [1, 2, 3] → l1.insert(1, "End") → l1 → Output: [1, 'End', 2, 3]`
  - `|l1 = [1, 2, 3] → l1.insert(7, "End") → l1 → Output: [1, 2, 3, 'End']`
- **list.remove(item)**: Removes first occurrence of the item from the list.
  - `|l1 = [1, 2, 3, 3, 3, 4] → l1.remove(3) → l1 → Output: [1, 2, 3, 3, 4]`
- **list.pop()**: Removes and returns the last element from the list.
  - `|l1 = [1, 2, 3, 3, 3, 4] → l1.pop() → Output: 4 → |l1 → [1, 2, 3, 3, 3]`
- **list.pop(index)**: Removes and returns the item from the given index.
  - `|l1 = [1, 2, 4, 5] → l1.pop(0) → Output: 1 → |l1 → [2, 4, 5]`
  - `|l1 = [1, 2, 3, 4] → l1.pop(3) → Output: 4 → |l1 → [1, 2, 3]`

## List: Methods

- `list.reverse()`: Reverses the list.  
→ `l1 = [1, 2, 3, 4, 5] → l1.reverse() → l1 → Output: [5, 4, 3, 2, 1]`
- `list.sort()`: Sorts the list. If list contains only numbers or characters, list will get sorted either ascending or descending. However, if list contains numbers and characters both, it will generate an error.  
→ `l1 = [5, 8, 1, 9] → l1.sort() → l1 → Output: [1, 5, 8, 9]`  
→ `l1 = [5, 8, 1, 9] → l1.sort(reverse=True) → l1 → Output: [9, 8, 5, 1]`  
→ `l2 = ['Z', 'B', 'P', 'T', 'E'] → l2.sort()`  
→ `print(l2) → Output: ['B', 'E', 'P', 'T', 'Z']`  
→ `l2 = ['Z', 'B', 'P', 'T', 'E'] → l2.sort(reverse=True)`  
→ `print(l2) → Output: ['Z', 'T', 'P', 'E', 'B']`  
→ `l3 = ['Z', 'B', 'P', 66, -78] → l3.sort() → Output: TypeError: '<' not supported between instances of 'int' and 'str'`

## List: Methods

- `list.copy()`: Returns a shallow copy of the list.
  - `l1 = [11, 22] → l2 = l1.copy() → print(l2)`
  - Output: [11, 22]
  - `l2[0] = 44 → print("l2:", l2, "l1:", l1)`
  - Output: l2: [44, 22] l1: [11, 22]
- `list.clear()`: Remove all items from the list and returns empty list.
  - `l1 = [1, 2, 3, 3, 3, 4] → l1.clear() → l1 → Output: []`
- Lists are Mutable i.e. we can change any item in the list.
  - `l1= [11, 22, 33] → print(l1) → Output: [11, 22, 33]`
  - `l1[1] = 55 → print(l1) → Output: [11, 55, 33]`

## List: Built-in Functions

- **len(list)**: Gives the total length of the list.  
→ `l1 = [1, 2, 3, 4, 5] → len(l1) → Output: 5`
- **max(list)**: Returns item from the list with max value.  
→ `l1 = [1, 2, 3, 4, 5] → max(l1) → Output: 5`  
→ `l2 = ['A', 'B', 'Z'] → max(l2) → Output: 'Z'`
- **min(list)**: Returns item from the list with min value.  
→ `l1 = [1, 2, 3, 4, 5] → min(l1) → Output: 1`  
→ `l2 = ['A', 'B', 'Z'] → min(l2) → Output: 'A'`
- **list(sequence)**: Converts the sequence to the list.  
→ `t1 = (1, 2, 3, 4, 5) → print(t1) → Output: (1, 2, 3, 4, 5)`  
→ `list(t1) → Output: [1, 2, 3, 4, 5]`

- **Use of double colon [::]:**
  - `l1 = [1, 2, 3, 4, 5] → l1[0: 5: 2]` → Output: [1, 3, 5]
  - `l2 = [1, 2, 3, 4, 5] → l2[::-1]` → Output: [1, 2, 3, 4, 5]
  - `l3 = [1, 2, 3, 4, 5] → l3[4: 0: -2]` → Output: [5, 3]
- **Compare two lists:**
  - `l1 = [1, 2, 3]; l2 = [1, 2, 3] → print(l1 == l2)` → Output: True
  - `l1 = [1, 2, 3]; l2 = [3, 1, 2] → print(l1 == l2)` → Output: False
- **Getting the index of an element:**
  - `l1 = list("Hello") → print(l1)` → Output: ['H', 'E', 'L', 'L', 'O']
  - `l1.index('L')` → Output: 2
  - `l1.index('L', 2)` → Output: 3

## List

- **Enumerate:** Indexes data to keep track of indices and corresponding data mapping.

```
store_list = ["Apple", "Banana", "Mango", "Orange"]
```

```
for position, name in enumerate(store_list):  
    print(position, name)
```

```
store_dict =  
dict((position, name) for position, name in enumerate(store_list))  
print(store_dict)
```

Output:  
0 Apple  
1 Banana  
2 Mango  
3 Orange

Output:  
{0: 'Apple', 1: 'Banana', 2: 'Mango', 3: 'Orange'}

## List

- **Zip:** Creates a list of tuples by pairing up elements of lists.

```
subjects = ["Maths", "Science", "English"]
subject_count = [1, 2, 3]
```

```
subject_list = zip(subjects, subject_count)
```

```
for i in subject_list:
    print(i)
```

Output:

```
('Maths', 1)
('Science', 2)
('English', 3)
```

- **Nested Lists:**

```
nested_list = [[1, 2, 3], ['A', 'B', 'C']]
```

```
print(nested_list) → Output: [[1, 2, 3], ['A', 'B', 'C']]
```

```
print(nested_list[0]) → Output: [1, 2, 3] ← First List
```

```
print(nested_list[0][1]) → Output: 2 ← Second Element from First List
```

# List

- **List Comprehensions:** Creates a new list by applying an expression to each element of an iterable.

Basic Syntax:

```
[<expression> for <element> in <iterable>]
```

OR:

```
[<expression> for <element> in <iterable> if <condition>]
```

- **List Comprehensions:**

Example:

```
squares = [x * x for x in (1, 2, 3, 4)]  
print(squares) → Output: [1, 4, 9, 16]
```

Above code is Similar with the following one:

```
squares = []  
for x in (1, 2, 3, 4):  
    squares.append(x * x)  
print(squares) → Output: [1, 4, 9, 16]
```

- **List Comprehensions:**

Example:

```
cubes= [x * x * x for x in range(8)]
```

```
print(cubes) → Output: [0, 1, 8, 27, 64, 125, 216, 343]
```

```
up= [s.upper() for s in "HelloWorld"]
```

```
print(up) → Output: ['H', 'E', 'L', 'L', 'O', 'W', 'O', 'R', 'L', 'D']
```

# List

- **List Comprehensions:**

Example:

```
even= [x for x in range(10) if x % 2 == 0]
print(even) → Output: [0, 2, 4, 6, 8]
```

```
vowels= [x if x in "aeiou" else '*' for x in "apple"]
print(vowels) → Output: ['a', '*', '*', '*', 'e']
```

# List

- **List Comprehensions:**

Example:

```
add= [x + y for x, y in [(1, 2), (3, 4), (5, 6)]]  
print(add) → Output: [3, 7, 11]
```

```
items = ['1', '2', '3', '4']  
new_items= [int(item) for item in items]  
print(new_items) → Output: [1, 2, 3, 4]
```

# Quiz

Guess the Output!

```
L = [1, 2, 3, 4]  
print(sum(L))
```

- A. Error
- B. None
- C. 10
- D. None of these

# Quiz

To add a New Element to the List, a command is used:

- A. list1.add(4)
- B. list1.addLast(5)
- C. list1.append(9)
- D. list1.addEnd(3)

# Quiz

Guess the Output!

data = [2, 3, [5, 6]]

data[2][0]

- A. 5
- B. 2
- C. [5, 6]
- D. 6

# Quiz

Guess the Output:

```
I1 = [12, 34, 11, 45]  
I1.pop(2)
```

- A. [12, 34, 11, 45]
- B. [12, 34, 11]
- C. [12, 11, 45]
- D. [12, 34, 45]

# Quiz

Guess the Output:

```
I1 = [1, 2, 3]
I1.append(['A', 'B', 'C'])
print(I1)
```

- A. [1, 2, 3]
- B. Error
- C. [1, 2, 3, 'C']
- D. [1, 2, 3, ['A', 'B', 'C']]

# Quiz

|1 = [1, 2]

|2 = [3, 4]

|3 = [1, 2]

|4 = [5, ]

Guess the Output!

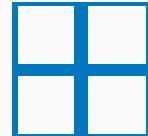
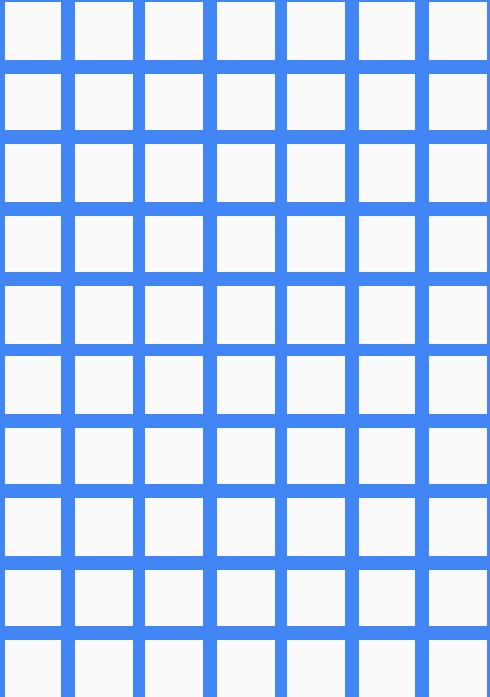
|1 + |2

|1 - |2

|1 == |3

|1 is |3

|4\*6



*Tuple*

# Tuple

- Tuple: Collection of comma-separated values/elements/items enclosed in the parentheses.
- Tuple are like lists, except they are “**Immutable**”.
- Immutable: Cannot change the content of these objects once created.
  - Example: `tup1 = ('C', 1, 2.5)`
  - Example: `tup2 = 'a', 'b', 'c'` → `print(tup2)` → Output: ('a', 'b', 'c')
- Use of parentheses is optional.
- Empty tuple: `tup3 = ()`
- Single value tuple with comma: `tup4 = (50,)`
- Single value without comma is not a Tuple: `tup4 = (50)` → integer
- The symbol `_` can be used as a disposable variable name.
  - `a = 1, 2, 3, 4` → `print(a)` → Output: (1, 2, 3, 4)
  - `_, x, y, _ = a` → `print(x, y)` → Output: 2 3

# Tuple

- Accessing the Tuple: `tup1 = ('A', 12, 'B', 45)`
  - `tup1[0]` → Output: 'A'    `tup1[0:]` → Output: ('A', 12, 'B', 45)
  - `tup1[0: 3]` → Output: ('A', 12, 'B')    `tup1[0: -2]` → Output: ('A', 12)
- Updating the tuple is not possible, as tuple is an immutable sequence.
- Example: `tup1[0] = 100` → Will raise an error.
  - Output: `TypeError: 'tuple' object does not support item assignment`
- Deleting the particular element of the tuple is not possible due to immutable property of tuple.
- Example: `del tup1[0]` → Will raise an error.
  - Output: `TypeError: 'tuple' object does not support item deletion`
- However, deleting the whole tuple is possible:
- Example: `del tup1`

## Tuple: Operations on the tuple

- **len(tuple)**: Returns the total length of the tuple.  
→ `tup1 = (1, 2, 3, 4) → len(tup1) → Output: 4`
- `tuple1 + tuple2`: Concatenation of the two tuples.  
→ `tup1 = (1, 2); tup2 = ('A', 'B', 'C') → tup1 + tup2`  
→ `Output: (1, 2, 'A', 'B', 'C')`
- `tuple1 * 3`: Repetition of the tuple elements.  
→ `tup1 = (1, 2) → tup1*3 → Output: (1, 2, 1, 2, 1, 2)`
- Membership operator **in**  
→ `tup1 = (1, 2, 3, 4) → print(2 in tup1) → Output: True`

## Tuple: Operations on the tuple

- Iteration over the tuple:

```
a = ('a', 'b', 'c')  
for x in a:  
    print(x)
```

Output:  
a  
b  
c

- max(tuple)**: Returns the item with max value.  
→ `tup1 = (1, 2, 3, 4) → max(tup1) → Output: 4`
- min(tuple)**: Returns the item with min value.  
→ `tup1 = (1, 2, 3, 4) → min(tup1) → Output: 1`
- tuple(sequence)**: Converts the sequence to the tuple.  
→ `l1 = [1, 2, 3, 4] → print(l1) → Output: [1, 2, 3, 4]`  
→ `tuple(l1) → Output: (1, 2, 3, 4)`

## Tuple: Operations on the tuple

- Add item in tuple:

```
tup1 = (3, 7, 1, 2, 9, 4, 6, 7, 2)
```

```
tup1 = tup1[:5] + (12, 15, 16) + tup1[5:]
```

```
print(tup1)
```

```
Output: (3, 7, 1, 2, 9, 12, 15, 16, 4, 6, 7, 2)
```

- Number of times an item has repeated:

```
tuple.count(item)
```

```
print(tup1.count(3))
```

```
Output: 1
```

## Tuple: Operations on the tuple

- Find the index of the item:

`tuple.index(item)`

```
print(tup1.index(16))
```

Output: 6

- Use of step parameter:

```
tup1 = (3, 7, 1, 2, 9, 4, 6, 7, 2)
```

```
print(tup1[0: 6: 3]) → Output: (3, 2)
```

```
print(tup1[:: 3]) → Output: (3, 2, 6)
```

```
print(tup1[:: -1]) → Output: (2, 7, 6, 4, 9, 2, 1, 7, 3)
```

# Quiz

Suppose  $t = (1, 2, 3, 4)$  is a Tuple. Which of the following statement is incorrect?

- A. `print(t[3])`
- B. `t[0] = 67`
- C. `print(max(t))`
- D. `print(len(t))`

# Quiz

Suppose `a = (1)`. What will be the output  
of `print(type(a))`

- A. Tuple
- B. Integer
- C. Error
- D. None

# Quiz

What type of data is:  
`p = [(1, 2, 3), (4, 5, 6)]`

- A. Array of Tuples
- B. List of Tuples
- C. Invalid Data type
- D. Tuples of List

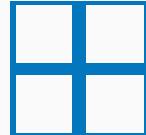
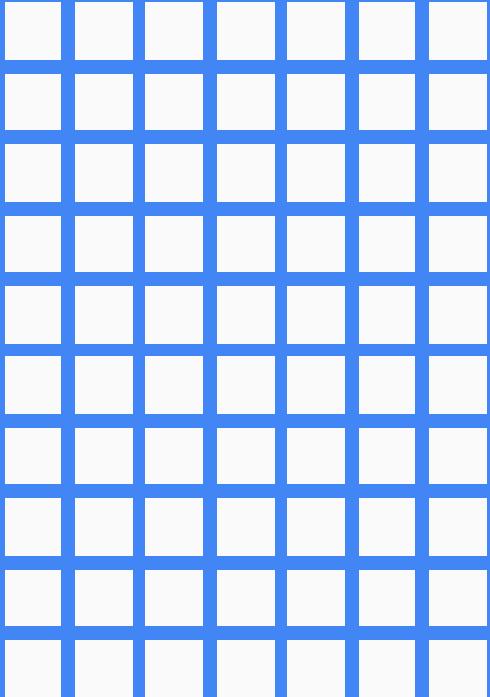
# Quiz

a, b = 1, 2

a, b = b, a

print(a, b)

- A. 1 2
- B. 2 1
- C. Can't update values
- D. Can't assign values



# Dictionary

## Dictionary

- Dictionary is collection of comma-separated set of objects. Items in the dictionary forming a **keys-values** pair enclosed in the curly braces.
- Each key is separated from its value with a colon(:)
- **Syntax:** `dict1 = {"Key" : "Value"}`
- Keys should be **unique** and **immutable** type (string, number, tuple) as they are used for indexing while values can be different or same and having same or different data types.
- Example: `dict1 = {"name": "ram", "city": "Nagpur"}`
- Example: `dict2 = {}` → Empty Dictionary
- Example: `dict3 = {"subjects" : ["maths", "physics"], ("code", "semester") : [202, 2]}`

## Dictionary: Operations

→ `dict1 = {"Name": "Zara", "age": 15}`

- Accessing the dictionary:
  - `print(dict1)` → Output: {'Name': 'Zara', 'age': 15}
- Accessing with the keys:
  - `print(dict1['Name'])` → Output: 'Zara'
  - `print(dict1['name'])` → Output: KeyError: 'Name'
  - `print(dict1.get('Name'))` → Output: 'Zara'
  - `print(dict1.get('name'))` → Output: *No\_Output*

## Dictionary: Operations

→ `dict1 = {"Name": "Zara", "age": 15}`

- Updating the dictionary:

`dict1['age'] = 25 → print(dict1) → Output: {'Name': 'Zara', 'age': 25}`

- Updating the dictionary:

`dict1['school'] = 'Tip-Top' → Adds a new entry.`

`print(dict1) → Output: {'Name': 'Zara', 'age': 15, 'school': 'Tip-Top'}`

- Deletion on the dictionary:

`del dict1['school'] → print(dict1) → Output: {'Name': 'Zara', 'age': 25}`

`dict1.clear() → Removes all entries → Output: {}`

`del dict1 → Deletes the dictionary`

## Dictionary: Properties of Keys

- Keys should be unique. No duplicate keys are allowed. If two keys having same name present, the latest one will be in use.
- Keys must be immutable i.e. string, number or tuple. Mutable keys like lists are not allowed.

```
d = {"Name": "Zara", "age": 15, "name": "Sara"}
```

```
print(d) → Output: {'Name': 'Zara', 'age': 15, 'name': 'Sara'}
```

```
d = {"Name": "Zara", "age": 15, "Name": "Sara"}
```

```
print(d) → Output: {'Name': 'Sara', 'age': 15}
```

```
d = {[{"Name": "Zara", "age": 15, "Gender": "F"}]
```

```
print(d) → Output: TypeError: unhashable type: 'list'
```

# Dictionary: Built-in Methods

- Comparison of two dictionaries:
- Length of the dictionary:
- Iteration over the dictionary:

```
d1 = {1: 'A', 2: 'B'}; d2 = {1: 'A', 2: 'B'}
```

```
print(d1==d2) → Output: True
```

```
print(len(d1)) → Output: 2
```

```
d1 = {1: 'A', 2: 'B', 3: 'C'}
```

```
for x in d1:  
    print(x, ":", d1[x])
```

Output:

1 : A

2 : B

3 : C

## Dictionary: Built-in Methods

- **Iteration over the dictionary:** Default Iteration will only retrieve Keys.

```
d1 = {1: 'A', 2: 'B', 3: 'C'}  
for x in d1:  
    print(x)
```

Output:

1  
2  
3

## Dictionary: Built-in Methods

- **Iteration over the dictionary:** The items() function enables to retrieve both keys and values.

```
d1 = {1: 'A', 2: 'B', 3: 'C'}
for x, y in d1.items():
    print(x, y)
```

Output:

```
1 A
2 B
3 C
```

## Dictionary: Built-in Methods

- **Iteration over the dictionary:** The keys() and values() function enables to retrieve only keys and values respectively.

```
d1 = {1: 'A', 2: 'B', 3: 'C'}  
for x in d1.keys():  
    print(x)
```

Output:

1  
2  
3

```
d1 = {1: 'A', 2: 'B', 3: 'C'}  
for x in d1.values():  
    print(x)
```

Output:

A  
B  
C

## Dictionary: Methods

- `dict1 = {"Name": "Zara", "age": 15}`
- `dict.copy()`: Returns a shallow copy of the dictionary.
  - `dict2 = dict1.copy() → dict2`
  - Output: `dict1 = {"Name": "Zara", "age": 15}`
- `dict.clear()`: Removes all items from the dictionary.
  - `dict2.clear() → dict2 → Output: {}`
- `dict.get(key)`: Returns a value for the given key.
  - `dict1.get('age') → Output: 15`
- `dict.items()`: Returns a list of dictionaries key-value tuple pairs.
  - `dict1.items() → Output: dict_items([('Name', 'Zara'), ('age', 15)])`

## Dictionary: Methods

→ `d = {"Name": "Zara", "age": 15, "Gender": "Female"}`

- `dict.pop(key_name)`: Returns and Removes the Key and its Value from a Dictionary.

→ `d.pop('age')` → Output: 15

→ `print(d)` → Output: {'Name': 'Zara', 'Gender': 'Female'}

- `dict.popitem()`: Returns and Removes last item from a dictionary.

→ `d.popitem()` → Output: ('Gender', 'Female')

→ `print(d)` → Output: {'Name': 'Zara'}

## Dictionary: Methods

→ `d = {"Name": "Zara", "age": 15}`

- `dict.keys()`: Returns a list of all keys from the dictionary.

→ `d.keys() → Output: dict_keys(['Name', 'age'])`

- `dict.values()`: Returns a list of all values from the dictionary.

→ `d.values() → Output: dict_values(['Zara', 15])`

- `dict.update()`: Adds another dictionaries key-value pairs in the original one.

→ `d1 = {"Name": "Zara", "age": 15}`

→ `d2 = {'gender': 'f'}`

→ `d1.update(d2)`

→ `print(d1) → Output: {'Name': 'Zara', 'age': 15, 'gender': 'f'}`

## Dictionary: Nesting Dictionary

```
d = {"Zara" : {"Age": 15, "Phone": "111-2222-333"},  
      "Sam" : {"Age": 16, "Phone": "333-5656-444"}}
```

```
for i, j in d.items():  
    print('Name :',i)  
    for a, b in j.items():  
        print(a,':',b)  
    print('*****')
```

**Output:**

Name : Zara  
Age : 15  
Phone : 111-2222-333  
\*\*\*\*\*

Name : Harshita  
Age : 16  
Phone : 333-5656-444  
\*\*\*\*\*

# Quiz

Guess the Output!

```
d = {"Peter":45, "Sam":34}  
print(list(d.keys()))
```

- A. 'Peter', 'Sam'
- B. ['Peter', 'Sam']
- C. ('Peter', 'Sam')
- D. {'Peter':45, 'Sam':34}

# Quiz

Guess the Output!

```
a={1:"A",2:"B",3:"C"}
```

```
a.clear()
```

```
print(a)
```

- A. None
- B. {1: None, 2: None, 3: None}
- C. {}
- D. None of these

# Quiz

Which of the following statement if FALSE:

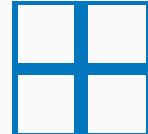
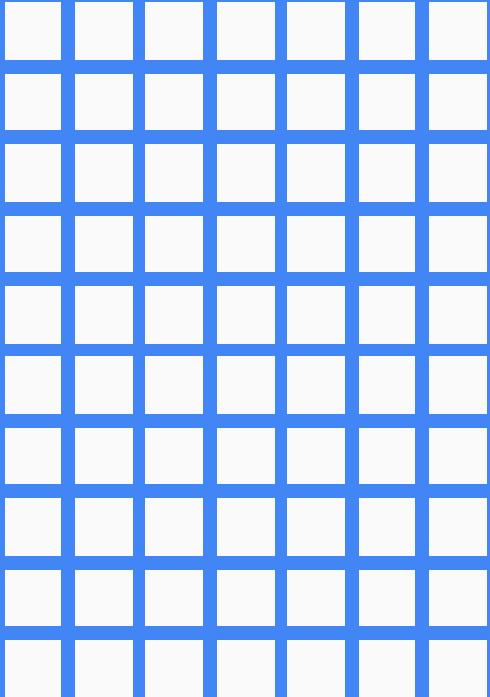
- A. More than One key can have the same value
- B. Values of the dictionary can be accessed as dict[key]
- C. Values of the dictionary must be unique
- D. Values of the dictionary can be a mixture of letters and numbers

# Quiz

Guess the Output!

```
d1 = {'B':23, 'A':34, 'D':23, 'C':56}  
print(sorted(d1))
```

- A. ['B', 'A', 'D', 'C']
- B. ['A', 'B', 'C', 'D']
- C. [23, 23, 34, 56]
- D. Error



**Set**

# Set

- A set object is an unordered collection of distinct hashable objects. It is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference. Sets do not support indexing, slicing, or other sequence-like behavior.
- There are currently two built-in set types: **set** and **frozenset**. The **set** type is **mutable** - the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The **frozenset** type is **immutable** and hashable - its contents cannot be altered after it is created; it can, therefore, be used as a dictionary key or as an element of another set.

- Set creation: Use `set()`
  - `set1 = set([1, 2, 3, 4])` → print(set1) → Output: {1, 2, 3, 4}
  - `set1 = set((1, 2, 3, 4))` → print(set1) → Output: {1, 2, 3, 4}
  - `set1 = set({1, 2, 3, 4})` → print(set1) → Output: {1, 2, 3, 4}
  - `set1 = {1, 2, 3, 4}` → print(set1) → Output: {1, 2, 3, 4}
- Creation of Empty Data Types:
  - Empty List: `l = []`
  - Empty Tuple: `t = ()`
  - Empty Dictionary: `d = {}`
  - Empty Set: `s = set()`

# Set

- The Set Type:

```
s1 = {1, 2, 3, 4}
```

```
s1.add(30)
```

```
print(s1) → Output: {1, 2, 3, 4, 30}
```

- The FrozenSet Type:

```
s2 = frozenset{1, 2, 3, 4}
```

```
s2.add(30) → Output: AttributeError: 'frozenset' object has no attribute 'add'
```

# Set

- `s1 = set([1, 2, 3, 4]) → print(s1) → Output: {1, 2, 3, 4}`
- **Iteration over the set:**
  - `for x in s1: print(x, end=" ") → Output: 1 2 3 4`
- **Add item in set:** Use `add()` for adding one element and `update(sequence)` for adding multiple elements.
  - `s1.add(5) → s1 → Output: {1, 2, 3, 4, 5}`
  - `s1.update([6, 7]) → s1 → Output: {1, 2, 3, 4, 5, 6, 7}`
  - `s1.update((8, 9)) → s1 → Output: {1, 2, 3, 4, 5, 6, 7, 8, 9}`
  - `s1.update('HI') → s1 → Output: {1, 2, 3, 4, 5, 6, 7, 8, 9, 'H', 'I'}`
- **Remove item from set:** `pop()` removes arbitrary/random element, `remove()` removes given element and `discard()` discards given element.
  - `s1.pop() → Output: 1 → s1 → Output: {2, 3, 4, 5, 6, 7, 8, 9, 'H', 'I'}`
  - `s1.remove(4) → s1 → Output: {2, 3, 5, 6, 7, 8, 9, 'H', 'I'}`
  - `s1.discard(9) → s1 → Output: {2, 3, 5, 6, 7, 8, 'H', 'I'}`

- **Intersection of sets:** Use & → A & B is the set that contains all elements of A that also belong to B.
  - `set1 = set([1, 2, 3]); set2 = set([2, 3, 4])`
  - `print(set1 & set2)` → Output: {2, 3}
- **Union of sets:** Use | → A | B is the set of all distinct elements of A and B.
  - `print(set1 | set2)` → Output: {1, 2, 3, 4}
- **Set difference:** Use - → A - B is the set that contains all elements of A which does not contain in B and B - A is the set that contains all elements of B which does not contain in A.
  - `print(set1 - set2)` → Output: {1}
  - `print(set2 - set1)` → Output: {4}

- **Symmetric difference:** Use  $\Delta$  →  $A \Delta B$  is the set that contains all elements which are not common in both the sets A and B
  - `set1 = set([1, 2, 3]); set2 = set([2, 3, 4])`
  - `print(set1 ^ set2)` → Output: {1, 4}
- **issubset and issuperset:**
  - `set1 = set([1, 2, 3]); set2 = set([2, 3, 4]); set3 = set([1, 2, 3, 4, 5])`
  - `print(set1 <= set2)` → Output: False →  $\leq$  : subset
  - `print(set3 >= set1)` → Output: True →  $\geq$  : superset
- **Shallow copy of sets:** Use `set.copy()`
  - `set4 = set1.copy()` → `print(set4)` → Output: {1, 2, 3}
- **Clear the sets:** Use `set.clear()`
  - `set4.clear()` → `print(set4)` → Output: set()

## Set Operations using Methods and Builtins:

- `a = {1, 2, 2, 3, 4}; b = {3, 3, 4, 4, 5}; c = {1, 2}; d = {7, 9}`
- **Intersection:** `a.intersection(b)` → Output: `{3, 4}`
- **Union:** `a.union(b)` → Output: `{1, 2, 3, 4, 5}`
- **Difference:** `a.difference(b)` → Output: `{1, 2}`
- **Difference:** `b.difference(a)` → Output: `{5}`
- **Symmetric Difference:** `a.symmetric_difference(b)` → Output: `{1, 2, 5}`
- **Symmetric Difference:** `b.symmetric_difference(a)` → Output: `{1, 2, 5}`
- **Subset:** `c.issubset(a)` → Output: True
- **Superset:** `b.issuperset(a)` → Output: False
- **Disjoint:** `a.isdisjoint(b)` → Output: False
- **Disjoint:** `c.isdisjoint(d)` → Output: True

# Quiz

Guess the Output!

`s1 = {1, 2, 7, 8}`

`s2 = {5, 6, 1, 7}`

`print(s1 ^ s2)`

- A. `{1, 7}`
- B. `{1, 2, 7, 8, 5, 6}`
- C. `{2, 5, 6, 8}`
- D. `{}`

# Quiz

Guess the Output!

```
s1 = {1, 2, 7, 8}  
s1.remove(7)  
print(s1)
```

- A. 7
- B. {1, 2, 8}
- C. Can't remove the value
- D. Index is not found error

# Quiz

Guess the Output!

```
s1 = {1, 2, 7, 8}
```

```
s2 = {1, 2, 7, 8}
```

```
print(s1 <= s2)
```

```
print(s1 >= s2)
```

- A. True True
- B. True False
- C. False True
- D. False False

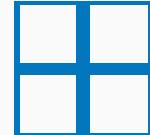
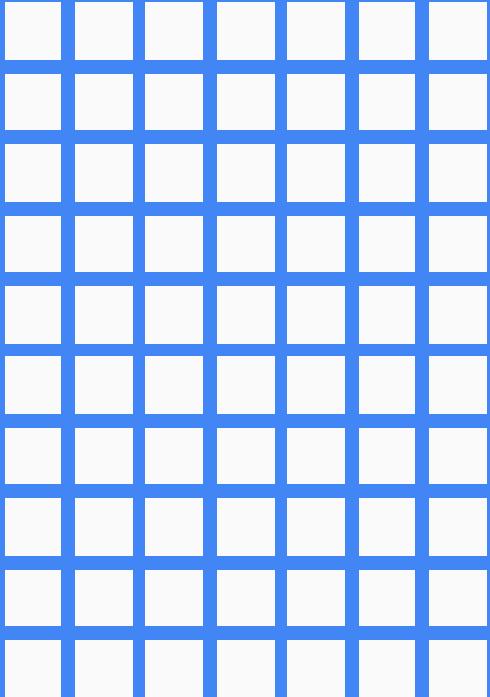
# Quiz

Guess the Output!

```
s1 = {"hello">#ABC!"}
```

```
print('#' in s1)
```

- A. False
- B. True
- C. Invalid Set
- D. None



# Functions

# Functions

- A function is a block of organized, reusable code that is used to perform a single, related action.
- Advantages of using functions:
  - Reducing duplication of code
  - Decomposing complex problems into simple pieces
  - Improving clarity of the code
  - Reuse of the code
  - Information hiding
- Types of functions:
  - 1. Built-in functions: Functions that are predefined into a library.
  - 2. User defined functions: Functions that are created by the programmer to meet the requirements.

## Functions: User defined functions

- User defined function begins with **def** keyword followed by ***function name***. The function may or may not take one or more arguments as input within parentheses. End the function declaration with **colon(:)**. A block of code is written inside the function body which will be an executable statement.

Syntax: Defining the function →

```
def function_name(arguments):
```

```
    function_statement1
```

```
    function_statement2
```

```
return statement
```

→ The **return statement** is optional

## Functions: User defined functions

- Calling/Invoking the function is done using the *function name*, parentheses (opening and closing) and *parameters or arguments* if any.

Syntax: Calling the function →

`variable_name = function_name(arguments)`

`def avg_of_number(x, y):` ← Declaring the function

`print("Average of %d and %d is %d:" %(x, y, (x + y)/2))`

`avg_of_number(4, 5)` ← Calling the function

Output: The average of 4 and 5 is 4.5

## Functions: User defined functions

Example: Without **Return Statement**:

```
def printme(a, b):  
    c = a + b  
    print("Addition is:", c)
```

Calling the function:

```
printme(2, 3)
```

Output: Addition is: 5

## Functions: User defined functions

Example: With **Return Statement**:

```
def printme(a, b):  
  
    c = a + b  
  
    return c
```

Calling the function:

```
c = printme(2, 3)  
  
print("Addition is:", c)
```

Output: Addition is: 5

## Functions: User defined functions

Example: With **Return Statement**: Returning Multiple Values

```
def printme(a, b):
    c = a + b
    d = a - b
    return c, d
c, d = printme(10, 5)
print("Addition is:", c)
print("Subtraction is:", d)
```

Output:

Addition is: 15

Subtraction is: 5

## Functions Arguments

**Required Arguments:** Positional arguments are required to pass in correct order.

Example:

```
def printme(a):
```

```
    print(a)
```

```
printme("Hello") → Output: Hello
```

```
printme() → Output: TypeError: printme() missing 1 required positional argument: 'a'
```

## Functions Arguments

**Keyword Arguments:** Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

Example:

```
def printme(name, age):  
    print("Name:", name, "Age:", age)
```

`printme(name = "Sam", age = 23) → Output: Name: Sam Age: 23`

`printme(age = 25, name = "John") → Output: Name: John Age: 25`

## Functions Arguments

**Default Arguments:** This argument assumes a default value if a value is not provided in the function call for that argument.

Example:

```
def printme(name, age = 25):  
    print("Name:", name, "Age:", age)
```

printme("Sam") → Output: Name: Sam Age: 25

printme(age = 28, name = "John") → Output: Name: John Age: 28

## Functions: Anonymous Function

- These functions are called **anonymous** because these are not declared in the standard manner by using the **def** keyword. We can use the **lambda** keyword to create small anonymous functions.
- Lambda takes any number of arguments, but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.

## Functions: Anonymous Function

Syntax: **lambda [arguments]: expression**

Example:

```
sum = lambda x, y : x + y
```

```
print("The value is:", sum(10, 20)) → Output: The value is 30
```

No need to assign lambda function to a variable, it is optional.

```
(lambda x, y : x + y)(15, 15) → Output: 30
```

## Functions: \*args and \*\*kwargs

- **\*args** allows us to pass variable number of arguments to the function.
- The name of **\*args** is just a convention you can use anything that is a valid identifier.

```
def sum(*args):  
    sum = 0  
    for i in args:  
        sum = sum + i  
    print("Sum is:", sum)
```

sum(1)	→ Output: Sum is: 1
sum(1, 2, 3)	→ Output: Sum is: 6
sum(11, 22, 33, 44)	→ Output: Sum is: 110

## Functions: \*args and \*\*kwargs

- **\*\*kwargs** allows us to pass variable number of keyword arguments.
- The name of **\*kwargs** is just a convention you can use anything that is a valid identifier.

```
def show(**kwargs):  
    for i, j in kwargs.items():  
        print(i, ":", j)
```

```
show(Name = "Sam", Age = 25, City = "Nagpur")
```

Output:

Name : Sam

Age : 25

City : Nagpur

## Functions: \*args and \*\*kwargs

```
def show(**kwargs):
    for i, j in kwargs.items():
        print(i, ":", j)

d = {"Name": "Sam", "Age": 25, "City": "Nagpur"}
show(**d) → We can pass dictionary as **kwargs
```

Output:

Name : Sam  
Age : 25  
City : Nagpur

# Scope of Variables

- The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python
- **Local Variables:** Variables which are defined inside a function body have a local scope. Local variables can be accessed only inside the function in which they are declared.
- **Global Variables:** Variables which are defined outside a function have a global scope. Global variables can be accessed throughout the program body by all functions.

# Scope of Variables

Example:

```
total = 0          → total is a Global Variable  
def sum(a, b):  
    total = a + b  → a and b are Local Variables  
    count = 0       → count is a Local Variable  
    print("Inside a function the total is:", total)
```

```
sum(10, 20)  
print("Outside a function the total is:", total)
```

Output:

```
Inside a function the total is: 30  
Outside a function the total is: 0
```

# Scope of Variables

Example:

```
total = 0          → total is a Global Variable  
def sum(a, b):  
    total = a + b  → a and b are Local Variables  
    count = 0       → count is a Local Variable  
    print("Inside a function the total is:", total)  
  
print("Outside a function the count is:", count)
```

Output:

NameError: name 'count' is not defined

## Scope of Variables

- Nonlocal Variables: Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.
- We use **nonlocal** keyword to create nonlocal variable.

```
def outer():
    x = "local"
    def inner():
        x = "nonlocal"
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
```

Output:

inner: nonlocal  
outer: local

# Global Keyword

- In Python, **global** keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

## Rules of **global** Keyword:

The basic rules for global keyword in Python are:

- When we create a variable inside a function, it's **local** by default.
- When we define a variable outside of a function, it's **global** by default. You don't have to use **global** keyword.
- We use **global** keyword to read and write a **global** variable inside a function.
- Use of **global** keyword outside a function has no effect.

## Global Keyword

Example 1: Accessing global Variable From Inside a Function

```
c = 1  
def add():  
    print("Global Variable c:",c)
```

```
add()
```

Output: Global Variable c: 1

Example 1: Modifying Global Variable From Inside the Function

```
c = 1  
def add():  
    c = c + 2  
    print("Global Variable c:",c)
```

```
add()
```

Output: UnboundLocalError: local variable 'c' referenced before assignment

## Global Keyword

Example 3: Changing Global Variable From Inside a Function using global

```
c = 0
```

```
def add():
    global c
    c = c + 2
    print("Inside add():", c)
```

```
add()
print("In main:", c)
```

Output:

```
Inside add(): 2
In main: 2
```

## Global Keyword

Example 4: Using a Global Variable in Nested Function

```
def foo():
    x = 20
```

```
def bar():
    global x
    x = 25
```

```
print("Before calling bar:", x)
print("Calling bar now")
bar()
print("After calling bar:", x)
```

```
foo()
print("x in main:", x)
```

Output:

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main: 25
```

## Python Built-in Functions

- Python bin(): The bin() method converts and returns the binary equivalent string of a given integer.
- Syntax: bin(num)

Example:

```
number = 5  
print('The Binary equivalent of 5 is:', bin(number))
```

Output:

The Binary equivalent of 5 is: 0b101

## Python Built-in Functions

- Python hex(): The hex() method converts an Integer number to the corresponding hexadecimal string.
- Syntax: hex(num)

Example:

```
number = 435
```

```
print('The Hexadecimal equivalent of 435 is:', hex(number))
```

Output:

The Hexadecimal equivalent of 435 is: 0x1b3

## Python Built-in Functions

- Python oct(): The oct() method takes an integer number and returns its octal representation.
- Syntax: oct(num)

Example:

```
number = 10  
print('The Octal equivalent of 10 is:', oct(number))
```

Output:

The Octal equivalent of 10 is: 0o12

## Python Built-in Functions

- Python divmod(): The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder.
- Syntax: divmod(x, y) → x → Numerator and y → Denominator

Example:

```
print('divmod(8, 3) = ', divmod(8, 3))
print('divmod(5, 5) = ', divmod(5, 5))
print('divmod(7.5, 2.5) = ', divmod(7.5, 2.5))
print('divmod(6.6, 1.23) = ', divmod(6.6, 1.23))
```

Output:

```
divmod(8, 3) = (2, 2)
divmod(5, 5) = (1, 0)
divmod(7.5, 2.5) = (3.0, 0.0)
divmod(2.6, 1.2) = (5.0, 0.4499999999999973)
```

## Python Built-in Functions

- Python sum(): The sum() function adds the items of an iterable and returns the sum.
- Syntax: sum(iterable, start)  
iterable → Sequence(list, tuple)  
start → This value is added to the sum of items of the iterable.

```
numbers = [2.5, 3, 4, -5]
num_Sum_1 = sum(numbers)
print('Without Start:', num_Sum_1)
start = 10
num_Sum_2 = sum(numbers, 10)
print('With Start:', num_Sum_2)
```

Output:

Without Start: 4.5  
With Start: 14.5

## Python Built-in Functions

- Python map(): The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results.
- Syntax: map(function, iterable)

function → The map() passes each item of the iterable to this function.

iterable → Sequence(list, tuple) which is to be mapped.

```
num1 = [4, 5, 6]
num2 = [5, 6, 7]
```

```
result = map(lambda n1, n2: n1+n2, num1, num2)
print(list(result))
```

Output:

[9, 11, 13]

## Python Built-in Functions

- Python map():

```
def calculate_Square(n):
    return n*n

numbers = (1, 2, 3, 4)
result = map(calculate_Square, numbers)
print(result)
```

```
numbers_Square = list(result)
print(numbers_Square)
```

Output:

```
<map object at 0x000002272693C860>
[1, 4, 9, 16]
```

## Python Built-in Functions

- Python filter(): The filter() method constructs an iterator from elements of an iterable for which a function returns true.
- Syntax: filter(function, iterable)

function → The function that tests if elements of an iterable returns true or false. If None, the function defaults to Identity function which returns false if any elements are false.

iterable → The iterable which is to be filtered, could be sets, lists, tuples, or containers of any iterators.

In simple words, the filter() method filters the given iterable with the help of a function that tests each element in the iterable to be true or not.

## Python Built-in Functions

- Python filter():

```
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']
```

```
def filter_Vowels(alphabet):
    vowels = ['a', 'e', 'i', 'o', 'u']
    if(alphabet in vowels):
        return True
    else:
        return False
```

```
filtered_Vowels = filter(filter_Vowels, alphabets)
print('The filtered vowels are:')
for vowel in filtered_Vowels:
    print(vowel)
```

Output:

The filtered vowels are:

a  
e  
i  
o

## Python Built-in Functions

- Python filter():

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print("Less than Zero Items:")
print(less_than_zero)
```

Output:

Less than Zero Items:

[-5, -4, -3, -2, -1]

## Recursive Functions

- Recursive Function: A Recursive Function is a function that calls itself in its definition.

Example:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)
```

factorial(0) → Output: 1

factorial(3) → Output: 6

factorial(5) → Output: 120

## Generators

- Generators: Python generators are a simple way of creating iterators. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).
- It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.
- If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.
- The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

### Differences between Generator function and a Normal function:

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

## Generators

```
def my_gen():
    n = 1
    print('This is printed first')
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

for item in my_gen():
    print(item)
```

Output:

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

## Generators

```
# A simple generator for Fibonacci Numbers
def fib(limit):
```

```
# Initialize first two Fibonacci Numbers
```

```
a, b = 0, 1
```

```
# One by one yield next Fibonacci Number
```

```
while a < limit:
```

```
    yield a
```

```
    a, b = b, a + b
```

```
# Iterating over the generator object using for loop
```

```
print("Fibonacci Series:")
```

```
for i in fib(5):
```

```
    print(i)
```

Output:

Fibonacci Series:

0

1

1

2

3

# Quiz

Guess the Output!

```
def sayhello():
    print('Say Hello')
```

sayhello()

sayhello()

A. Say Hello

Say Hello

A. 'Say Hello'

'Say Hello'

# Quiz

Where can we define a function?

- A. In Module
- B. In Class
- C. In an another function
- D. All of the above
- E. None of the above

# Quiz

What are the two main types of function?

- A. System, Custom
- B. User-Defined, Custom
- C. System, User-Defined
- D. Built-in, User-Defined
- E. Built-in, User

# Quiz

Python supports the creation of anonymous functions at runtime, using construct called

- A. Pi
- B. Lambda
- C. Anonymous
- D. Little
- E. None of the above

# Quiz

Guess the Output!

```
def display(b, n):  
    while n > 0:  
        print(b,end="")  
        n=n-1
```

```
display('z', 3)
```

- A. zzz
- B. zz
- C. Infinite Loop

# Quiz

Guess the Output!

```
def loc():
```

```
    x = 100
```

```
    print(x)
```

```
    x = x + 1
```

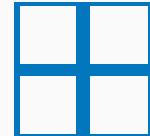
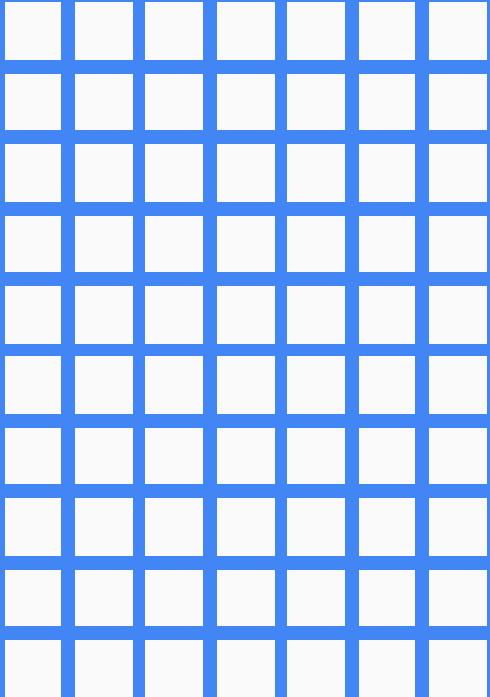
```
loc()
```

- A. 101

- B. 100

- C. 99

- D. Error



# Modules

# Modules

- Module allows us to logically organize our Python code i.e. Modules are used to categorize Python code into smaller parts.
- A module is simply a Python file, where classes, functions and variables are defined. Grouping similar code into a single file makes it easy to access.
- Module Advantage:
  - **Reusability:** Module can be used in other python code providing the reusability of the code.
  - **Categorization:** Similar type of attributes can be placed in the one module provides the better categorization.

## Modules

- Using the ***import*** statement: ***import*** statement can be used to import a module.

Create a file named as ***basic.py***: Write a function inside this file. Here ***basic.py*** is our **module** in which ***add()* & *mul()*** functions are defined.

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

```
    print("Addition of", a, "and", b, "is:", a + b)
```

```
def mul(a, b):
```

```
    print("Multiplication of", a, "and", b, "is:", a * b)
```

## Modules

- Using the ***import*** statement: ***import*** statement can be used to import a module.

Create ***sample.py*** file.

Using ***import*** statement, we will import the module i.e. ***basic.py*** file in our ***sample.py*** file in which we will use ***add()*** function from module basic.

```
import basic
```

```
basic.add(12, 12) → Output: Addition of 12 and 12 is: 24
```

## Modules

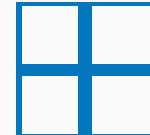
- Using the ***from...import*** statement: ***from...import*** statement can be used to import particular attribute from a module.

Using ***from...import*** statement, we will import only `mul()` function.

```
from basic import mul
```

`mul(12, 12)` → Output: Multiplication of 12 and 12 is: 144

- The ***dir(module\_name)*** function: Returns a list of all attributes defined in a module.
- Example: `print(dir(basic))` → Output: ['add', 'mul']



## Files & I/O

- Input & Output Statements:
- Input: **input()** function is used to take a input from user.

Example:

```
x = int(input("Enter the number: "))  
print("x:", x)
```

Output:

Enter the number: 23

x: 23

- Output: **print()** function is used to display an output.

- Python also provides supports of reading and writing data to Files. Python provides the facility of working on Files. A File is an external storage on hard disk from where data can be stored and retrieved.

### Operations on File:

- **Opening a file:** Before working on a file, we have to first open a file using `open()` function which returns an object of file.
- The `open()` function: `open(file_name, access_mode)`
  - Example: `with open("sample.txt", 'w') as f:  
 f.write("Hello Python")`
- **file\_name:** Name of the file.
- **access\_mode:** This determines the mode in which the file has to be opened: read, write or append mode. Default is **read**.

- **Writing to a file:** `write()` function is used to write a string into a file.
- The `write()` function: `write(string)`
  - Example: `with open("sample.txt", 'w') as f:  
 f.write("Hello Python")` → Output: 12
- **Reading from a file:** `read()` function is used to read data from a file.
- The `read()` function: `read()`
  - Example: `with open("sample.txt", 'r') as f:  
 print(f.read())` → Output: 'Hello Python'
- **Closing a file:** After finishing the work on file, we should close a file using `close()` function.
- The `close()` function: `file_object.close()`
  - Example: `f.close()`

## Files: Access Modes

Mode	Description
r	Opens a file for read only. Default Mode. Pointer is at the beginning of the file.
rb	Opens a binary file for read only. Default Mode. Pointer is at the beginning of the file.
w	Opens a file for write only. Overwrites the file if exists otherwise creates a new file.
wb	Opens a binary file for write only. Overwrites the file if exists otherwise creates a new file.
a	Opens a file for appending. The file pointer is at the end position. If a file does not exists a new file will be created and append the data at the end of file.
ab	Opens a binary file for appending. If a file does not exists a new file will be created and append the data at the end of file.

## Files

```
with open("sample.txt", 'w') as f:  
    f.write("Welcome to the world of Python")
```

```
with open("sample.txt", 'r') as f:  
    print(f.read())
```

```
with open("sample.txt", 'r') as f:  
    print(f.read(20))
```

Output:  
'Welcome to the world of  
Python'

*Reads first 20 bytes from the  
beginning of the file.*

Output:  
'Welcome to the world'

## Files: Attributes of the file

- The file object attributes: Returns various information related to the file.

Example: `with open("sample.txt", 'r') as f:  
 print(f.read())`

- **file\_object.name**: Returns the name of the file.
  - Example: `print(f.name)` → Output: 'sample.txt'
- **file\_object.mode**: Returns the mode in which the file is being opened.
  - Example: `print(f.mode)` → Output: 'r'
- **file\_object.closed**: Returns **True** if the file is closed, **False** otherwise.
  - Example: `print(f.closed)` → Output: True

## Files:

- **File position:** `tell()` method returns the current position within the file i.e. cursor position.
- The `seek()` method: Changes the current position of cursor within the file. `seek(0)` will return the cursor at the beginning of file.

Example: `with open("sample.txt", 'r') as f:`

```
    print(f.read())
    print(f.tell())
    f.seek(0)
    print(f.tell())
```

Output:

'Welcome to the world'

20

0

## OS Module

- Python **os** module provides methods that help us performing file-processing operations.
- **import os**: First we have to import **os** module of python.
- **os.getcwd()**: Displays the current working directory.
- **os.mkdir("directory\_name")**: Creates a new directory.
- **os.chdir("new\_directory\_name")**: Changes the current directory.
- **os.rmdir("directory\_name")**: Removes the directory.
- **os.rename("old\_file\_name", "new\_file\_name")** :Renaming the file name.
- **os.remove("file\_name")**: Deleting the file.

## OS Module

```
import os module
```

← import os module

```
print(os.getcwd())
```

← Output: C:\Users\Ankush

```
os.mkdir("sample")
```

← Makes a new folder named “sample”

```
os.rename("sample.txt", "test.txt")
```

← Renaming the *sample.txt* file as  
*test.txt*

```
os.remove("test.txt")
```

← Removes the *test.txt* file

# Quiz

Which one of the following is not attributes of file?

- A. closed
- B. rename
- C. name
- D. mode

# Quiz

What is the use of tell() method in files?

- A. It tells you the current position within the file
- B. It tells you the end position within the file
- C. It tells you the start position within the file
- D. None of the above

# Quiz

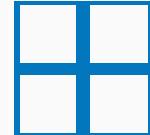
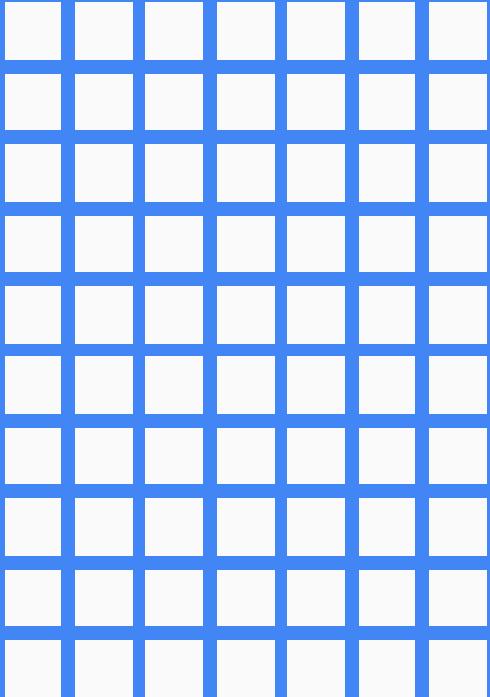
In file handling, what does these terms means 'r', 'a', 'w'?

- A. right, append, wrong
- B. read, append, write
- C. Both A and B
- D. None of the above

# Quiz

What is/are the basic I/O connection/s in file?

- A. Standard Input
- B. Standard Output
- C. Both A and B
- D. None of the above



**Date & Time**

## Date and Time

- Python provides **time** and **calendar** modules to deal with **Date** and **Time**.
- To retrieve current time, first import **time** module, calling predefined function **localtime()** which takes parameters **time.time()** we get time structure. The **time.time()** returns number of **ticks** i.e number in seconds since 12.00 A.M. January 1 1970. These ticks are known as “**epoch**”.

```
import time  
  
lt = time.localtime(time.time())  
  
print("Current Time:", lt)  
  
print("Ticks:", time.time())
```

Current Time:

time.struct\_time(tm\_year=2018,  
tm\_mon=4, tm\_mday=17, tm\_hour=9,  
tm\_min=19, tm\_sec=56, tm\_wday=1,  
tm\_yday=107, tm\_isdst=0)

Ticks: 1523936996.169862

## Date and Time: Time Structure

Attribute	Description	Values
tm_year	4 Digit Current Year	2018
tm_mon	Current Month	1 to 12
tm_mday	Current Day	1 to 31
tm_hour	Current Hour	0 to 23
tm_min	Current Minute	0 to 59
tm_sec	Current Second	0 to 61
tm_wday	Week Day	0 to 6 (0 is MONDAY)
tm_yday	Year Day	1 to 366
tm_isdst	Daylight Savings	-1, 0, 1

## Date and Time

- **Formatted Time:** The function **asctime()** returns a formatted time which includes day, month, date, time and year.

```
import time
```

```
print("Current Time:", time.asctime(time.localtime(time.time())))
```

```
print("Current Time:", time.asctime(time.localtime()))
```

```
print("Current Time:", time.asctime())
```

Output: Same O/P for above three statements.

Current Time:  
Tue Apr 17 09:31:25 2018

## Date and Time

- **Sleep Time:** The function **sleep()** is used to stop the execution of the script for the given interval of time.

```
import time  
  
print("Current Time:", time.asctime())  
  
print("Sleeping...for 15 seconds")  
  
time.sleep(15)  
  
print("Current Time:", time.asctime())
```

Output:

Current Time:  
Tue Apr 17 09:39:00 2018

Sleeping...for 15 seconds

Current Time:  
Tue Apr 17 09:39:15 2018

## Date and Time

- **Calendar:** Calendar module is used to display calendar. The method `calendar.calendar` takes parameter as *theyear* of which the calendar has to be displayed.

```
import calendar  
print("Calendar:", calendar.calendar(2018))
```

## Date and Time

- **Calendar:** The method `calendar.month(theyear, themonth)` provides calendar for the given year and month.

```
import calendar  
print("Calendar:", calendar.month(2018, 6))
```

June 2018						
Mo	Tu	We	Th	Fr	Sa	Su
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

## Date and Time

- **DateTime:** DateTime module deals to manipulate dates or time.

```
import datetime
```

```
today = datetime.date.today()
```

```
print("Today's Date:", today)
```

Output: Today's Date: 2018-08-07

## Date and Time

- **DateTime:** DateTime module deals to manipulate dates or time.

```
import datetime
```

```
today = datetime.date.today()
print("Date Components:")
print("Date:", today.day)
print("Month:", today.month)
print("Year:", today.year)
print("WeekDay:", today.weekday())
```

Output:

```
Date Components:
Date: 7
Month: 8
Year: 2018
WeekDay: 1
```

## Date and Time

- **DateTime:** DateTime module deals to manipulate dates or time.

```
import datetime
```

```
now = datetime.datetime.now()
print("Current Date and Time:",now)
print("In Specified Format:",now.strftime("%y %m %d"))
```

Output:

```
Current Date and Time: 2018-08-07 12:41:39.778254
In Specified Format: 18 08 07
```

## Date and Time

```
import datetime  
now = datetime.datetime.now()  
print("Current Year in Short:", now.strftime("%y"))  
print("Current Year in Full:", now.strftime("%Y"))  
print("Current Month in Numeric:", now.strftime("%m"))  
print("Current Month in Name in Short:", now.strftime("%h"))  
print("Current Month in Name in Full:", now.strftime("%B"))  
print("Current Weekday in Week:", now.strftime("%w"))  
print("Week number of the Year:", now.strftime("%W"))  
print("Day of the Month:", now.strftime("%d"))  
print("Day of the Year:", now.strftime("%j"))  
print("Day of the Week in Short:", now.strftime("%a"))  
print("Day of the Week in Full:", now.strftime("%A"))
```

## Date and Time

### Output:

Current Year in Short: 18

Current Year in Full: 2018

Current Month in Numeric: 08

Current Month in Name in Short: Aug

Current Month in Name in Full: August

Current Weekday in Week: 2

Week number of the Year: 32

Day of the Month: 07

Day of the Year: 219

Day of the Week in Short: Tue

Day of the Week in Full: Tuesday

## Date and Time

```
import datetime  
now = datetime.datetime.now()  
  
print("Current Hour:", now.strftime("%H"))  
print("Current Minute:", now.strftime("%M"))  
print("Current Second:", now.strftime("%S"))  
print("Shows PM or AM:", now.strftime("%p"))  
print("Local Date and Time:", now.strftime("%c"))  
print("Local Date Only:", now.strftime("%x"))  
print("Local Time Only:", now.strftime("%X"))
```

## Date and Time

### Output:

Current Hour: 12

Current Minute: 41

Current Second: 39

Shows PM or AM: PM

Local Date and Time: Tue Aug 7 12:41:39 2018

Local Date Only: 08/07/18

Local Time Only: 12:41:39

## Date and Time

- **DateTime:** timedelta is used to estimate the date and time for both future and past.

```
import datetime  
future = datetime.timedelta(days = 233)  
f = now + future  
print("233 Days after Current Date, new Date will be:", f.date())  
past = datetime.timedelta(days = 118)  
p = now - past  
print("118 Days before Current Date, the Date was:", p.date())
```

Output:

233 Days after Current Date, new Date will be: 2019-03-28  
118 Days before Current Date, the Date was: 2018-04-11

# Quiz

Guess the Output!

```
import time
```

```
time.time()
```

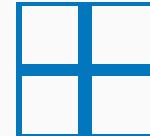
- A. The number of hours passed since 1st January 1970
- B. The number of days passed since 1st January 1970
- C. The number of seconds passed since 1st January 1970
- D. The number of minutes passed since 1st January 1970

# Quiz

Guess the Output!

```
import time  
time.asctime()
```

- A. Current time only
- B. Current date only
- C. Current time and date
- D. Current time, day, month and date



## Exceptions

# Exception Handling

- Whenever an **Exception** occurs the program halts the execution which means further code stops to execute. Exception in a code can be handled. There are many **Standard Errors** predefined in the library which occurs whenever specific error is found in the code.
- On next slide, you will see which standard errors or standard exceptions are predefined in the library.

# Exception Handling: Standard Exceptions

Name	Description
StopIteration	Raised when the next() method of an iterator does not point to any object.
FloatingPointError	Raised when a floating point calculation fails.
ZeroDivisionError	Raised when division by zero takes place for all numeric types.
AttributeError	Raised in case of failure of attribute reference or assignment.
ImportError	Raised when an import statement fails.
IndexError	Raised when an index is not found in a sequence.
KeyError	Raised when the specified key is not found in the dictionary.
NameError	Raised when an identifier is not found in the local or global namespace.

# Exception Handling: Standard Exceptions

Name	Description
UnboundLocalError	Raised when trying to access a local variable in a function or method but no value has been assigned to it..
IOError	Raised for operating system-related errors.
SyntaxError	Raised when there is an error in Python syntax.
TypeError	Raised when an operation or function is attempted that is invalid for the specified data type.
ValueError	Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
RuntimeError	Raised when a generated error does not fall into any category.

# Exception Handling

- **Exception Handling:** The suspicious code may result into raising an exception. Enclose such a code inside the **try** block. The **try** block is followed by **except** statement. If exceptions occurs while executing the code inside the **try** block, exceptions will be handled by **except** statements. There can be multiple **except** statements for the multiple exceptions. The **except** statements are followed by **else-block** which is good place when there occurs no exceptions, code inside this block will execute.
- **Finally Block:** The code inside the **finally** block will always be executed irrespective of exception.

# Exception Handling

```
try:  
    a = 10/0  
    print(a)  
    d = {"name":"sara"}  
    print(d["Name"])  
  
except ZeroDivisionError :  
    print("Divide by zero is not possible")  
  
except KeyError :  
    print("Key is not found in the dictionary")  
  
else:  
    print("Hurray! No Exceptions")  
  
finally:  
    print("This is final block")
```

Output:

Divide by zero is not possible  
This is final block

# Quiz

When will the else part of try-except-else be executed?

- A. When an exception occurs
- B. When no exception occurs
- C. Always
- D. Never

# Quiz

Which of the following is not an exception handling keyword?

- A. try
- B. except
- C. accept
- D. finally

# Quiz

Which error will generate by following code snippet?

```
I = [1, 2, 3]
```

```
I[5]
```

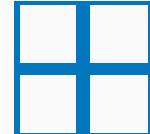
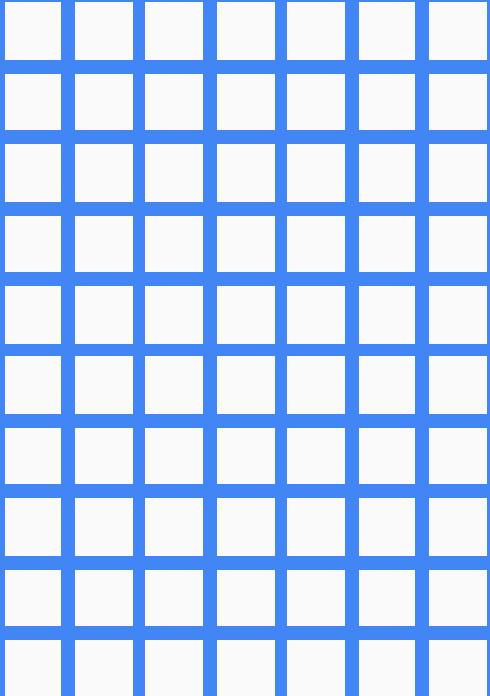
- A. NameError
- B. ValueError
- C. IndexError
- D. TypeError

# Quiz

Which error will generate by following code snippet?

```
a = 4 + '5'
```

- A. NameError
- B. ValueError
- C. IndexError
- D. TypeError



Object Oriented

# Object Oriented Programming

- The basic idea behind an object-oriented programming (OOP) is to combine both data and associated procedures (known as methods) into a single ***unit*** which operate on the data. Such a ***unit*** is called an class. Python is an object-oriented language, everything in Python is an **object**.
- **OOP Terminologies:**
- **Class:** A user-defined prototype or structure for an object that defines a set of attributes that characterize any object of the class. The attributes are **data members** (class variables and instance variables) and **methods**, accessed via **dot** notation.
- **Class Variables:** Variables which defined within a class but outside any of the class's methods and have access to all instances of the class.

# Object Oriented Programming

- **Data Member:** A class variable or instance variable which holds data.
- **Instance Variables:** Variables which defined inside a method and can be accessed only within a method.
- **Instance:** An individual object of a certain class.
- **Object:** An unique instance of the structure of the class which comprises both data members and methods.
- **Inheritance:** The transfer of the characteristics of a class to other classes that derived from it. Such a class which transfers its characteristics is called as **Parent** or **Super** Class and classes which are derived from parent class are called as **Child** or **Sub** Classes.

# Object Oriented Programming

- **Creating Class:** Use **class** statement followed ***class\_name*** followed by indentation i.e. **colon(:)** will create a class of given name. The ***class\_name*** should be in Title Cased.

```
class Employee:  
    print("This is the Employee Class")
```

- **Creating Object/Instance of Class:** An Object of a class have all access of data members and methods of a class. To create an object:

***object\_name= class\_name()*** ← Each Object should be unique.

```
emp1 = Employee()  
print(emp1)
```

**Output:**  
This is the Employee Class  
<\_\_main\_\_.Employee object at 0x036EC490>

# Object Oriented Programming

- **Constructors:** A constructor is a special type of method (function) which is used to initialize the instance members of the class. Constructor can be parameterized and non-parameterized as well. Constructor definition executes when we create object of the class.
- A constructor is a class function that begins with double underscore (\_). The name of the constructor is always the same `__init__()`
- While creating an object, a constructor can accept arguments if necessary. When we create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.
- Every class must have a constructor, even if it simply relies on the default constructor.

# Object Oriented Programming

```
class Employee:  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
  
    def display(self):  
        print("Name:", self.name)  
        print("Salary:", self.salary)  
  
emp1 = Employee("Sara", 25000)  
emp2 = Employee("Smith", 30000)  
emp1.display()  
emp2.display()
```

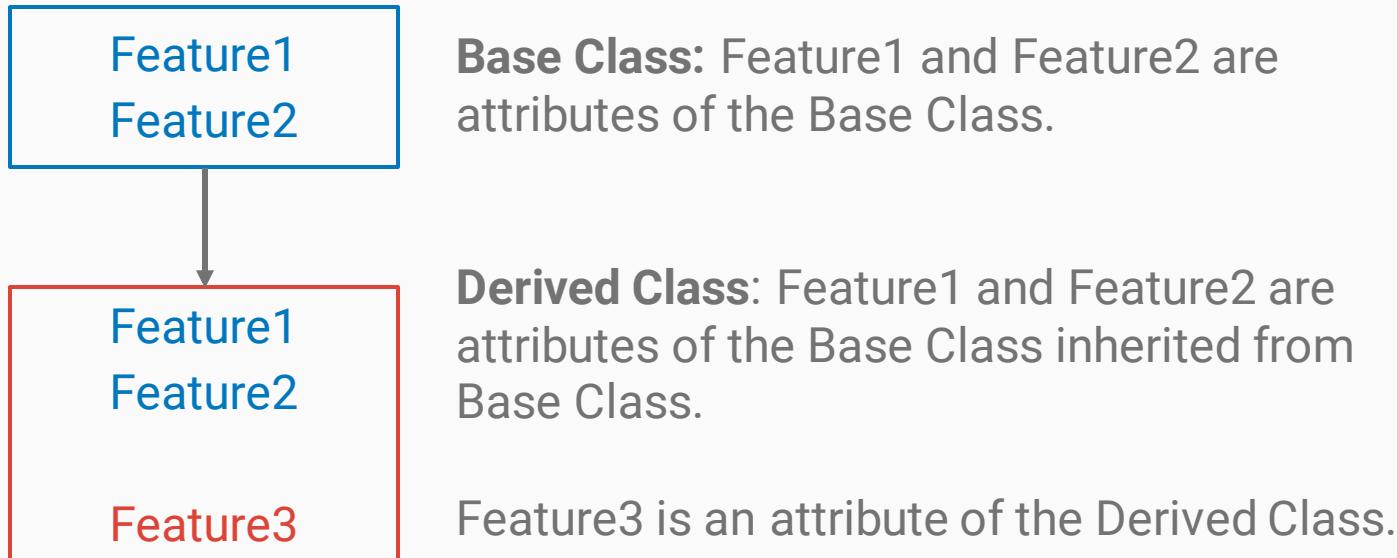
← `__init__()` method is a parameterized constructor taking two parameters name and salary which initialize instance members of the class.

Output:  
Name: Sara  
Salary: 25000  
Name: Smith  
Salary: 30000

← class methods can be accessed by object of the class: `object_name.method_name()`

# Object Oriented Programming

- Simple Inheritance



# Object Oriented Programming

Inheritance Syntax:

```
class BaseClassName:  
    # this_is_base_class
```

```
class DerivedClassName1(BaseClassName):  
    # this_is_derived_class1
```

```
class DerivedClassName2(DerivedClassName1):  
    # this_is_derived_class2
```

# Object Oriented Programming

```
class Rectangle:
```

```
    def __init__(self, length, breadth):
```

```
        self.length = length
```

```
        self.breadth = breadth
```

```
class Area(Rectangle):
```

```
    def __init__(self, length, breadth):
```

```
        Rectangle.__init__(self, length, breadth)
```

```
    def calarea(self):
```

```
        print("Area:", self.length * self.breadth)
```

```
obj1 = Area(12, 12)
```

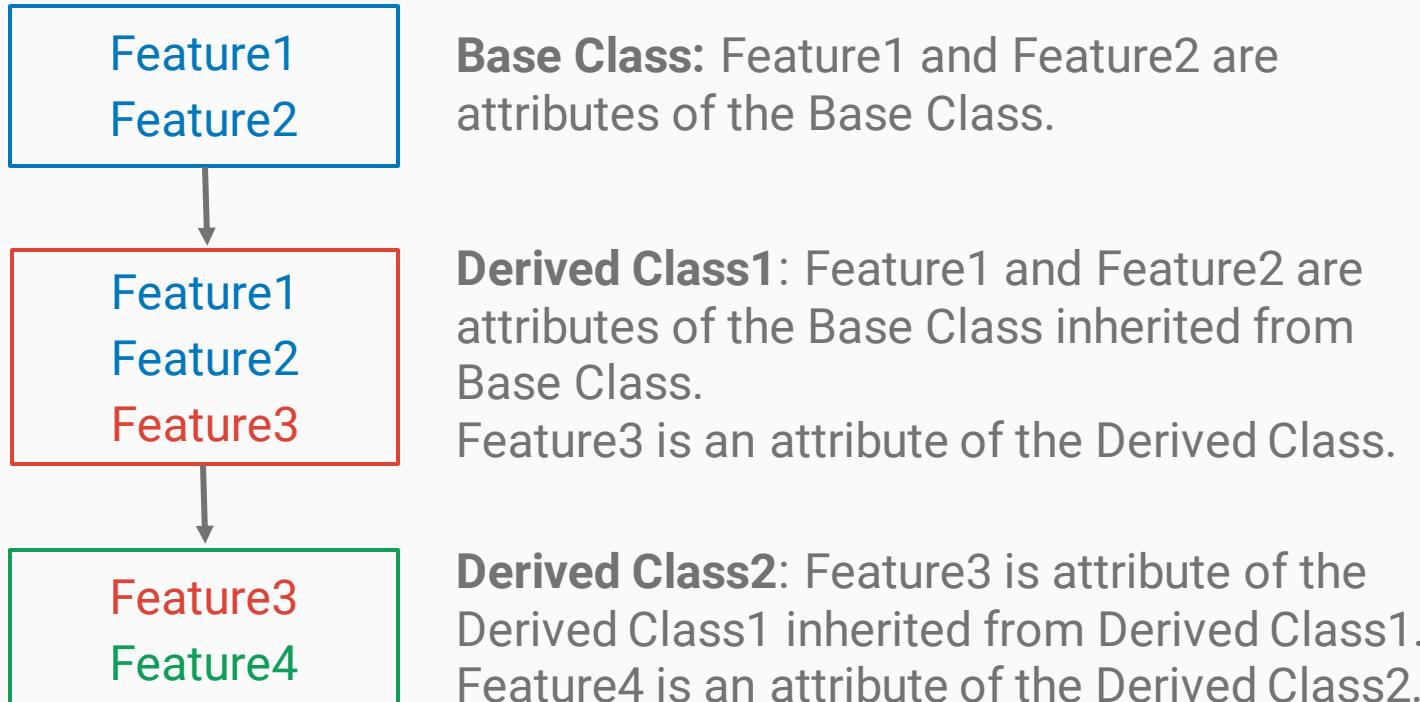
```
obj1.calarea()
```

Output:

Area: 144

# Object Oriented Programming

- Inheritance: Multilevel Inheritance



# Object Oriented Programming

```
class Animal:  
    def type(self):  
        print("Vegetarian")
```

```
class Pet(Animal):  
    def eat(self):  
        print("Eating Grass")
```

```
class Cow(Pet):  
    def feature(self):  
        print("Farming")
```

```
obj1 = Cow()  
obj1.feature()  
obj1.eat()  
obj1.type()
```

Output:  
Farming  
Eating Grass  
Vegetarian

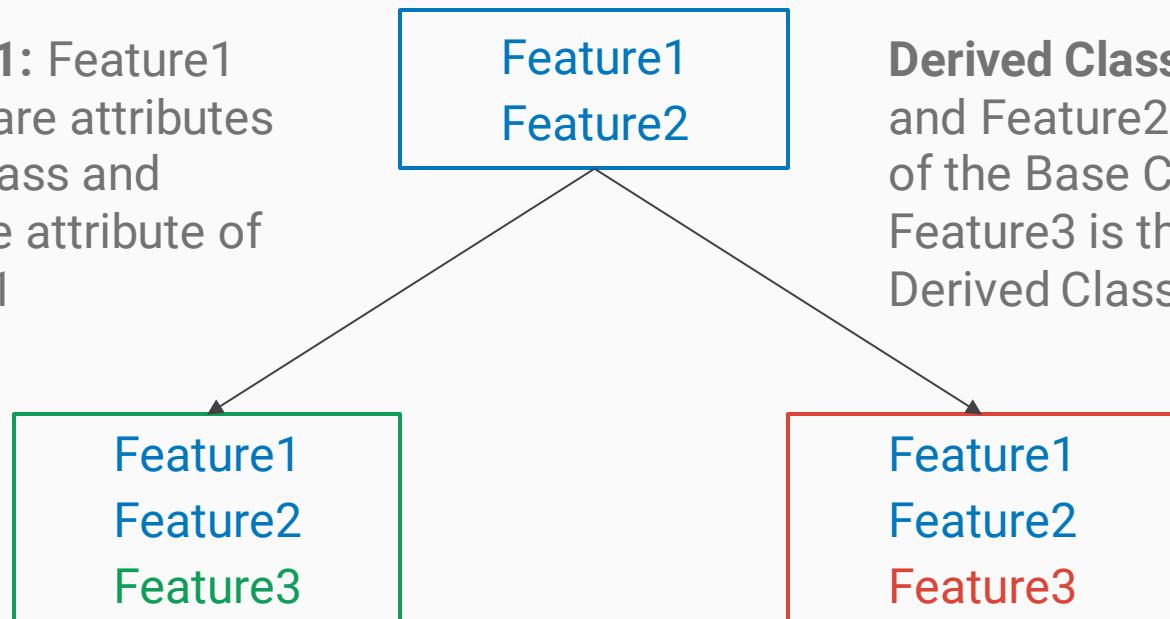
# Object Oriented Programming

- Inheritance: Hierarchical Inheritance

**Base Class:** Feature1 and Feature2 are attributes of the Base Class.

**Derived Class1:** Feature1 and Feature2 are attributes of the Base Class and Feature3 is the attribute of Derived Class1

**Derived Class2:** Feature1 and Feature2 are attributes of the Base Class and Feature3 is the attribute of Derived Class2



# Object Oriented Programming

```
class A:  
    def type(self):  
        print("Type of A called")
```

```
class B(A):  
    def type1(self):  
        print("Type of B called")
```

```
class C(A):  
    def type2(self):  
        print("Type of C called")
```

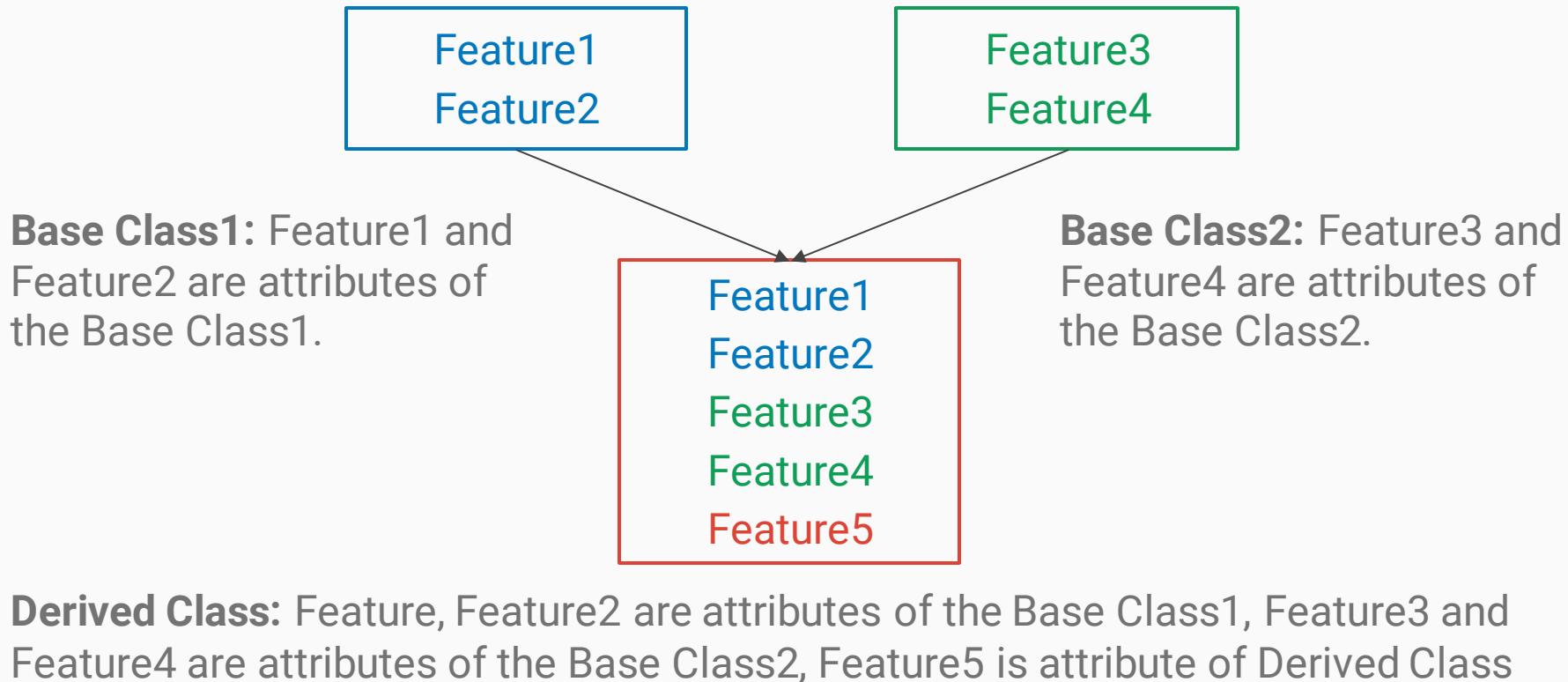
```
obj1 = B()  
obj1.type1()  
obj2 = C()  
obj2.type2()
```

Output:  
Type of B called

Type of C called

# Object Oriented Programming

- Inheritance: Multiple Inheritance



# Object Oriented Programming

```
class Vegetarian:  
    def type(self):  
        print("Vegetarian")  
  
class Pet:  
    def eat(self):  
        print("Eating Grass")  
  
class Cow(Vegetarian, Pet):  
    def feature(self):  
        print("Farming")  
  
obj1 = Cow()  
obj1.feature()  
obj1.eat()  
obj1.type()
```

Output:  
Farming  
Eating Grass  
Vegetarian

# Object Oriented Programming

- **Overriding Methods:** To override the parent method in the child class to perform different functionality but having same name is called as Overriding the method. This is done through **Inheritance** only.

```
class Parent:  
    def display(self):  
        print("Calling Parent Method")
```

```
class Child(Parent):  
    def display(self):  
        print("Calling Child Method")  
        print("Welcome to the Child")
```

```
obj1 = Child()  
obj1.display()  
obj2 = Parent()  
obj2.display()
```

**Output:**  
Calling Child Method  
Welcome to the Child  
  
Calling Parent Method

# Object Oriented Programming

- **Data Hiding:** If we want some attributes of the class which should not be directly visible to outsiders, we make such attributes as **private**. To make private variables, use **\_\_variablename**

```
class JustCounter:  
    __secretcounter = 0  
    def display(self):  
        self.__secretcounter += 1  
        print("Counter:", self.__secretcounter)
```

```
obj1 = JustCounter()  
obj1.display()  
obj2 = JustCounter()  
obj2.display()  
print(obj1.__secretcounter)
```

Output:

Counter: 1

Counter: 2

AttributeError: 'JustCounter' object  
has no attribute '\_\_secretcounter'

# Quiz

\_\_\_\_\_ is used to create an object.

- A. A Class
- B. A Constructor
- C. An Object
- D. A method

# Quiz

What are the methods which begin and end with two underscore characters called?

- A. In-built methods
- B. Special methods
- C. Additional methods
- D. User-defined methods

# Quiz

Suppose B is the subclass of A, to invoke  
`__init__` method in A from B, use:

- A. A.`__init__(self)`
- B. B.`__init__(self)`
- C. A.`__init__(self, B)`
- D. B.`__init__(self, A)`

# Quiz

Guess the type of Inheritance:

class A:

pass

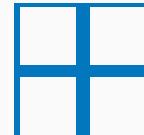
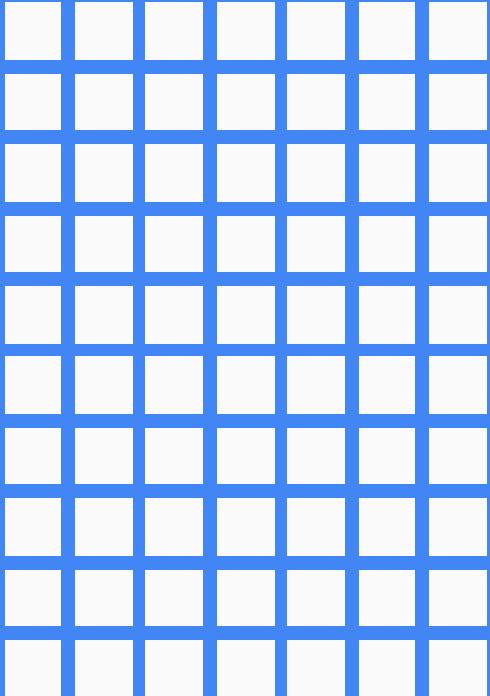
class B(A):

pass

class C(B):

pass

- A. Multiple
- B. Multi-level
- C. Hierarchical



# Regular Expressions

# Regular Expressions

- **Regular Expressions: What is Regular Expression?**
- A regular expression in a programming language is a special text string used for describing a search pattern. It is extremely useful for extracting information from text. Simply put, regular expression is a sequence of characters mainly used to find and replace patterns in a string or file.
- In Python, we have **re** module that helps with regular expressions. So, first we have to **import re**
- Uses of Regular Expressions:
  - Search a string (search and match)
  - Finding a string (findall)
  - Break string into a sub strings (split)
  - Replace part of a string (substitute)

# Regular Expressions

- **Regular Expressions: Various methods of RegEx**
- The ‘re’ package provides multiple methods to perform queries on an input string.

```
import re
○ re.match()
○ re.search()
○ re.findall()
○ re.split()
○ re.sub()
○ re.compile()
```

# Regular Expressions

- The ***match*** function: This method finds match if it occurs at start of the string. This function returns a match object on **Success** and **None** on failure. To display the matching string, ***group()*** method is used which helps to return the matching string. This function is used to find matches at the **beginning** of the string only.
  - Syntax: `re.match("pattern", "string")`

```
import re
str1 = "Welcome, to Python World!"
mobj = re.match(r"Welcome", str1)
print(mobj)
print(mobj.group())
```

Output:  
`<_sre.SRE_Match object; span=(0, 7),  
match='Welcome'>`  
`Welcome`

# Regular Expressions

- The **search** function: This method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern. This function returns a match object on **Success** and **None** on failure. To display the matching string, **group()** method is used which helps to return the matching string.
  - Syntax: `re.search("pattern", "string")`

```
import re
str1 = "Welcome, to Python World!"
mobj = re.search(r"Python", str1)
print(mobj)
print(mobj.group())
```

Output:  
`<_sre.SRE_Match object; span=(12, 18), match='Python'>`  
`Python`

# Regular Expressions

- The ***findall*** function: This method is used to get a list of all matching patterns. It has no constraints of searching from start or end. It can work like `re.search()` and `re.match()` both.
  - Syntax: `re.findall("pattern", "string")`

```
import re
str1 = "India is the incredible, India
is the largest democratic country."
mobj = re.findall(r"India", str1)
print(mobj)
```

Output:  
['India', 'India']

# Regular Expressions

- The **split** function: This method is used to split **string** by the occurrences of given **pattern**. Method **split()** has another argument “**maxsplit**”. It has default value of **zero**. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string.
  - Syntax: `re.split("pattern", "string" [, maxsplit = 0])`

```
import re
str1 = "India-is-the-incredible!"
mobj = re.split(r'-', str1)
print(mobj)
mobj = re.split(r'-', str1, maxsplit = 2)
print(mobj)
```

Output:  
['India', 'is', 'the', 'incredible!']  
['India', 'is', 'the-incredible']

# Regular Expressions

- The **sub** function: This method is used to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.
  - Syntax: `re.sub("pattern", "replacewith", "string")`

```
import re  
str1 = "India-is-the-incredible!"  
mobj = re.sub(r'-', '', str1)  
print(mobj)
```

Output:  
India is the incredible

# Regular Expressions

- The **compile** function: This method is used combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.
  - Syntax: `re.compile("pattern")`

```
import re
str1 = "india is the incredible and
largest democratic country."
pattern = re.compile("india")
mobj = pattern.sub(r"India", str1)
print(mobj)
```

Output:  
India is the incredible and largest  
democratic country.

# Regular Expressions: Commonly used operators

Operator	Description
.	Matches with any single character except newline '\n'.
?	Matches 0 or 1 occurrence of the pattern to its left.
+	Matches 1 or more occurrences of the pattern to its left.
*	Matches 0 or more occurrences of the pattern to its left.
\w	Matches with an alphanumeric character.
\W	Matches with non alphanumeric character.
\d	Matches with digits [0-9]
\D	Matches with non-digits.

# Regular Expressions: Commonly used operators

Operator	Description
\s	Matches with a single whitespace character. (space, newline, return, tab, form)
\S	Matches any non-whitespace character.
\b	Boundary between word and non-word.
\B	Doesn't boundary between word and non-word.
[...]	Matches any single character in a square bracket.
[^...]	Matches any single character not in square bracket.
\	It is used for special meaning characters like \. to match a period or \+ for plus sign.
^ and \$	^ and \$ match the start or end of the string respectively.

# Regular Expressions: Commonly used operators

Operator	Description
{n, m}	Matches at least n and at most m occurrences of preceding expression.
{n}	Matches exactly n number of occurrences of preceding expression.
{n,}	Matches n or more occurrences of preceding expression.
a   b	Matches either a or b.
()	Groups regular expressions and returns matched text.
\t \n \r	Matches tab, newline, return

## Regular Expressions: User Case Data

Extract user's Name, Age, Mobile and Email:

```
data = ""
```

Saurabh is 23 and his mobile: 111-1111-111 and email is saurabh@gmail.com

Piyush is 26 and his mobile: 111-2222-111 and email is p.nagpur@gmail.com

Mayank is 44 and his mobile: 111-3333-444 and email is m.mayank@yahoo.com

Seeta is 18 and her mobile: 111-4444-111 and email is seet-234@hotmail.com

Swapnil is 25 and her mobile: 111-8888-555 and email is swapy25@gmail.com

Amar is 39 and her mobile: 111-9999-111 and email is amar\_rao@yahoo.com

Prabha is 19 and her mobile: 111-7777-111 and email is prabha\_23@gmail.com

Ganesh is 31 and his mobile: 111-5555-666 and email is g.ganesh1@gmail.com

""

# Regular Expressions: User Case Data

Extract user's Name, Age, Mobile and Email:

Observations: Which patterns we can make?

1. User's **Name** is starting with **capital letter**.
2. User's **Age** is **two digits** number.
3. User's **Mobile** contains **all digits** and have specific pattern like **xxx-xxxx-xxx** where **x** are digits from **0 to 9**.
4. User's **Email** contains **alphabets, digits, special symbols** followed by mandatory **@** symbol followed by **email providers** name which contains only **alphabets** followed by **dot(.)** symbol followed by **3** letters alphabets string indicating **domain name**.

## Regular Expressions: User Case Data

```
import re
name = re.findall(r'[A-Z][a-z]*', data)

age = re.findall(r'\s[0-9]{2}\s', data)

mobile = re.findall(r'\d{3}-\d{4}-\d{3}', data)

email = re.findall(r'[A-Za-z0-9_.-]+@\w+\.\w+', data)

user = zip(name, age, mobile, email)
for i in user:
    print(i)
```

## Regular Expressions: User Case Data

### Output:

```
('Saurabh', ' 23 ', '111-1111-111', 'saurabh@gmail.com')
('Piyush', ' 26 ', '111-2222-111', 'p.nagpur@gmail.com')
('Mayank', ' 44 ', '111-3333-444', 'm.mayank@yahoo.com')
('Seeta', ' 18 ', '111-4444-111', 'seet-234@hotmail.com')
('Swapnil', ' 25 ', '111-8888-555', 'swapy25@gmail.com')
('Amar', ' 39 ', '111-9999-111', 'amar_rao@yahoo.com')
('Prabha', ' 19 ', '111-7777-111', 'prabha_23@gmail.com')
('Ganesh', ' 31 ', '111-5555-666', 'g.ganesh1@gmail.com')
```

# Quiz

Guess the Output!

```
import re  
re.sub('evening', 'morning', 'good  
evening')
```

- A. 'good evening'
- B. 'good morning'
- C. 'morning'
- D. 'evening'

# Quiz

Guess the Output!

```
import re  
re.findall('good', 'good is good')
```

- A. 'good'
- B. 'good', 'good'
- C. ('good', 'good')
- D. ['good', 'good']

# Quiz

What does `re.match` do?

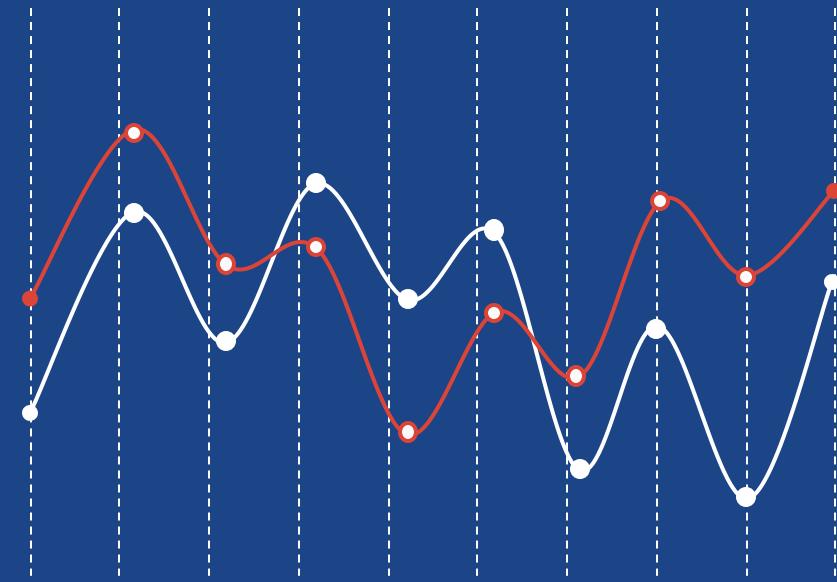
- A. Matches a pattern at the start of the string
- B. Matches a pattern at the any position in the string
- C. Matches all occurrences of the pattern
- D. None of the above

# Quiz

What does `re.search` do?

- A. Matches a pattern at the start of the string
- B. Matches a pattern at the any position in the string
- C. Matches all occurrences of the pattern
- D. None of the above

# Essential #10 Tips



## Tip #1: In-Place Swapping of Two Numbers

```
x, y = 10, 20  
print("Original:",x, y)  
x, y = y, x  
print("After Swapping:",x, y)
```

Output:

Original: 10 20  
After Swapping: 20 10

## Tip #2: Chaining of Comparison Operators

```
n = 10  
result = 1 < n < 20  
print(result)  
result = 1 > n <= 9  
print(result)
```

Output:

True

False

## Tip #3: Work with Multi-Line Strings

```
multiStr = "This Block contains \  
Multiple Lines of Lines."  
print(multiStr)
```

# Using String Continuation  
This Block contains Multiple Lines of Lines.

```
multiStr = ""This Block contains  
Multiple Lines of Lines.""  
print(multiStr)
```

# Using Triple Quotes  
This Block contains  
Multiple Lines of Lines.

```
multiStr = ("This Block contains "  
"Multiple Lines of Lines.")  
print(multiStr)
```

# Using Parentheses by Splitting String  
This Block contains Multiple Lines of Lines.

#### Tip #4: Storing Elements of a List into New Variables

```
testList = [1, 2, 3]
```

```
x, y, z = testList
```

```
print(x, y, z)
```

Output:

```
1 2 3
```

## Tip #5: Dictionary / Set Comprehensions and Create a Dictionary from Two Related Sequences

```
testDict = {i: i * i for i in range(10)}  
testSet = {i * 2 for i in range(5)}  
print(testSet)  
print(testDict)
```

```
t1 = (1, 2, 3)  
t2 = (10, 20, 30)  
print(dict(zip(t1,t2)))
```

Output:

```
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}  
{1: 10, 2: 20, 3: 30}
```

## Tip #6: Simplify if Statement

To verify multiple values:

Use ⇒

`m = 9`

`if m in [1, 3, 5, 7]:`

`pass`

Instead of ⇒

`if m==1 or m==3 or m==5 or m==7:`

`pass`

## Tip #7: Combining Multiple Strings

```
test = ['I', 'Like', 'Python', 'Automation']
print("."join(test))
```

Output:

ILikePythonAutomation

## Tip #8: Reverse the String / List

```
testList = [1, 3, 5]  
testList.reverse()  
print(testList)
```

```
for element in reversed([1, 3, 5]):  
    print(element, end = " ")
```

```
print("Test Python"[:: -1])
```

```
[1, 3, 5][:: -1]
```

*# Reverse the List Itself*  
Output: [5, 3, 1]

*# Reverse the List using for Loop*  
Output: 5 3 1

*# Reverse the String in Line*  
Output: nohtyP tseT

*# Reverse the List using Slicing*  
Output: [5, 3, 1]

## Tip #9: Return Multiple Values from Function

```
def x():
    return 1, 2, 3, 4
```

```
a, b, c, d = x()
```

```
print(a, b, c, d)
```

Output:

```
1 2 3 4
```

## Tip #10: Find the Most Frequent Value in a List

```
test = [1,2,3,4,2,2,3,1,4,4,4]
```

```
print(max(set(test), key=test.count))
```

Output:

4

End

# Thanks!

## Signitive Technologies

Sneh Nagar, Behind ICICI Bank,  
Chhatrapati Square, Nagpur 15

Landmark:  
Bharat Petrol Pump  
Chhatrapati Square

Contact: 9011033776  
[www.signitivetech.com](http://www.signitivetech.com)



“Keep Learning, Happy Learning”

---

# Best Luck!

Have a Happy Future

