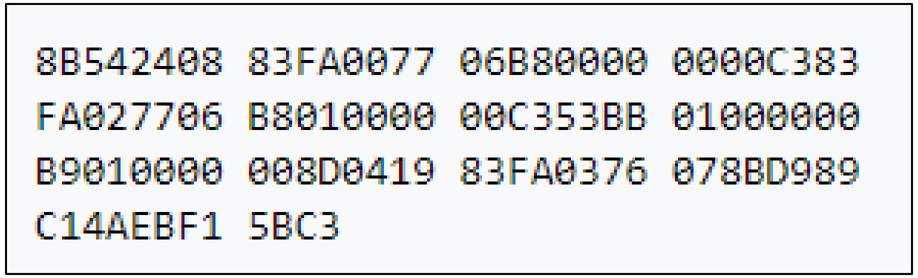
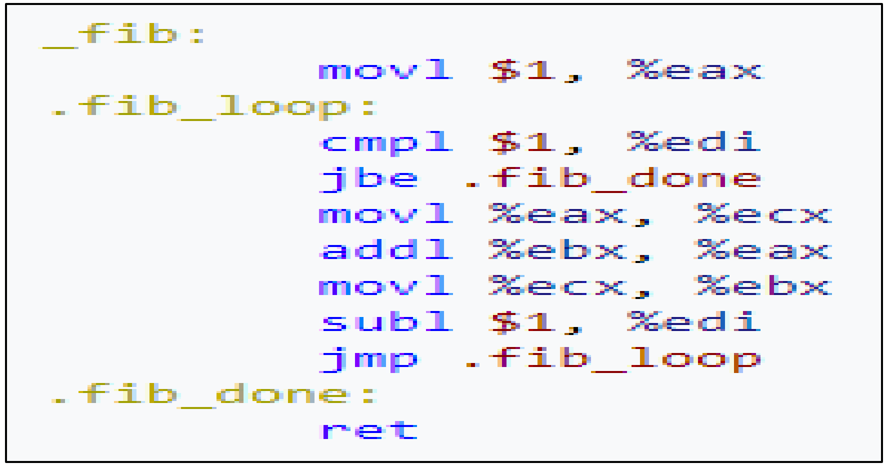
Object Oriented Programming

Recompiled –
Prof. Naiswita Parmar
Assistant Professor,
CSE, SET,
Navrachana University

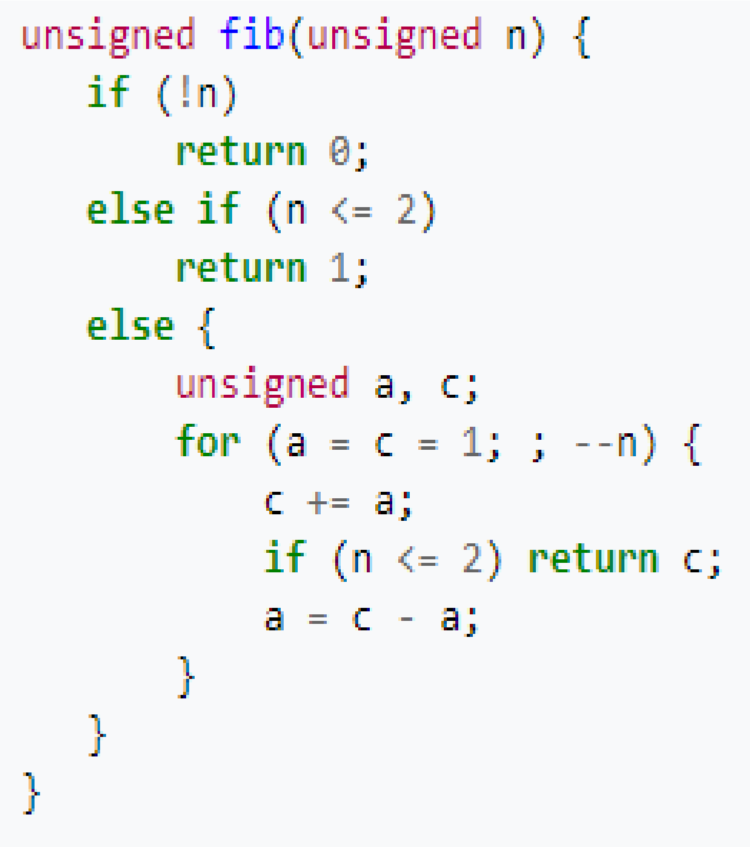
Programming Languages



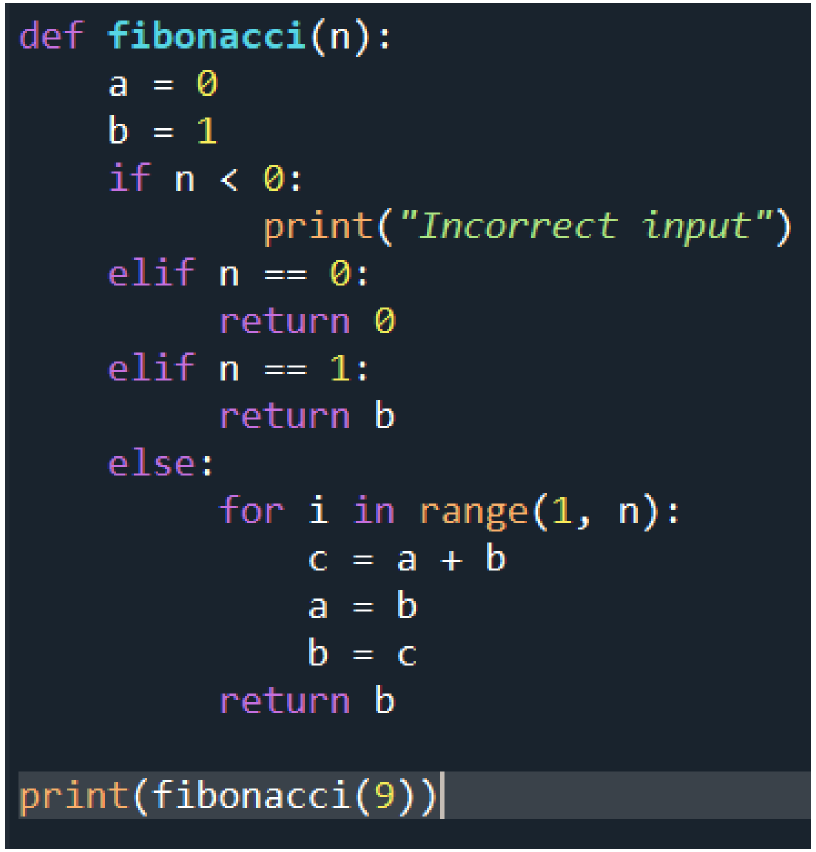
```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD989
C14AEBF1 5BC3
```



```
_fib:
        movl $1, %eax
.fib_loop:
        cmpl $1, %edi
        jbe  .fib_done
        movl %eax, %ecx
        addl %ebx, %eax
        movl %ecx, %ebx
        subl $1, %edi
        jmp  .fib_loop
.fib_done:
        ret
```



```
unsigned fib(unsigned n) {
    if (!n)
        return 0;
    else if (n <= 2)
        return 1;
    else {
        unsigned a, c;
        for (a = c = 1; ; --n) {
            c += a;
            if (n <= 2) return c;
            a = c - a;
        }
    }
}
```



```
def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return 0
    elif n == 1:
        return b
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b

print(fibonacci(9))
```

Programming Languages

- Programming Languages based on Abstraction provided
  - Lower Level Languages
  - Higher Level Language

Programming Languages

How Higher-Level Program Executes

- Preprocessor
- Converters (Compiler / Interpreter)
- Linkers/Loaders (In OS)

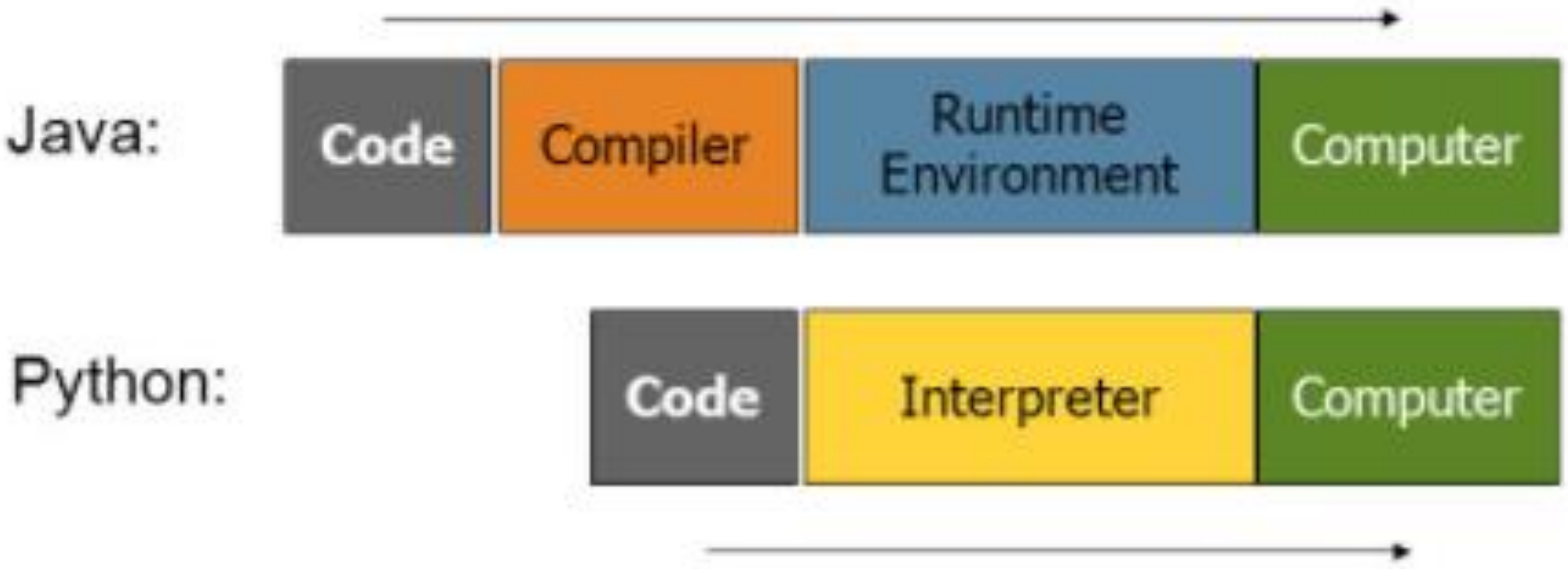
Programming Languages

Compiled	Interpreted
• Compiler takes entire program as input and converts to machine code	• Takes line by line as input and convert to equivalent machine instructions
• Source code converted to machine language form BEFORE execution	• Conversion happens at runtime
• Machine specific executable file (.exe) file is generated	• No .exe file
• Source code not needed every time to run the program	• Source code needed each time to run
• Generally, compiled programs are efficient and execute faster	• More overhead at runtime (syntax checking, linking as well as translation to machine code)
• C, C++, Java, C#, Fortran, COBOL	• Python, Java (Compiled and Interpreted), JavaScript, Perl, PHP

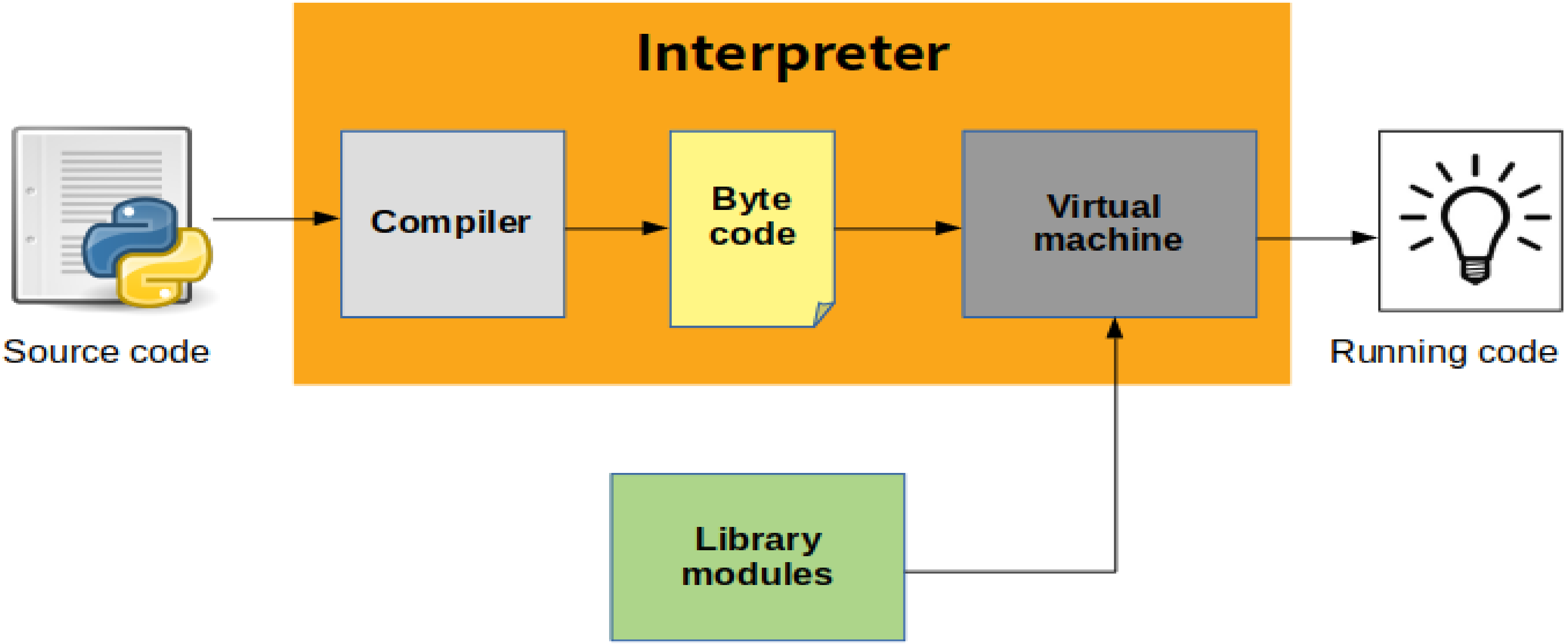
How Program Executes

- C/C++
- Java
- C#
- Python

How Program Executes



How Program Executes



Programming Paradigms

- It is a style, or “way,” of program design.
- It is an approach to solve problem using some programming language.
- Different programming languages follow different approach/style of program design and development
- Some popular and important paradigms are:
  - Structured Programming
  - Procedural Programming / Modular programming
  - Object Oriented Programming

Structured Programming

Structured programming is a programming paradigm aimed at improving

- the clarity,
- quality, and
- development time of a computer program

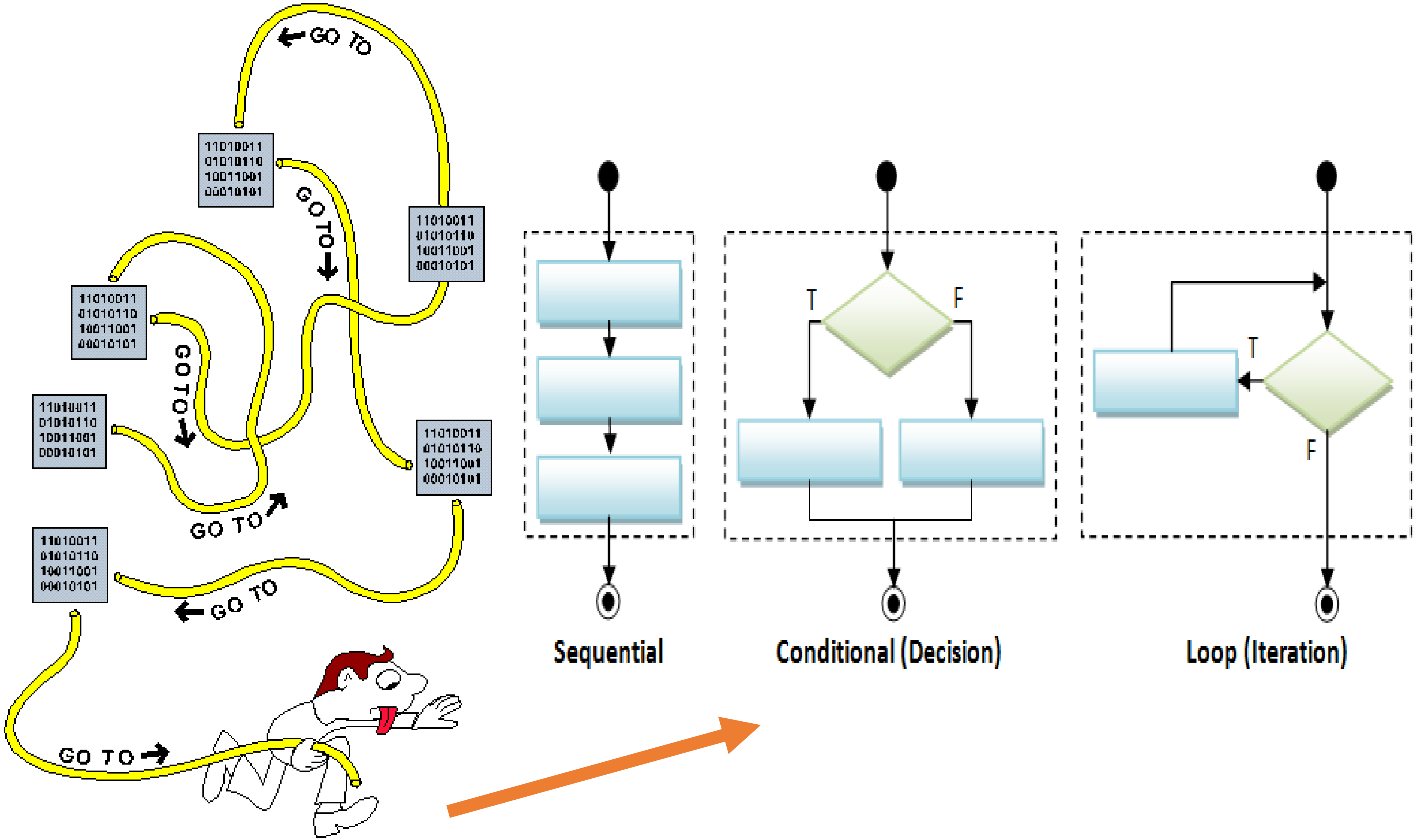
by making **extensive use of the structured control flow** constructs of selection (**if/then/else**) and repetition (**while and for**), block structures, and subroutines.

Structured Programming

- ❑ Program control flow is defined using 3 structures:
  - ✓ Sequences – perform sequence of actions (top-down)
  - ✓ Decisions – perform selection between alternative actions
  - ✓ Loops – perform repetition of same actions

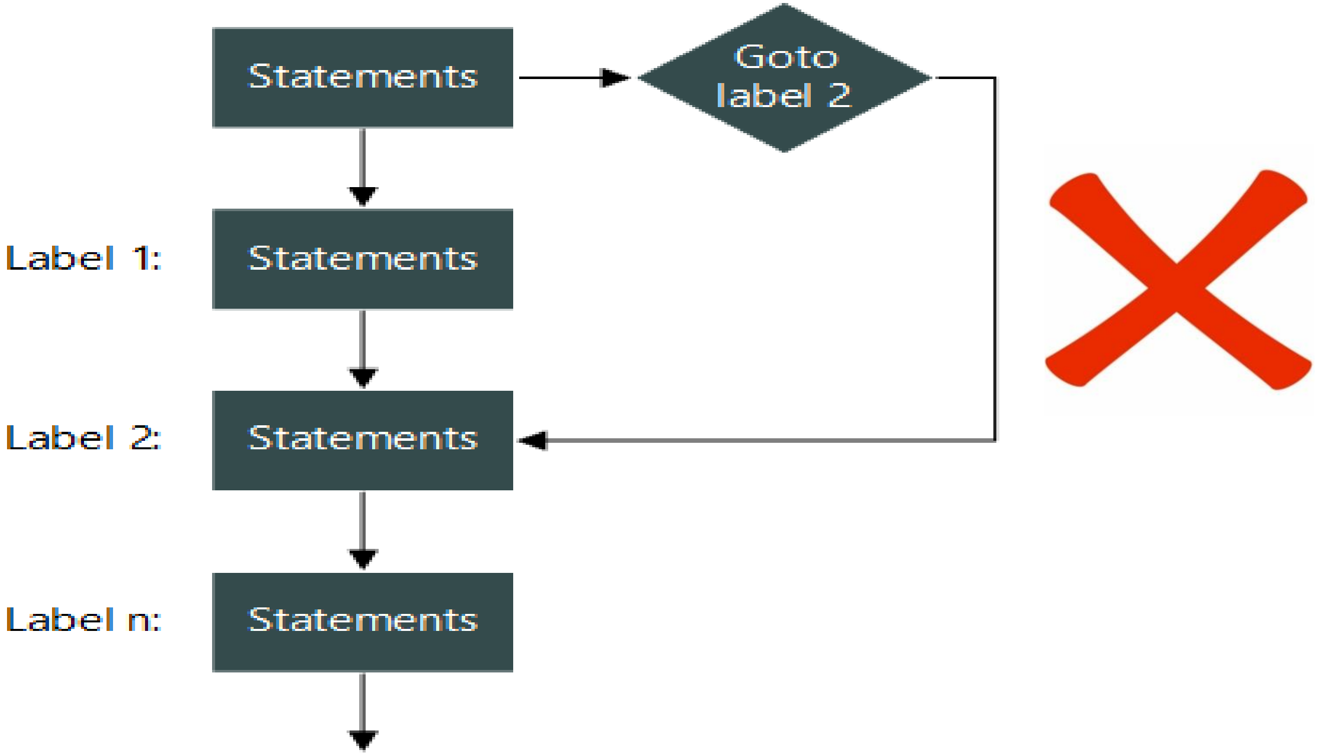
- ❑ **DOES NOT ALLOW to use the jump statement** (GOTO) to transfer program control from one line to another
- ❑ Why ?
- ❑ Use of GOTO increases the complexity of code and program becomes harder to maintain/modify *(results in spaghetti code)*

Structured Programming



Structured Programming

- Dijkstra, E. W., "*Go To Considered Harmful*," Communications of the ACM, March 1968
- Any program construction could be created more simply with the sequence, repetition and decision constructions WITHOUT USING GOTO

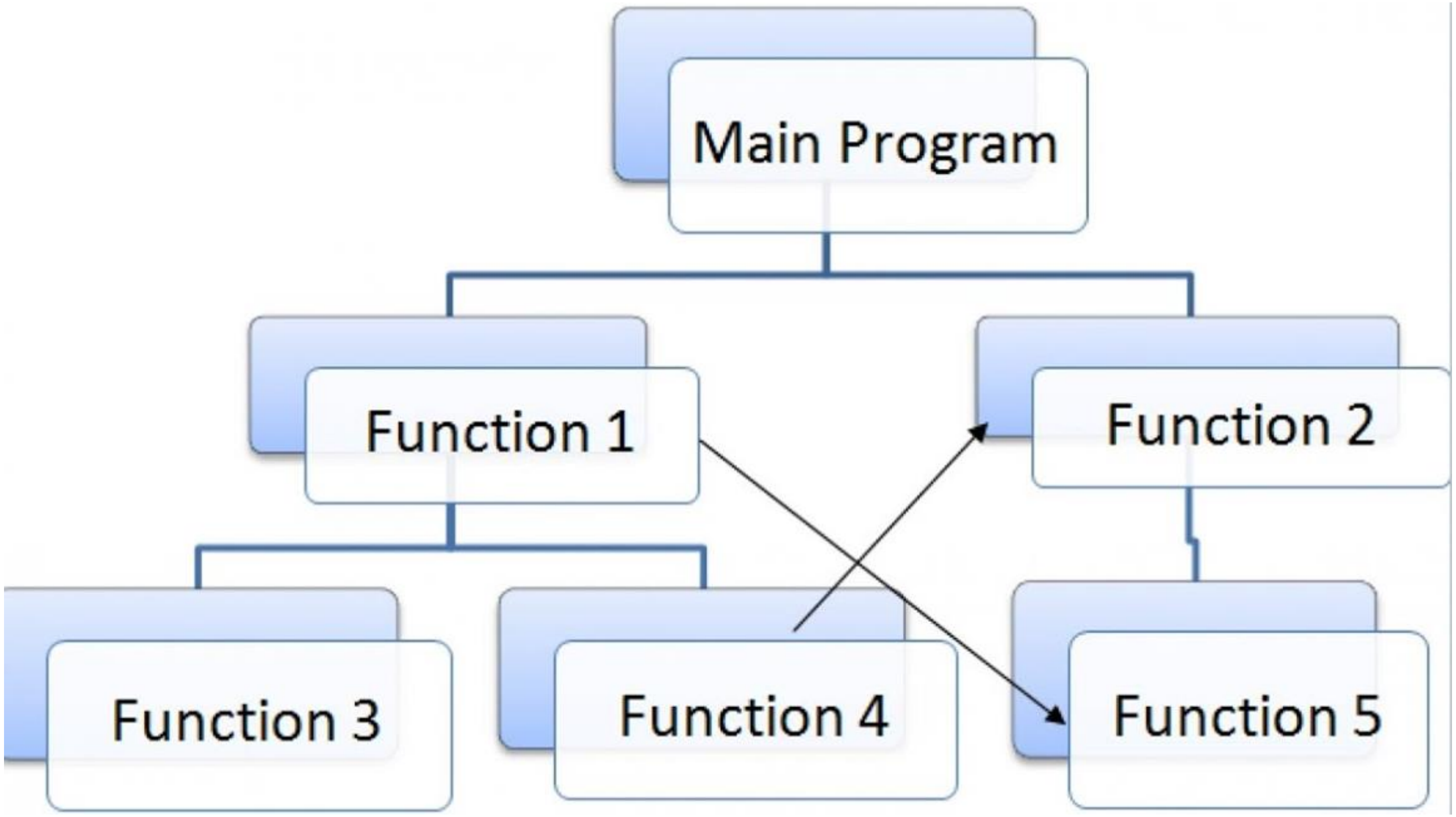


Procedure Oriented Programming

- Whole program is divided into collection of functions/procedures/modules/sub-routine/method
- Fundamental building blocks of a program are functions
- Function contains a set of statements to perform a task.
- Provides ability to reuse the code using functions.
- E.g. - C: developed by Dennis Ritchie and Ken Thompson

```
int sum(int num1, int num2) {
   int num3;
   num3 = num1 + num2;
   return (num3);
}
```

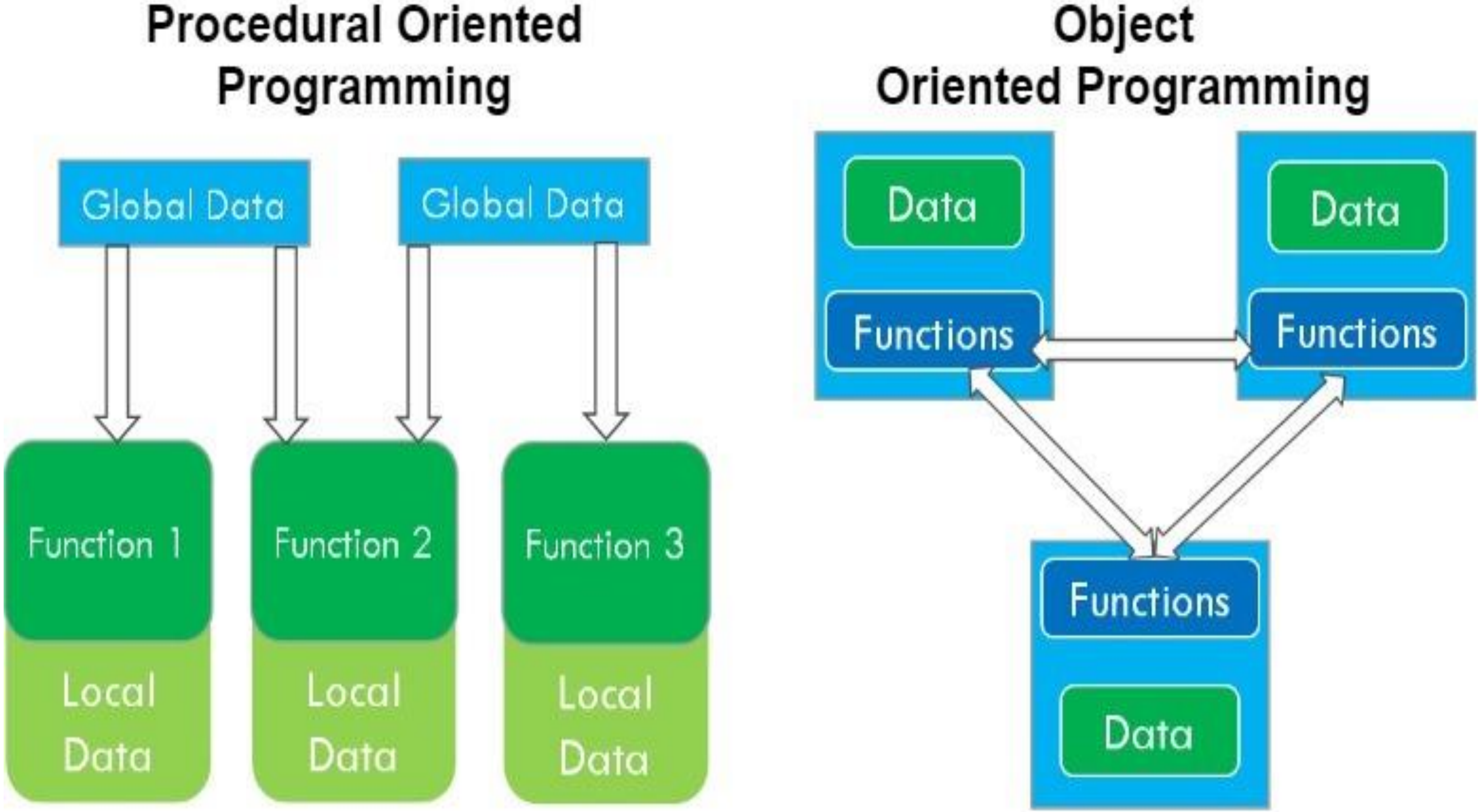
Procedure Oriented Programming



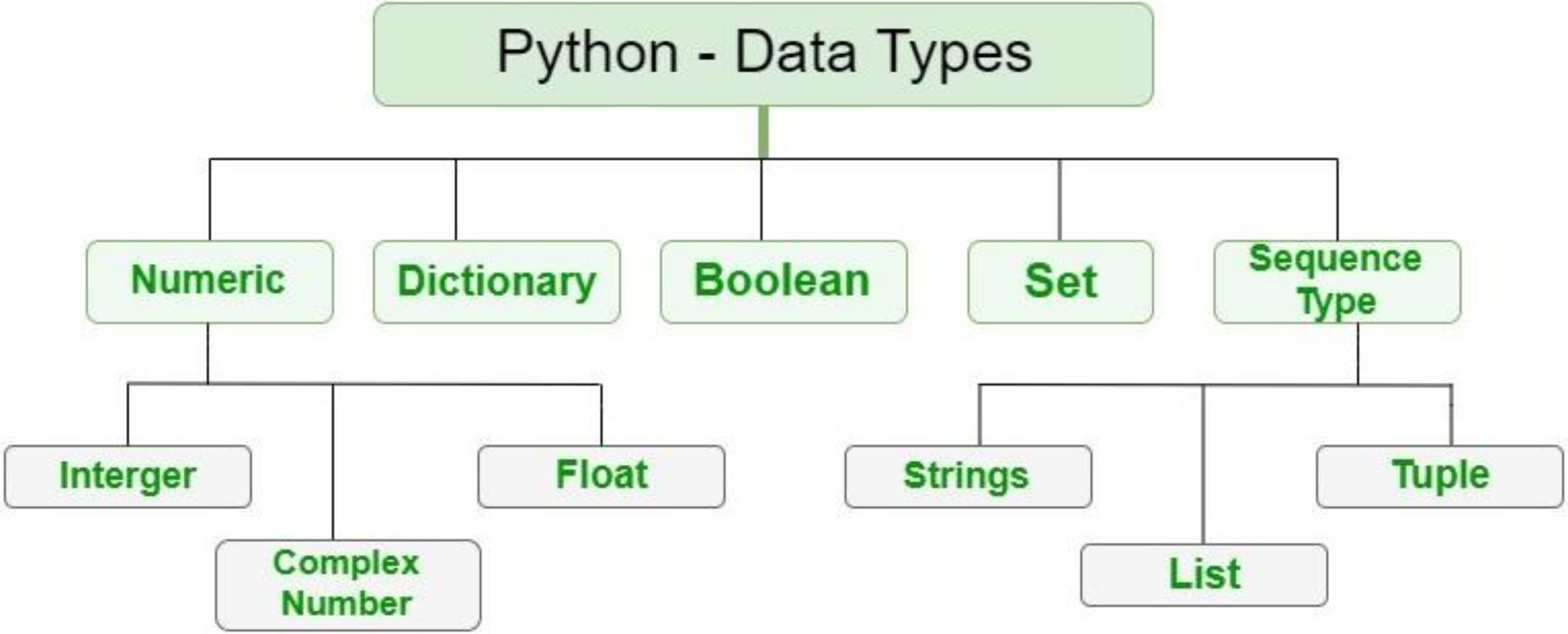
Object Oriented Programming

- Design and develop programs around Classes/objects which represent real world entities like Student, Customer, Employee, BankAccount etc.
- Object is the fundamental building block of program
- Object – encapsulates data and functions together into a single unit.
- Advantages - data hiding, code reusability, high level/less complexity and easier to modify/maintain the code, Easier to map real world entities into program

Procedure Vs Object Oriented Programming



Data Types


Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Complex Number
Float
Strings
List
Tuple

Python Programming

- Data-Types and Variables

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

- pi=3.14
- Print(type(pi)) # <class “float”>
- Print(isinstance(3.14,float)) or Print(isinstance(pi,float)) #true

Reference : www.w3schools.com

- Data-Types and Variables

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Reference : www.w3schools.com

- Data-Types and Variables

Example	Data Type
x = str("Hello World")	str
x = int(20)	int
x = float(20.5)	float
x = complex(1j)	complex
x = list(("apple", "banana", "cherry"))	list
x = tuple(("apple", "banana", "cherry"))	tuple
x = range(6)	range
x = dict(name="John", age=36)	dict
x = set(("apple", "banana", "cherry"))	set
x = frozenset(("apple", "banana", "cherry"))	frozenset
x = bool(5)	bool
x = bytes(5)	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Reference : www.w3schools.com

Python Programming

- Operators

Types of Operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Other Operators

Arithmetic Operators

Arithmetic Operators : Performs basic mathematical calculations.

1. + (Additon / Sum)

a = 4 + 8 + 3

where a is 15

2. – (Subtraction)

a= 8 – 5

Where a is 3

b = 12 - 3 - 5

Where b is 4

Arithmetic Operators

Arithmetic Operators : Performs basic mathematical calculations.

3. * (Multiplication)

a = 4 * 8 * 3

where a is 96

4. / (Division) always returns floating point number

a= 8 / 5

Where a is 1.6

b = 60 / 5 / 4 / 2

Where b is 1.5

Arithmetic Operators

Arithmetic Operators : Performs basic mathematical calculations.

5. // (Floor Division)

a = 60 // 8

where a is 7

a= -11//3

Where a is -4

6. % (Modulo remainder)

a= 8 % 5

Where a is 3

Arithmetic Operators

Arithmetic Operators : Performs basic mathematical calculations.

7. ** (used for exponent or power)

Eg. 2**5 = 32 (2 raised to 5)

Relational / Comparison Operators

Relational / Comparison Operators : Compares two operands and returns Boolean value (True / False). They are used to create conditions which evaluates to True or False. Hence used in condition driven control statements like decision making and looping structures.

1. > (greater than)

Eg. 2 > 5 returns False

2. < (less than)

Eg. 2 < 5 returns True

3. >= (greater than or equals to)

Eg. 5 >= 5 returns True

Relational / Comparison Operators

Relational / Comparison Operators : Compares two operands and returns Boolean value (True / False).

4. <= (less than or equals to)

Eg. 2 <= 2 returns True

5. == (equals to)

Eg. 2 == 5 returns False

Eg. 5 == 5 returns True

6. != (not equal to)

Eg. 5 != 5 returns False

6 != 5 returns True

Logical Operators

Logical Operators are used to check multiple conditions simultaneously

1. and (logical and) : Here all the conditions must be true then only overall result is true

Truth Table

True	and	True	=	True
True	and	False	=	False
False	and	True	=	False
False	and	False	=	False

Eg 5>2 and 6>3 and 10>7 is True

5>2 and 6<3 and 10>7 is False

Logical Operators

2. or (logical or) : Here from multiple conditions, only one must be true, then the whose expression is true.

Truth Table

True	or	True	=	True
True	or	False	=	True
False	or	True	=	True
False	or	False	=	False

Eg 5>2 or 6>3 or 10>7 is True

5>12 or 6<3 or 10>7 is True

5>12 or 6<3 or 10>17 is False

Logical Operators

3. not (logical not) : It reverses the result of condition

Truth Table

not True = False

not False = True

Eg not (5>2 or 6>3 or 10>7) is False

not (5>12 and 6<3 and 10>7) is False

not (5>12 and 6<3 and 10>17) is True

Assignment Operators

Assignment Operators : Assigns values from Right Hand side (RHS) to Left Hand Side (LHS). Hence LHS must be a valid variable and RHS can be any expression or a single variable.

1. = (Assignment operator)

Eg. **a = 5** # Here **a** (LHS) is variable name, = is assignment operator and 5 is a value or can be variable

Eg. **b = a** # copies/overwrites value of variable **a** to **b, hence b is also 5 now.**

Eg. b = a * 3 +4 #Here value of b is 19 , Here RHS is a mathematical expression

Assignment Operators

2. +=

Eg. **a += 5** # Here the expression is equivalent to a=a+5

3. -=

Eg. **a -= 5** # It is equivalent to a=a-5

4. *=

Eg. **a *= 5** # It is equivalent to a=a*5

5. /=

Eg. **a /= 5** # It is equivalent to a=a/5

Assignment Operators

6. %=

Eg. **a %= 5** # It is equivalent to a=a%5

7. //=

Eg. **a //= 5** # It is equivalent to a = a//5

8. **=

Eg. **a **= 5** # It is equivalent to a=a**5

Bitwise Operators

Bitwise Operators are used to manipulates bits at bit level, where minimum unit of data is considered as bit. Hence it operates on 1’s and 0’s. They make calculation process faster. Also used for several bit level hardware operations where a single byte is used to store states of hardware signals.

1. & (Bitwise AND)

Where 1 & 1 = 1,
1 & 0 = 0,
0 & 1 = 0,
0 & 0 = 0

Eg. **a = 20 (**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0)**	
b = 15 (	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1)**	**c = a & b = 4**
c = 4 (	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0)**	

Bitwise Operators

2. | (Bitwise OR)

Where 1 | 1 = 1,
1 | 0 = 1,
0 | 1 = 1,
0 | 0 = 0,

Eg. **a = 20 (0 0 0 1 0 1 0 0)**
b = 15 (0 0 0 0 1 1 1 1) **c = a | b = 31**
c = 31 (0 0 0 1 1 1 1 1)

Bitwise Operators

3. ^ (Exclusive OR)

Where 1 ^ 1 = 0,
1 ^ 0 = 1,
0 ^ 1 = 1,
0 ^ 0 = 0,

Eg. **a = 20 (0 0 0 1 0 1 0 0)**
b = 15 (0 0 0 0 1 1 1 1) **c = a ^ b = 27**
c = 27 (0 0 0 1 1 0 1 1)

Bitwise Operators

4. ~ (One’s Complement) It is unary operator, it flips the bit

Where ~1 = 0,

~0 = 1

Eg a=20 (0 0 0 1 0 1 0 0) Now if complement is performed a = ~a then

```
    0 0 0 1 0 1 0 0 (20)
  + 0 0 0 0 0 0 0 1  (1)
    0 0 0 1 0 1 0 1 (21) Hence we get 21
```

Then to get sign do complement of above no

```
    1 1 1 0 1 0 1 0 (now get -ve no.)This
```

value is stored at memory location

After the operation a=~a

Now, how the value of a is displayed (continue)

Bitwise Operators

1 1 1 0 1 0 1 0 (First read higher bit which is one.

Hence we got –ve sign

Now complement the above digits we get

0 0 0 1 0 1 0 1 (21)

Hence value is -21

This is how signed number managed in python.

Bitwise Operators

5. >> (Right Shift Bitwise) : It shifts bits towards right.

   a=20;

Eg. **a = 20 (0 0 0 1 0 1 0 0) (20)**

a = a>>1

```
         0  0  0  1  0  1  0  0   (20)
 -->0    0  0  0  0  1  0  1  0   (10) (add zero from left hence
                                         (1 bit is lost from
                                          right side)
```

Always number is halved, if we shift one bit towards right.

Bitwise Operators

6. << (Left Shift Bitwise) : It shifts bits towards left.

a=20;

Eg. **a = 20 (0 0 0 1 0 1 0 0) (20)**

a = a<<1

0 0 0 1 0 1 0 0 (20)

0 0 1 0 1 0 0 0 0← (Now 40)(add zero from right hence (1 bit is lost from left side)

Always number is doubled, if we shift one bit towards left.

Precedence of Arithmetic Operators

It is always through same mathematical principal

Ie B O D M A S

(Bracket of Expansion, Division, Multiplication, Addition, Subtraction)

Eg

8 + 2 * (3 - 2) + 3 * 8 / 4

8 + 2 * (1) + 3 * 8 / 4

8 + 2 * (1) + 3 * 2

8 + 2 * (1) + 6

8 + 2 + 6

16

Precedence of Arithmetic Operators

It is always through same mathematical principal

Ie B O D M A S

(Bracket of Expansion, Division, Multiplication, Addition, Subtraction)

Eg

$8 + 2 * (3 - 2) + 3 * 8 / 4$

$8 + 2 * (1) + 3 * 8 / 4$

$8 + 2 * (1) + 3 * 2$

$8 + 2 * (1) + 6$

$8 + 2 + 6$

16

Membership Operators

(1) in : Returns true if finds a value in sequence otherwise false

Eg

```
x = 22
marks = [ 10, 34, 22, 78, 33 ]
if ( x in marks ) :
    print(x, “ is in marks list”)
else:
    print(x, “ is not in marks list”)
```

OUTPUT :

```
22 is in marks list
```

Membership Operators

(2) not in : Returns true if finds a value in sequence otherwise false

Eg

```
x = 22
marks = [ 10, 34, 22, 78, 33 ]
if ( x not in marks ) :
    print(x, “ is not in marks list”)
else:
    print(x, “ is in marks list”)
```

OUTPUT:

```
22 is in marks list
```

Identity Operators

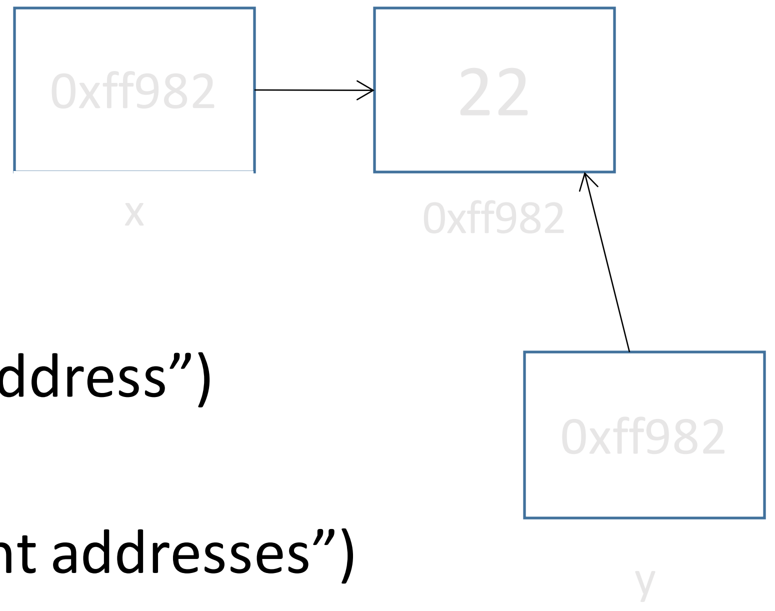
(1) is : Compare memory location of objects.

Eg

```
x = 22
y = 22
if( x is y):
        print (“x and y has same address”)
else
        print (“x and y has different addresses”)
```

OUTPUT

```
        x and y has same address
```



Identity Operators

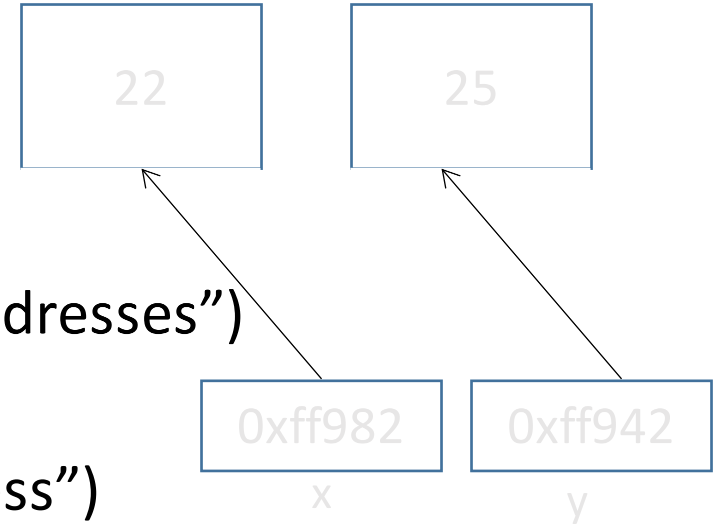
(2) is not : Compare memory location of objects.

Eg

```
x = 22
y = 25
if( x is not y):
        print (“x and y has different addresses”)
else
        print (“x and y has same address”)
```

OUTPUT

```
        x and y has different addresses
```



Type conversion in Operation and Precedence

Python Operator precedence

Operator Precedence from Highest to lowest

1. ** (Exponent)
2. ~ (bitwise unary complement)
3. + (unary plus)
4. - (unary minus)
5. * (multiplication)
6. / (division)
7. % (modulo remainder)
8. // (floor division)
9. + (binary plus)
10. - (binary minus)

Python Operator precedence

Operator Precedence from Highest to lowest

11. >> (Right Shift)
12. << (Left shift)
13. & (Bitwise and)
14. ^ (Bitwise Exclusive OR)
15. | (Bitwise OR)
16. <= (less than or equal to)
17. < (less than)
18. > (greater than)
19. >= (greater than or equal to)
20. == (equal to)
21. != (not equal to)

Python Operator precedence

Operator Precedence from Highest to lowest

22.= (Assignment operator)

23.%= (modulo assignment)

24./= (division assignment)

25.//= (floor division assignment)

26.-= (subtract assignment)

27.+= (plus assignment)

28.*= (multiply assignment)

29.**= (exponent assignment)

Python Operator precedence

Operator Precedence from Highest to lowest

30.is (Identity Operator)

31.is not (Identity Operator)

32.in (Membership)

33.in not (Membership)

34.not (logical not)

35.or (logical or)

36.and (logical and)

Type Conversion in Python

Revisiting Datatypes

There 5 standard data-types

1. Numbers
2. String
3. List
4. Tuple
5. Dictionary

Type Conversion in Python

Revisiting Datatypes

1. **Number**

Integer eg. are 12, -56, 032 (octal), 0x69a (hex)

Long eg. are 567L, 0x5678aL,

Float eg are 5.6, -34.23, 56.3+e9, 45.2-e4

Complex eg. are 4.78j,

Type Conversion in Python

Revisiting Datatypes

2. String

```
 str=“Navrachana University”
print(str) # displays ‘Navrachana University’
print (str[0]) # displays ‘N’
print(str[5:9]) #displays ‘chana’
print(str[5:]) #displays ‘chana University’
print(str + “ is great”) # displays Navrachana University is great
print(str * 3) # displays 3 times Navrachana University
```

Type Conversion in Python

Revisiting Datatypes

3. List

lst=["java","python",56,80.33]

print(lst[0]) #displays java

print(lst[1:3]) #displays ['python', 56]

Lst[1]=“C#” #edit is possible

Other example command applies like string operations

Type Conversion in Python

Revisiting Datatypes

4. Tuple

List uses [] bracket, while tuple uses () brackets.

List is mutable, while tuple is immutable

```
tup=("java","python",56,80.33)
print(tup[0])  #displays java
print(tup[1:3])  #displays ['python', 56]
Tup[1]=“C#”  # error
```

Type Conversion in Python

Revisiting Datatypes

5. Dictionary

Uses {} brackets, are hash tables, store data in key-value pairs

contacts={"Raju":98747757,"Amit":858583833,"Jai":23445566}

print(contacts) #displays {'Raju': 98747757, 'Amit': 858583833, 'Jai': 23445566}

print(contacts["Raju"]) # displays 98747757

Type Conversion in Python

Revisiting Datatypes

6. Set

Uses {} brackets , use hashing to store information, suitable for faster searching. As data values are stored through hash values. The time complexity is Big O(1).

```
>>> data={123, 654, 678, 1000}
>>> print(data)
{1000, 123, 678, 654}
```

Type Conversion in Python

Conversion Functions

1. int()

X=“567”

X=int(X) # converts string x to integer x (string to numeric format so that proper mathematical and relational operators can be applied.

2. float()

Y=“34.66”

Y=float(Y) # converts string float format to numeric float format

Type Conversion in Python

Conversion Functions

3. str() : Converts to String format

```
x=45
Str(x) # converts integer x to string x
```

4. ord() : Converts Character to Integer

```
    eg
     >>> ord('A')
           65
```

5. hex() : Converts integer to hexadecimal format.

```
     eg. >>> hex(210)
                '0xd2'
```

Type Conversion in Python

Conversion Functions

6. oct : Converts integer to octal number format.

```
>>> oct(210)
'0o322‘
```

7. bin() : Converts integer to binary number format.

```
>>> bin(20)
'0b10100‘
```

Type Conversion in Python

Conversion Functions

8. list : Converts any iterable object to list. Allows duplicates too.

Eg

```
>>> x="python"
>>> lst=list(x)
>>> print(lst)
['p', 'y', 't', 'h', 'o', 'n']
```

Eg 2

```
>>> j=543
>>> lst=list(j)
Traceback (most recent call last):
  File "<pyshell#39>", line 1, in <module>
    lst=list(j)
TypeError: 'int' object is not iterable
```

Type Conversion in Python

Conversion Functions

9. set : Converts any iterable object to set. Duplicates are not allowed.

Eg

```
>>> x="java"
>>> se=set(x)
>>> print(se)
{'j', 'a', 'v'}
>>> Eg 2
>>> x="python"
>>> lst=list(x)
>>> se=set(lst)
>>> print(se)
{'n', 't', 'p', 'y', 'h', 'o'}
```

Type Conversion in Python

Conversion Functions

10. tuple : Converts any iterable object to tuple which are immutable. Duplicates are allowed.

Eg

```
>>> prg="Java"
>>> tup=tuple(prg)
>>> print(tup)
('J', 'a', 'v', 'a')
```

Eg.

```
>>> tup[2]='w'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    tup[2]='w'
TypeError: 'tuple' object does not support item assignment
```

Type Conversion in Python

Conversion Functions

11. dict : Converts tuple and list of order key-value to dictionary but not set as set requires hashing.

Eg 1

```
>>> tup = (('Ajay', 984477555) ,('Fenil', 97636363), ('Nikita', 764636333))
>>> di=dict(tup)
>>> print(di)
{'Ajay': 984477555, 'Fenil': 97636363, 'Nikita': 764636333}
```

Python Lists

What is a list in Python?

- In Python, a list is one of the basic data-structure useful to store multiple data items.
  - Python ***list*** is the compound data types which can be used to *group values*.
  - List is a kind of *collections* in Python.
  - The ***list*** can be written as a list of ***comma-separated values*** (items) between ***square brackets***.
  - Lists might contain items with different data types.
  - **Let’s use Python’s List.**

Creating a Python's List

Following example will create a list of characters, numbers and words

```
# List of characters

>>> myList = ['a', 'b', 'c', 'd', 'e']
>>> myList
['a', 'b', 'c', 'd', 'e']

# List of numbers

>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]

# List of words

>>> insurance = ['sbi', 'hdfc', 'icici', 'lic']
>>> insurance
['sbi', 'hdfc', 'icici', 'lic']
```

Accessing Values from a list

- ❑ Elements of List can be used by referring its index number. For example, to print first value from list, use the following code:

```
>>> insurance[0]
'sbi'

>>> print(insurance[0])
sbi
```

Accessing Values from a list

❑ Like strings, lists can also be indexed and sliced:

```
# index 0 returns the first item
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0]
1

# index -1 returns the last item
>>> squares[-1]
25

# slicing returns a new list
>>> squares[-4:]
[4, 9, 16, 25]
```

Accessing Values from a list

❑ All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

❑Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Accessing Values from a list

- ❑ Lists are ***mutable*** type; hence it is possible to change the content of the list. Check the following examples:

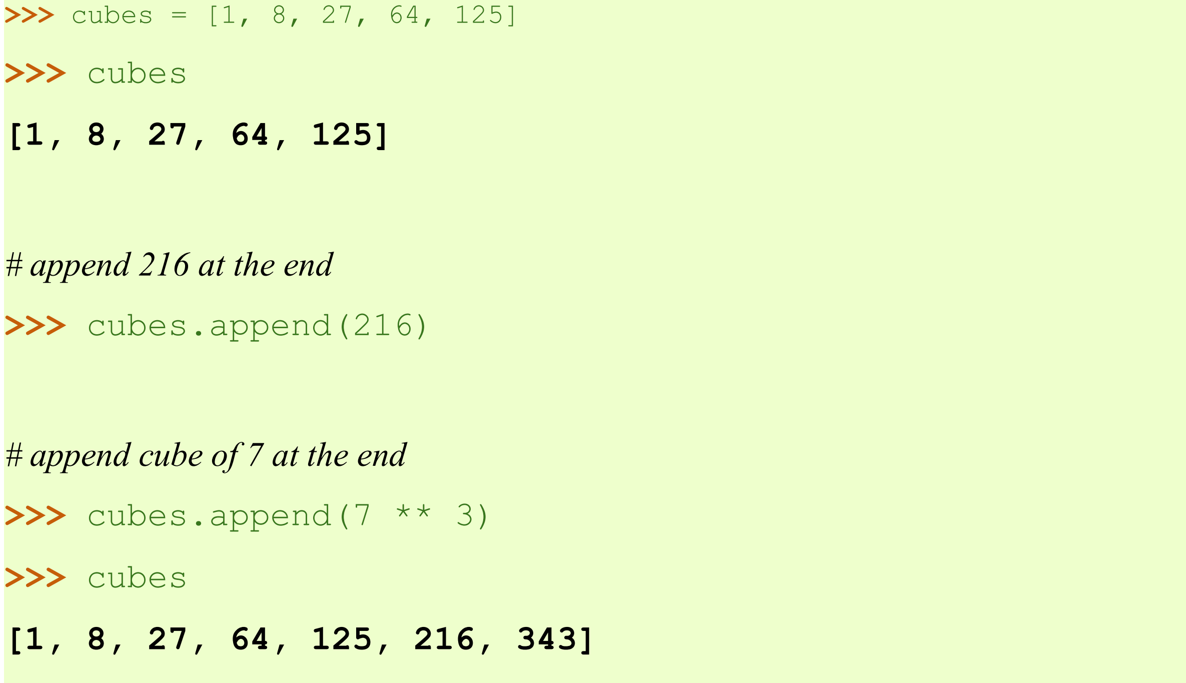
```
# something's wrong in following code; cube of 4 is not 65; let's change it.

>>> cubes = [1, 8, 27, 65, 125]
>>> cubes[3] = 64    # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

❑ Using ***append()*** with List.

❑ ***append()*** function can be used to add new items at the end of the list:



```
>>> cubes = [1, 8, 27, 64, 125]
>>> cubes
[1, 8, 27, 64, 125]

# append 216 at the end
>>> cubes.append(216)

# append cube of 7 at the end
>>> cubes.append(7 ** 3)
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

- Using ***insert()*** with List.
- ***insert()*** function will add element to a list at specified position (Index number). Check the following example:

```
>>> list1 = ['a', 'b', 'c']
>>> list1.insert(2, 'kp')
>>> list1
['a', 'b', 'kp', 'c']
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

- ❑ Using ***extend()*** with List.
- ❑ ***extend()*** function combines two lists. Check the following example:

```
>>> list1 = ['a', 'b', 'c']
>>> list1.insert(2, 'kp')
>>> list1
['a', 'b', 'kp', 'c']
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

- ❑ Using ***pop()*** with List.
- ❑ ***pop()*** function will get the value of an item and remove it from the given list.
- ❑ ***pop()*** function needs index number of a list as a parameter. Check the following example:

```
>>> list1 = ['a', 'b', 'c', 'd']
>>> list1

['a', 'b', 'c', 'd']

>>> list1.pop(2)

'c'

>>> list1

['a', 'b', 'd']
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

- Using ***remove()*** with List.
- ***remove()*** function deletes an item from a list.
- ***remove()*** function needs actual value of an item as a parameter. Check the following example:

```
>>> list1 = ['a', 'b', 'c', 'd']

>>> list1.remove('d')

>>> list1

['a', 'b', 'c']
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

❑ Using ***reverse()*** with List.

❑ ***reverse()*** function will reverse the elements of a given list.

```
>>> list1 = ['a', 'b', 'c', 'd']
>>> list1

['a', 'b', 'c', 'd']

>>> list1.reverse()

>>> list1

['d', 'c', 'b', 'a']
```

Python Lists – Built-in Methods

Let’s learn some of the top built-in methods available for Python-lists

❑ Using ***sort()*** with List.

❑ ***sort()*** function will sort the elements of a given list alphabetically or numerically.

```
>>> list1 = ['c', 'b', 'a', 'd']
>>> list1

['c', 'b', 'a', 'd']

>>> list1.sort()

>>> list1

['a', 'b', 'c', 'd']
```

Python Lists – Built-in Functions

Let’s learn some of the top built-in functions available for Python-lists

❑ Using ***len()*** with List.

❑ ***len()*** finds the number of items or elements in a list. Following code will return the length of the list-insurance

```
>>> insurance = ['sbi', 'icici', 'hdfc', 'hsbc']
>>> print(len(insurance))
4
```

Python Lists – Built-in Functions

Let’s learn some of the top built-in functions available for Python-lists

- ❑ Using ***min() and max()*** with List.
- ❑ Minimum and Maximum value can be identified using ***min()*** and ***max()*** function from the lists as show below:

```
>>> list1 = [1, 2, 5, 10, 8]
>>> print(min(list1))
1

>>> print(max(list1))
10
```

Operations on Python Lists

- It is possible to perform operations like addition, multiplication, and searching on Python Lists.
- Addition Operation on Python Lists.
- Addition operation on list is possible using **+** operator:

```
>>> list1 = [1, 2]
>>> list1
[1, 2]

>>> list2 = [3, 4]
>>> list2
[3, 4]

>>> list1 + list2
[1, 2, 3, 4]
```

Operations on Python Lists

- It is possible to perform operations like addition, multiplication, and searching on Python Lists.
- Multiplication Operation on Python Lists.
- Multiplication operation on list is possible using ***** operator:

```
>>> mylist = [1, 2]
>>> mylist
[1, 2]

>>> mylist * 3
[1, 2, 1, 2, 1, 2]
```

- The **+** and ***** operations do not modify the list. The original list will remain same even after use of **+** and ***** symbols on it.

Operations on Python Lists

- It is possible to perform operations like addition, multiplication, and searching on Python Lists.
- Search Operation on Python Lists.
- **in** keyword can be used to search an element in the given list.

```
>>> insurance = ['sbi', 'icici', 'hdfc', 'hsbc']
>>> insurance
['sbi', 'icici', 'hdfc', 'hsbc']

>>> 'sbi' in insurance
True

>>> 'seas' in insurance
False
```

Python Sets

(Collections)

Python Collections

Python Collections

Collections in Python are containers that are used to store collections of data. General purpose built-in containers of Python programming language are listed below:

- list
- set
- tuple
- dist

Python - list

- **In Python, a list** is one of the basic data-structure useful to store multiple data items.
  - Python ***list*** is the compound data types which can be used to *group values*.
  - List is a kind of *collections* in Python.
  - The ***list*** can be written as a list of ***comma-separated values*** (items) between ***square brackets***.
  - Lists might contain items with different data types.
  - **Let’s use Python’s List.**

Creating a Python's List

Following example will create a list of characters, numbers and words.

```
# List of characters

>>> myList = ['a', 'b', 'c', 'd', 'e']
>>> myList
['a', 'b', 'c', 'd', 'e']

# List of numbers

>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]

# List of words

>>> insurance = ['sbi', 'hdfc', 'icici', 'lic']
>>> insurance
['sbi', 'hdfc', 'icici', 'lic']
```

Accessing Values from a list

- Elements of List can be used by referring its index number. For example, to print first value from list, use the following code:

```
>>> insurance[0]
'sbi'

>>> print(insurance[0])
sbi
```

Python Set

Introduction to Python – Set

- A set is an unordered collection of elements.
- Every element is unique (no duplicate values in set) and are immutable (i.e. values cannot be changed). However, the set itself is mutable which means we can add or remove elements from set.

- Sets is very useful to perform mathematical set operations like union and intersection.

Python Set

Create a Python Set

- A set is created by placing all the elements within curly braces { } and each elements separated by comma.
- Set can have any number of elements with different data types like integer, float, string etc.
- **Example of set of an integers**

```
>>> set1 = {1, 2, 3}
>>>set1
{1, 2, 3}
```

Python Set

Create a Python Set

```
# set of mixed data types

>>>set2 = {101, "Kuntal Patel", 90.5}
>>>print(set2)
{'Kuntal Patel', 90.5, 101}
```

Python Set

Create a Python Set

Set removes duplicate entries and arranges elements in order.

```
>>> set3 = {1,3,2,5,4,3}
>>> set3
{1, 2, 3, 4, 5}
```

Python Set

Create a Python Set

Observe the following code:

```
>>> set2[0]
Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
set2[0]
TypeError: 'set' object is not subscriptable
```

Note: 'set' object does not support indexing in python. We can’t access an element of set using indexing or slicing concept.

Accessing Values from a Set

Changing Python Set Elements

Add() method can be used to add single element in to a set.

```
>>> set4 = {3, 5, 7}
>>> set4
{3, 5, 7}

>>> set4.add(6)
>>> set4
{3, 5, 6, 7}
```

Accessing Values from a list

Removing elements from Set

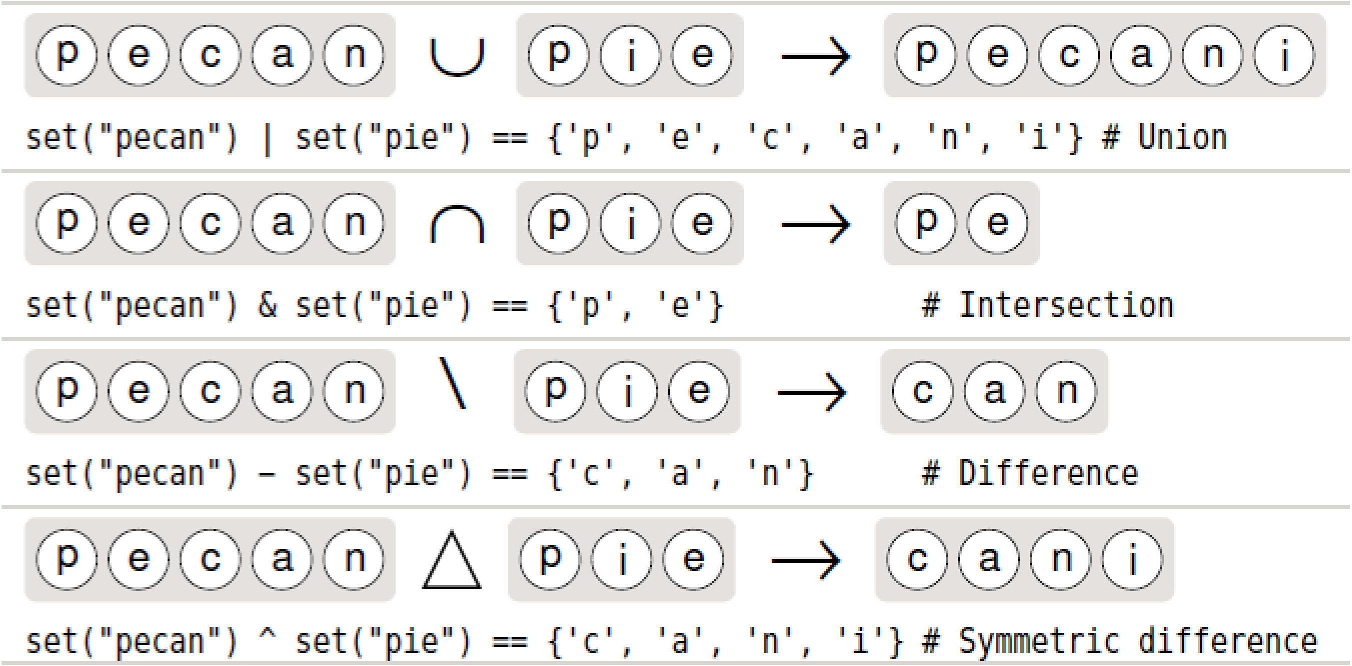
```
>>> set4.remove(7)
>>> set4
{3, 5, 6}
```

Note: Trying to remove element which does not exists in set will result in error. Observe the following code:

```
>>> set4(2)
Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
set4(2)
TypeError: 'set' object is not callable
```

Python Set Operations

Standard Set Operators


p e c a n ∪ p i e → p e c a n i
p e c a n ∩ p i e → p e
p e c a n \ p i e → c a n
p e c a n △ p i e → c a n i

```
set("pecan") | set("pie") == {'p', 'e', 'c', 'a', 'n', 'i'} # Union
set("pecan") & set("pie") == {'p', 'e'}           # Intersection
set("pecan") - set("pie") == {'c', 'a', 'n'}      # Difference
set("pecan") ^ set("pie") == {'c', 'a', 'n', 'i'} # Symmetric difference
```

Figure 3.4 *The standard set operators*

Reference: Ch. 3 - Programming in Python 3 by Summerfield

Python Set Operations

Python Set Operations - Union and Intersection

Python sets can be used to perform mathematical set operations like union and intersection.

Python Set Union

Union of Set A and Set B is a set of all elements from both the python sets. Union operation can be performed using union operator(|) or the method union().

```
>>> set5 = {2, 4, 6, 8}
>>> set6 = {6, 8, 10, 12}
>>> set5.union(set6)
{2, 4, 6, 8, 10, 12}

>>> print(set5 | set6)
{2, 4, 6, 8, 10, 12}
```

Python Set Operations

Python Set Intersection

Intersection of Set A and Set B is a set of elements that are common in both sets. Python intersection can be performed using intersection operator (&) or using the method intersection().

```
>>> set5 & set6
{8, 6}

>>> set5.intersection(set6)
{8, 6}
```

Python Set Methods

Python Set Methods

- s.add(x) Adds item x to set s if it is not already in s
- s.clear() Removes all the items from set s
- s.discard(x) Removes item x from set s if it is in s; see also set.remove()
- s.union(t) s | t ; Returns a new set that has all the items in set s and all the items in set t that are not in set
- s.intersection(t) s & t ; Returns a new set that has each item that is in both set s and set t

Python Programming

- Control Statements
  - Decision Making (if, if else, if elif elif else)
  - Looping (while, for)

Python Programming

- Collection Data types (List, Tuple, Set, Dictionary)

Python Programming

- Functions

```
def function_name(argument1, argument2,…….):
        statement1
        statement2
        statement3
        return somevalue
```

Python Programming

- Lambda Functions / Anonymous Function

```
x = lambda a : a + 10
print(x(5))   #invoke lambada/anonymous function

........................

x = lambda a, b : a * b
print(x(5, 6)) #invoke lambada/anonymous function

........................

x = lambda a, b, c : a + b + c
print(x(5, 6, 2)) #invoke lambada/anonymous function
```

References

- Michael H. Goldwasser, David Letscher , “Object-oriented Programming in Python “ , Pearson Prentice Hall, 2008
- Dusty Phillips , “Python 3 Object Oriented Programming” ,PACLT publications.
- Retrieved and referred from docs.python.org, July 2021.
- Retrieved and referred from https://www.pcmag.com/encyclopedia, July 2021.
- Retrieved and referred from https://stackoverflow.com, July 2021.
- Retrieved and referred from https://wikipedia.org, July 2021.
- Retrieved and referred from https://w3schools.com, July 2021.