Python Programming

What are we going to cover ?

Introduction

- Overview
- Why python
- Versions

Setup

- Python installation
- Environment setup
- Git configuration

Foundation

- Data types
- Operators
- Statement & Expression

Types

- Basic data types
- Collections
  - List
  - Tuple
  - Set
  - Dictionary

In python functions are treated as variables

Functions

- Named vs Lambda
- Nested functions
- Scopes

Advanced features

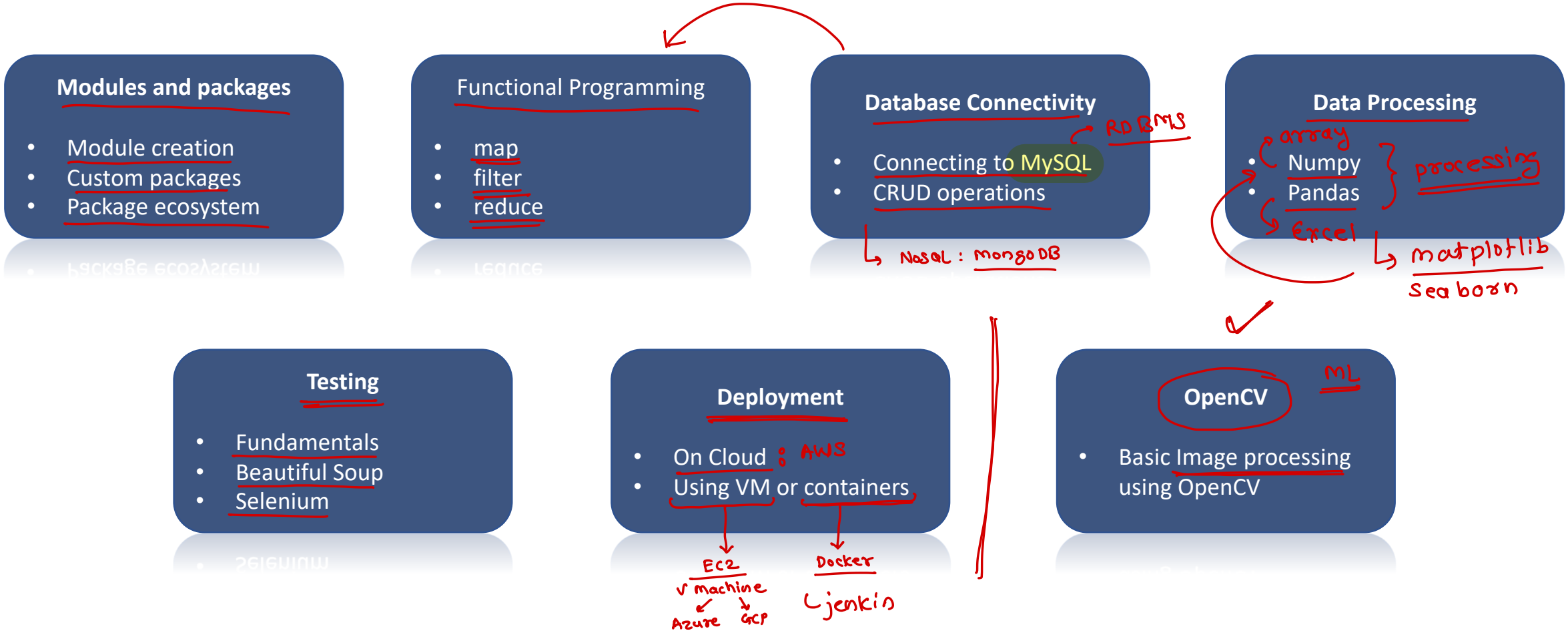
- List comprehension
- Generators
- Decorators → Flask → WS (REST)

OOP

- Class vs Object
- Inheritance
- Special methods

Python Is A versatile Language as it has
1)Python is non OOP or structured language.
2)It is also has OOP .
3)Its Functional language.
4)Its aspect oriented language.

What are we going to cover ?



What are we NOT going to cover ?

- Data Science
- Analytics and Analysis
- Machine Learning
- Image processing algorithms
- Web development
- Testing fundamentals
- Containerization (Docker)
- DevOps
- Linux administration
- Database fundamentals (MySQL/Mongo)
- CI/CD pipeline

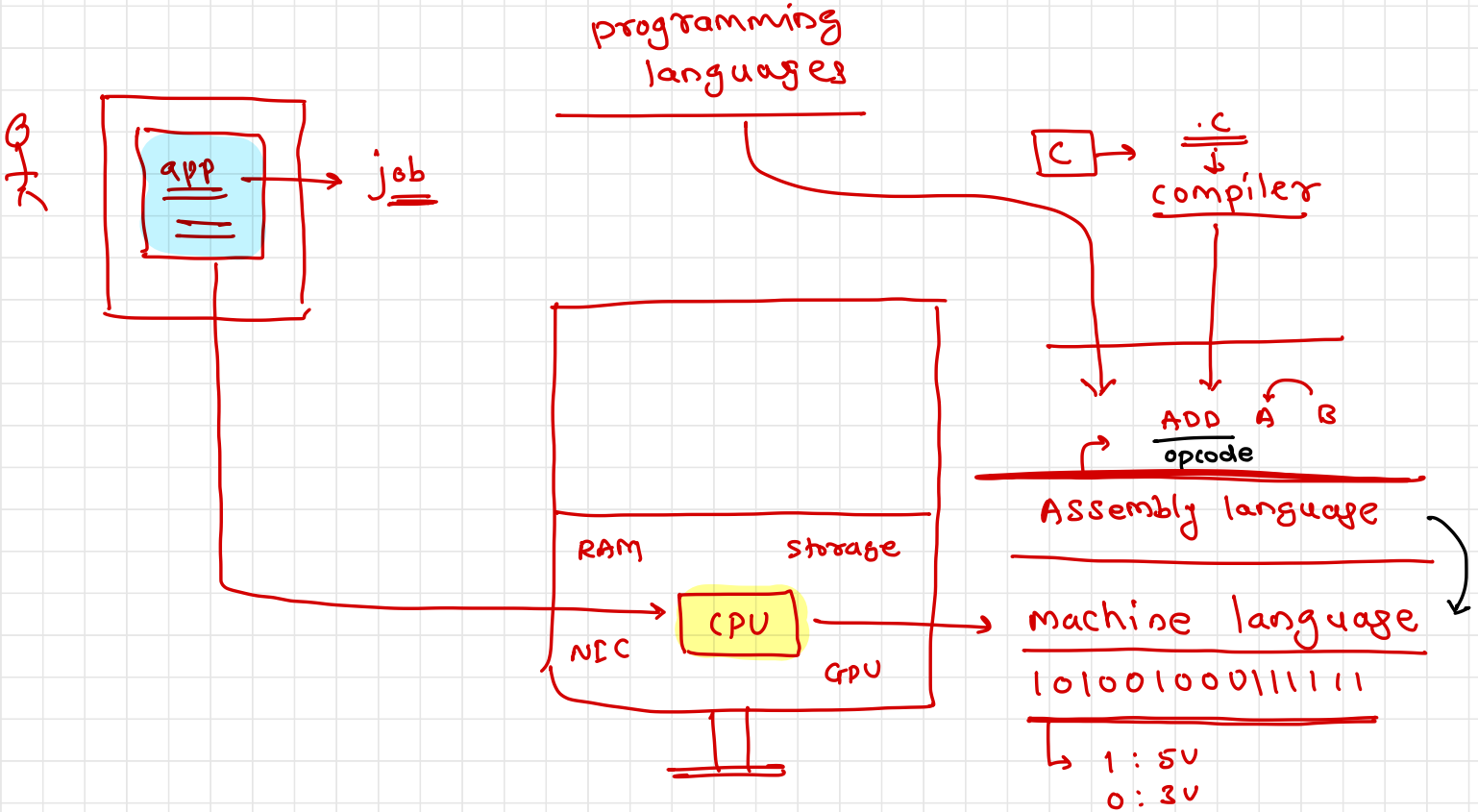
About your instructor

- 14+ years of development and Teaching experience
- Freelance Developer
- Associate Technical Director at Sunbeam
- Worked in various domains using different technologies
- Developed 180+ mobile applications on iOS and Android platforms
- Developed various web application using PHP, Python, MEAN and MERN stacks
- Certification completed
  - Certified Kubernetes Application Developer (CKAD)
  - Certified Kubernetes Administrator (CKA)
  - Certified Jenkins Engineer (CJE)
  - Docker Certified Associate (DCA)

Important Points

- The sessions will be conducted every Saturday and Sunday from 9:00 AM to 1:00 PM IST
- Session recordings will be provided on sunbeam LMS after every session on the same day
- The recordings will be available for 2 months
- Regular exercises will be provided
- Active participation from participants is expected

Language Fundamentals


programming languages
app
job
C
.c
compiler
ADD A B
opcode
Assembly language
RAM
storage
CPU
NIC
GPU
machine language
1010010001111111
1 : 5V
0 : 3V

What is a computer language?

- A language is a medium to interact with computer [CPU]
- A language is used to solve a problem by giving some solution (writing a program)
- Types
  - **Machine languages**: the code written in 0s and 1s and can be directly executed by CPU
  - **Assembly languages**: a language closely related to one or a family of machine languages, and which uses mnemonics to ease writing
  - Programming languages
    - **General purpose languages**: a programming language that is broadly applicable across application domains, and lacks specialized features for a particular domain (C, C++, python)
    - **Markup languages**: a grammar for annotating a document in a way that is syntactically distinguishable from the text (xml, yaml, HTML, xaml)
    - **Stylesheet language**: a computer language that expresses the presentation of structured documents (CSS)
    - **Configuration language**: a language used to write configuration files (Json, xml, .config, yaml)
    - **Query language**: a language used to make queries in databases and information systems [SQL]
    - **Scripting languages**: a language used to write simple to complex scripts [python, swift, javascript,...]

Cascading Style Sheet Language used for website decoration

cotlin,ruby,typescript.

Low Level vs High Level Languages

Because they generalize everything

As it is near to CPU

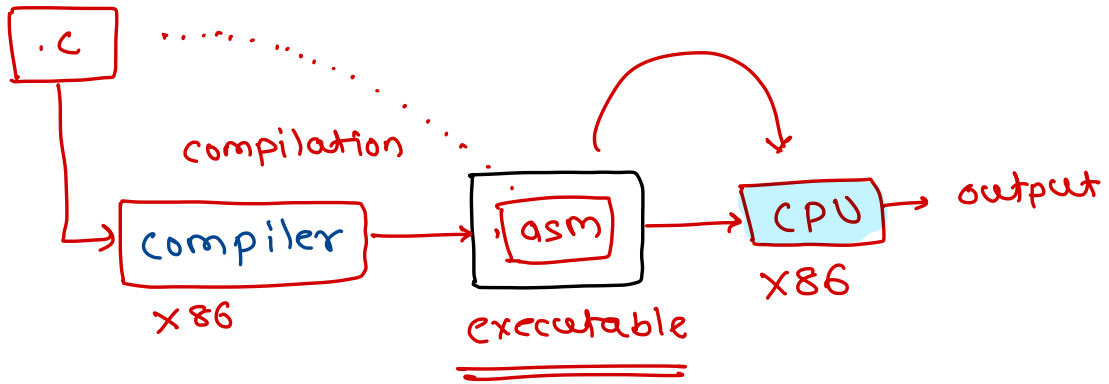
High Level Language	Low Level Language
It is programmer friendly language	It is a machine friendly language
High level language is less memory efficient	Low level language is high memory efficient
It is easy to understand	It is tough to understand
It is simple to debug	It is complex to debug comparatively
It is simple to maintain	It is complex to maintain comparatively
It is portable	It is non-portable
It can run on any platform	It is machine-dependent
It needs compiler or interpreter for translation	It needs assembler for translation
E.g. Python	E.g. Assembly

Compiled language

executable → native, fast performance, os dependent

Apllication developed using Compiled languages are os dependent i.e. if it is developed on Windows it will work on windows only and not of linux or other os.

- A programming language which involves an executable to execute the logic instead of executing the source code directly
- E.g. C, C++, Pascal, Objective-C, Swift etc.
- Stages
  - Pre-processing
    - Used to preprocess the code
    - Comments removal
    - Code expansion
    - Conditional compilation
  - Compiling
    - Compiler is used to translate pre-processed code into assembly instructions specific to the target processor architecture
  - Assembling
    - Assembler is used to translate the assembly instructions to object code
  - Linking
    - Linker used in this stage re-arranges the code and insert missing files to emit an executable

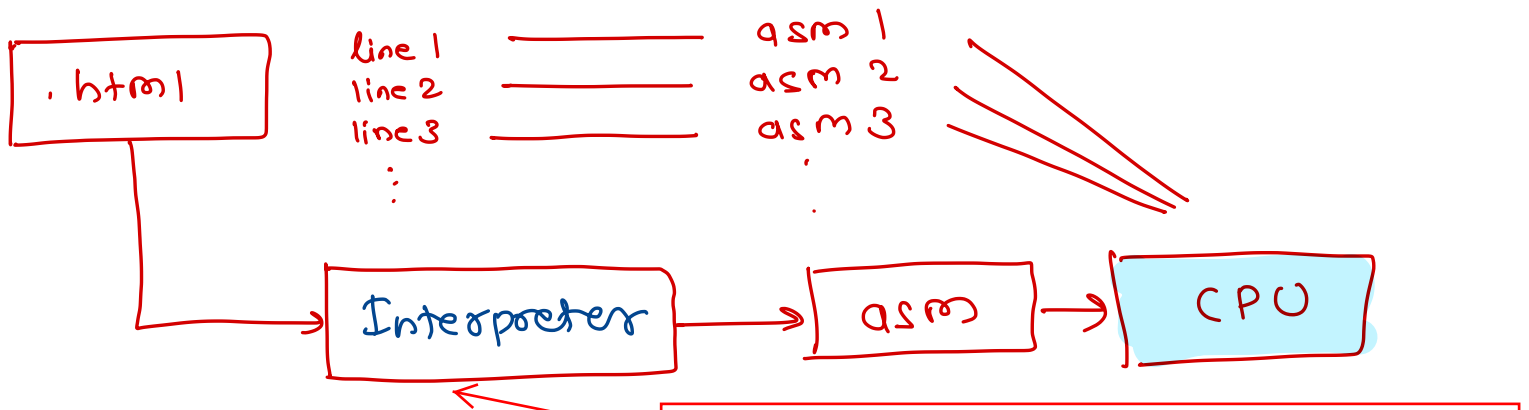


Interpreted Language : slower, OS independent

- A language mostly executes source code directly and freely, without previously compiling a programm into an executable
- E.g. Python*, JavaScript, Perl, BASIC etc. [websites]
- Interpretation
  - Interpreter used in the language executes the high level source code directly

Python is interpreted as well as compiled language

1)Application developed using Interpreted languages are os independent i.e. if it is developed on Windows it will also work on linux or other os.
2)Are slower as compared to interpreted languages.
3)All websites are written in interpreted languages as same website can be opened in any OS.



1)No executable is created in interpreted languages.
2)It executes the source code line by line and feeds to CPU

Python

Introduction

Data types are not known to the developer but to python only

Framework is collection of different libraries or functions.

- Python is a general purpose, high level and interpreted language
- It is dynamically typed and garbage collected language
- It supports programming paradigms like
  - Procedural programming
  - Object oriented programming
  - Functional programming
  - Aspect oriented programming (metaprogramming and magic metaobjects)
- Heavily dependent on indentation to create blocks which makes it very easy to read the code
- It has more than 130,000 packages included with wide range of functionality
  - Graphical user interfaces
  - Web frameworks : flask, Django
  - Networking
  - Automation
  - Web scraping : selenium

Automatic garbage collection as in C & c++ we used memory allocation methods in python its done automatically.

No curly brackets are used for syntax but tabs or spaces are used one space or two space etc.But curly braces is used for other applications.

libraries

Packages are libraries which can be used for development of various applications

History

- Python was conceived in the late 1980s by Van Guido Rossum in Netherlands
- It is considered as a successor to the ABC language (itself inspired by SETL)
- Its implementation began in December 1989
- Van Rossum continued as Python's lead developer until July 12, 2018
- When he announced his permanent vacation from his responsibilities, a team of five members was developed in Jan 2019 to lead the project

SET Theory as the language

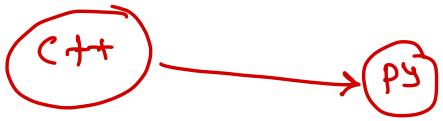
Versions

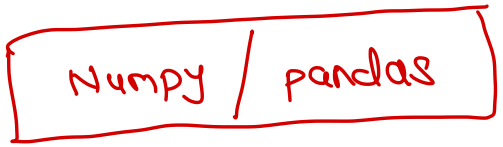
- Version 0.9.0 [Feb 1991]
  - Having features like classes with inheritance, exception handling, functions etc.
  - One of the major versions of python
- Version 1 [Jan 1994]
  - The major new features included in this release were the functional programming tools like labda, map, filter, reduce
  - The last version released was 1.6 in 2000
- Version 2 [Oct 2000]
  - Introduced features like list comprehension, garbage collection, generators etc.
  - Introduced its own license known as Python Software Foundation License (PSF)
  - The last version released was 2.7.16 in Mar 2019
- Version 3 [Dec 2008] * { Not backword compitable }
  - Python 3.0 is also called "Python 3000" or "Py3K"
  - It was designed to rectify fundamental design flaws in the language
  - Python 3.0 had an emphasis on removing duplicative constructs and modules

Old versions don't work on new versions or vice versa.

Features

- Free and open source
- Mature programming language
- Supportive user community
- Elegant design and easy to learn
- Extremely portable [pvm]
- Compiles to interpreted byte code
- Memory management using reference counting [garbage collection]
- Provides interfaces to various languages





What it is used for

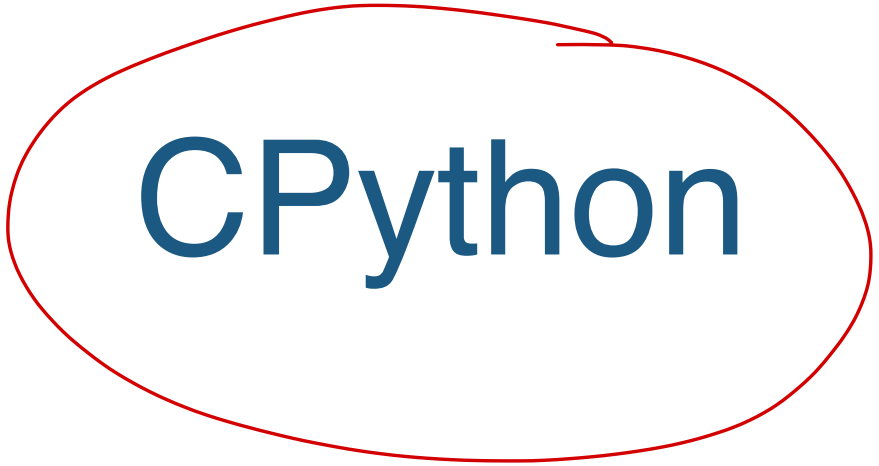
- Console application development
- GUI application development
- Web scripting : Scrapy
- Scientific application development : Scipy → numpy
- Extension languages
- XML/JSON processing : JSON, xml
- Applications heavily dependent on data : db connectors
- Machine learning : tensorflow, pytorch, sklearn
- Image processing : opencv
- Testing application
- Linux Administration

Environment Setup

- Python installation
  - Download and install from https://www.python.org/downloads/
- IDE
  - Offline
    - Visual studio code
    - Spyder
    - PyCharm
  - Online compilers

Python compilers

- Python has many versions like
  - CPython
  - Pypy
  - Jython
  - InronPython
  - Nuitka
  - Pyjs
  - RPython
- The standard version is called as CPython which is what most of us get when we download from the official site
- Henceforth when we say we are using python, it will be CPython


CPython

CPython

- CPython is the default and most widely used implementation of the language
- One of the reference implementation of Python language
- It is written in C and python
- Can be referred as both, compiler and interpreter
- It comes with a whole range of tools, libraries, and components

Semicolons are optional in python

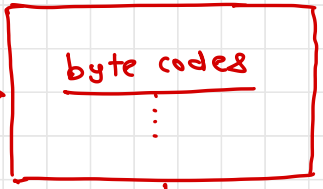
page1.py

Multiple statements can be added

cpython

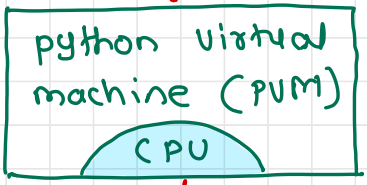
python3

compiles code

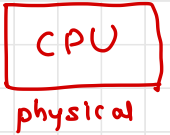


python3

interpretation



converts byte codes to asm code



physical

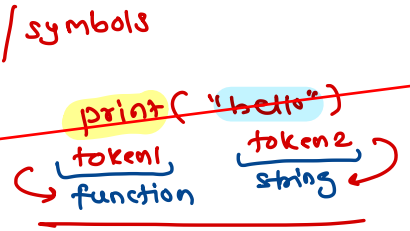
What happens in background when program is executed.
1)CPU is going to execute the code.
2)When run command is executed it calls python3 in background which is going to compile the code.
3)When code is compiled it gets converted to byte codes and it is the intermediate code between CPU and compiler.
4)Byte code is ment for PVM(Python Virtual Machine) PVM has its own virtual CPU.
5)The process in which byte code is converted to assembly code(.asm) is called as interpretation which is done by python3 therefore python is both interpretation and compiled language.
6)As the program for python3(3 is the version of python) is return in C therefore it is called as the cpython.

Python Virtual Machine (PVM)

- Written in C
- Compiles the bytecode into machine language
- It emulates the machine or CPU
- Executes bytecodes similar to the way a CPU executes the machine instructions
- Python memory manager
- Computation Stacks

How does it work ?

- Lexing
  - Breaks the lines into tokens
- Parsing
  - Parser uses the tokens generated by lexer to generate Abstract Syntax Tree (AST)
  - It depicts the relationship between the tokens
- Compiling
  - The compiler turns the AST into the code objects
- Interpreting
  - Interpreter executes generated code with the help of CPU



It understands the tokens which one is function,keyword,variable and creates a symbol table or checks if the syntax is correct and create AST.

if a token is function, string, value etc

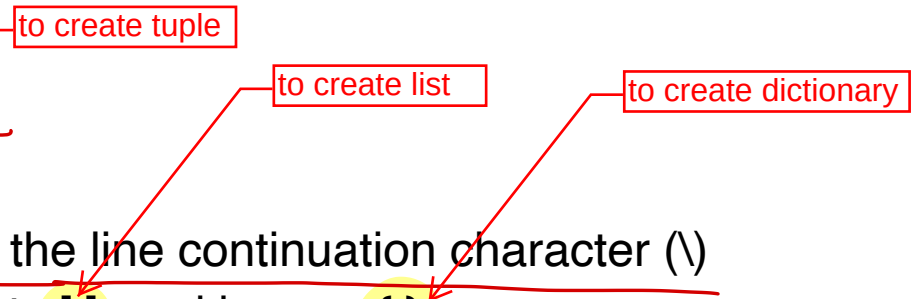
Foundation

Statements

instruction for computer [CPU]

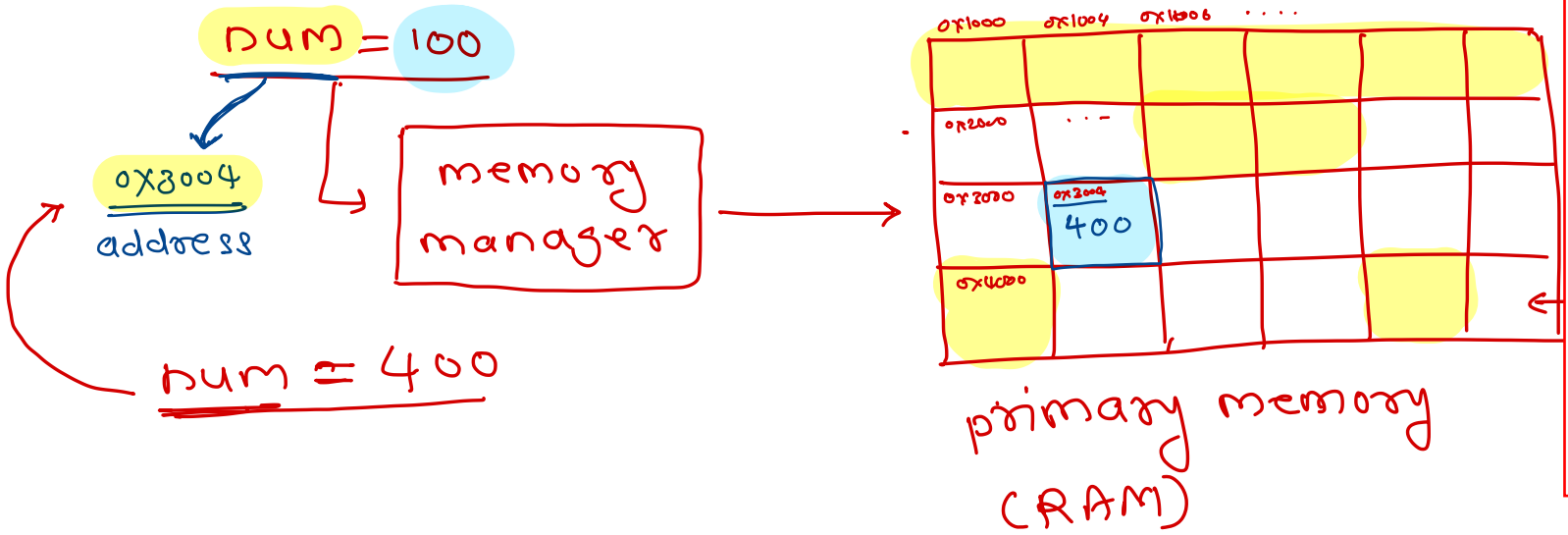
- Instructions that a Python interpreter can execute are called statements
  - E.g., a = 1 is an assignment statement

- **Single line statements**
  - The end of a statement is marked by a newline character
- **Multi-line statement**
  - We can make a statement extend over multiple lines with the line continuation character (\)
  - Line continuation is implied inside parentheses ( ), brackets [ ], and braces { }
  - We can also put multiple statements in a single line using semicolons
- **Comments** (C ignored while execution)
  - Comments are very important while writing a program
  - They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out
  - In Python, we use the hash (#) symbol to start writing a comment



Variable

- A variable is a named location used to store data in the memory
- It is helpful to think of variables as a container that holds data that can be changed later
- E.g.
  - value = 20
  - name = “steve”



1)Memory allocation is done by OS.
2)num=100 where num is for developer understanding as one cannot remember the address but OS uses the address to deal with the values.
3)When we change num=400 OS will change the value i.e. 100 will be replaced by 400 at the old location and will not allocate new memory for the as the name of variable is same.

Constants

- Constants are written in all capital letters and underscores separating the words
- E.g.
  - PI = 3.14
- In reality, we don't use constants in Python
- Naming them in all capital letters is a convention to separate them from variables, however, it does not actually prevent reassignment

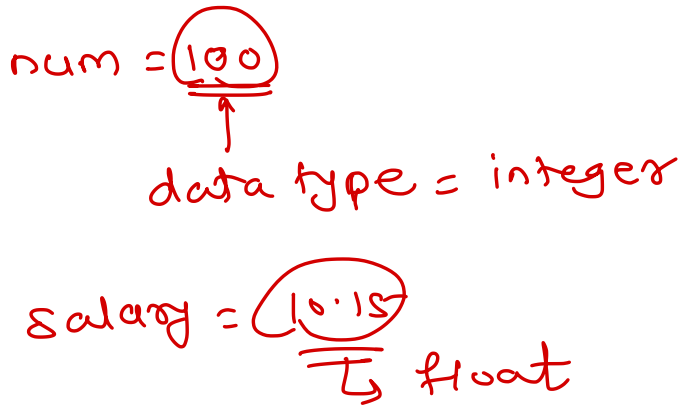
Rules and Conventions

- Constant and variable names should have a combination of letters in lowercase (**a to z**) or uppercase (**A to Z**) or digits (**0 to 9**) or an underscore (_)
- If you want to create a variable name having two words, use underscore to separate them
- Create a name that makes sense (use meaningful names)
- Use capital letters possible to declare a constant
- Never use special symbols like !, @, #, $, %, etc.
- Don't start a variable name with a digit

Data Types

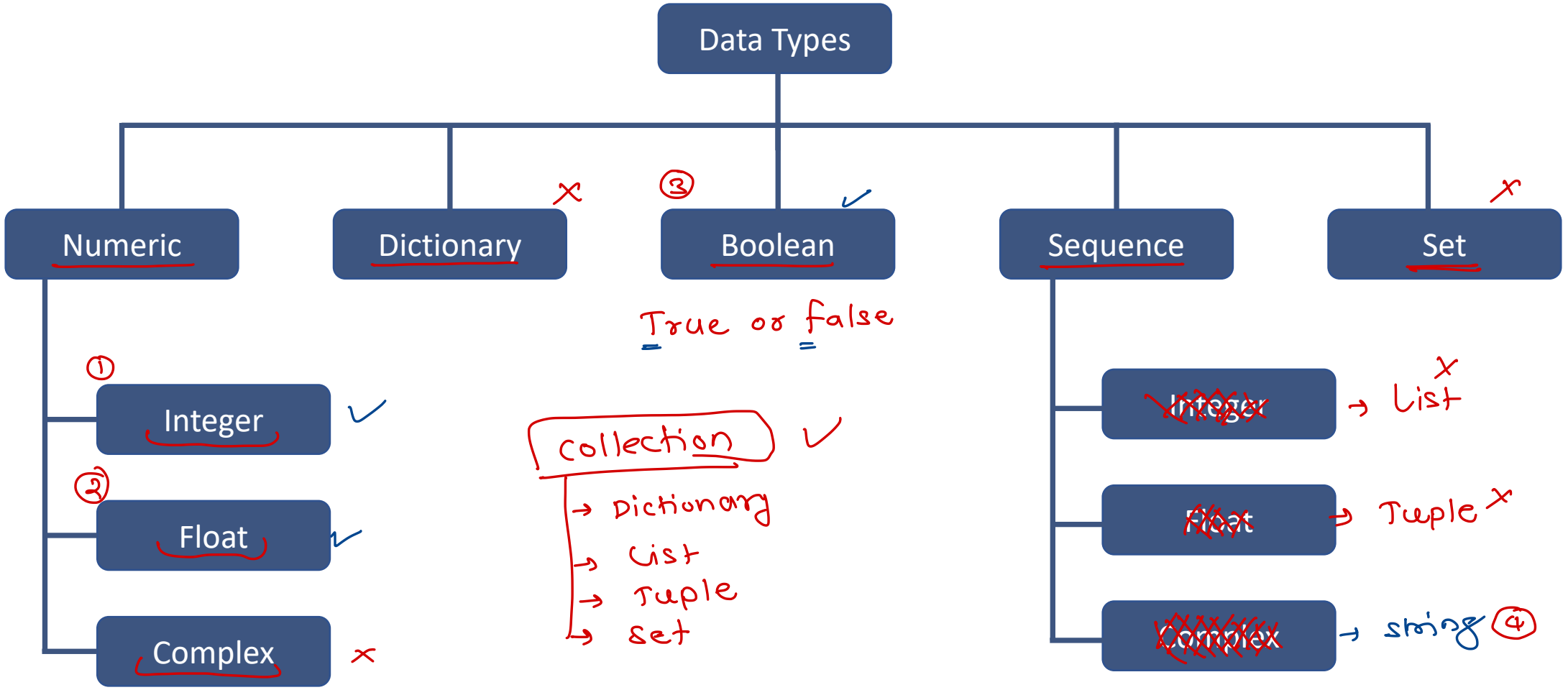
Need of Data Types

- Data type is an attribute associated with a piece of data that tells a computer system how to interpret its value
- Understanding data types ensures that data is collected in the preferred format and the value of each property is as expected
- It helps to know what kind of operations are often performed on a variable



Python Data Types

: In python, all datatypes are inferred
automatically assigned by python



Literals

- Literal is a raw data given in a variable or constant

num = 100
literal

- **Numeric Literals**
  - Numeric Literals are immutable (unchangeable)
  - Numeric literals can belong to 3 different numerical types:
    - Integer (value = 100)
    - Float (salary = 15.50)
    - Complex (x = 10 + 5j)

- **String literals**
  - A string literal is a sequence of characters surrounded by quotes
  - We can use both single, double, or triple quotes for a string
  - E.g., name = “steve” or name = ‘steve’

Literals

- **Boolean literals**
  - A Boolean literal can have any of the two values: True or False
  - E.g., can_vote = True or can_vote = False

- **Special literal**
  - Python contains one special literal i.e. None
  - We use it to specify that the field has not been created
  - E.g., value = None

- **Literal Collections**
  - Used to declare variables of collection types
  - There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals

Type Hinting

data types are dynamically assigned

- Due to the dynamic nature of Python, inferring or checking the type of an object being used is especially hard
- This fact makes it hard for developers to understand what exactly is going on in code they haven't written. As a result they resort to trying to infer the type with around 50% success rate.
- **Why type hints?**
  - **Helps type checkers:** By hinting at what type you want the object to be the type checker can easily detect if, for instance, you're passing an object with a type that isn't expected
  - **Helps with documentation:** A third person viewing your code will know what is expected where, ergo, how to use it without getting them TypeErrors
  - **Helps IDEs develop more accurate and robust tools:** Development Environments will be better suited at suggesting appropriate methods when know what type your object is

Type Conversion

- The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion
- Python has two types of type conversion.
  - Implicit Type Conversion
    - Python automatically converts one data type to another data type
    - This process doesn't need any user involvement
    - E.g.
      - result = 10 + 35.50
  - Explicit Type Conversion
    - Users convert the data type of an object to required data type
    - We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion
    - E.g.
      - num_str = str(1024)

Type Conversion

- Type Conversion is the conversion of object from one data type to another data type
- Implicit Type Conversion is automatically performed by the Python interpreter
- Python avoids the loss of data in Implicit Type Conversion
- Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user
- In Type Casting, loss of data may occur as we enforce the object to a specific data type

Operators

Operators

- Operators are special symbols in Python that carry out arithmetic or logical computation
- The value that the operator operates on is called the operand
- Types
  - Arithmetic operators
  - Comparison operators
  - Logical operators
  - Bitwise operators
  - Special operators
  - Membership operators

Arithmetic Operators

Operator	Meaning	Example
+	Add two operands or unary plus	x + y + 2
-	Subtract right operand from the left or unary minus	x – y
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y

Comparison operators

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Logical operators

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Assignment Operators

Operator	Meaning	Example
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5

Operator	Meaning	Example
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Bitwise operators

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Special operators

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Membership Operators

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Python Programming

Operators

Operators

- Operators are special symbols in Python that carry out arithmetic or logical computation
- The value that the operator operates on is called the operand
- Types
  1. Arithmetic operators *
  2. Comparison operators *
  3. Logical operators *
  - Bitwise operators –
  - Special operators – class
  - Membership operators – collection

Arithmetic Operators

Operator	Meaning	Example
+	Add two operands or unary plus	x + y + 2
-	Subtract right operand from the left or unary minus	x – y
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y

Comparison operators

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

Logical operators

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

and

T and T = T
T and F = F
F and T = F
F and F = F

or

T or T = T
T or F = T
F or T = T
F or F = F

not

not T = F
not F = T

Assignment Operators

Operator	Meaning	Example
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5

Operator	Meaning	Example
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Bitwise operators

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Special operators

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Membership Operators

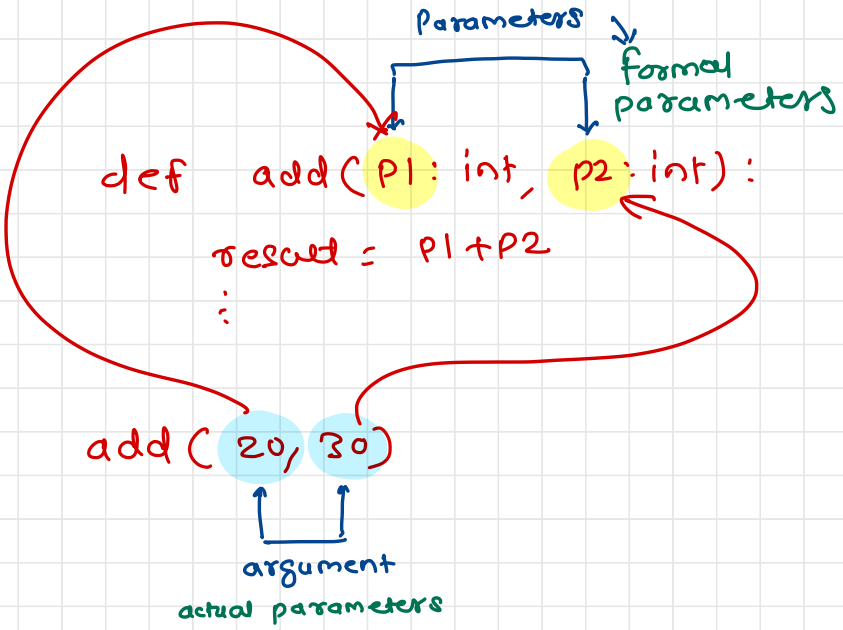
Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Functions

Functions

- In Python, a function is a group of related statements that performs a specific task
- Functions help break our program into smaller and modular chunks
- As our program grows larger and larger, functions make it more organized and manageable
- Furthermore, it avoids repetition and makes the code reusable
- Syntax

```
def function_name(parameters):
    """docstring"""
    statement(s)
```


Parameters
formal parameters
def add(p1: int, p2: int):
result = p1+p2
:
add(20, 30)
argument
actual parameters

Docstrings

- The first string after the function header is called the docstring and is short for documentation string
- It is briefly used to explain what a function does
- Although optional, documentation is a good programming practice
- We generally use triple quotes so that docstring can extend up to multiple lines
- This string is available to us as the __doc__ attribute of the function

help
- description
- logic
- parameters
- return value

Returning a value

- The return statement is used to exit a function and go back to the place from where it was called
- This statement can contain an expression that gets evaluated and the value is returned
- If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object

Function Types

- Functions can divided into following two types:
  - **Built-in functions**
    - Functions that readily come with Python are called built-in functions
    - If we use functions written by others in the form of library, it can be termed as library functions
    - E.g., str(), int(), float() etc.
  - **User defined functions**
    - Functions that we define ourselves to do certain specific task are referred as user-defined functions
    - User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug
    - If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
    - Programmers working on large project can divide the workload by making different functions

Scope of variables

- Scope of a variable is the portion of a program where the variable is recognized
- Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exists in the memory
- The lifetime of variables inside a function is as long as the function executes
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Global Scope

- In Python, a variable declared outside of the function or in global scope is known as a global variable
- This means that a global variable can be accessed inside or outside of the function

```
g_var = "global"
def foo():
    print("inside foo")
    print(g_var)

foo()
```

Local Scope

- A variable declared inside the function's body or in the local scope is known as a local variable

```
def foo():
    local_var = "local"

foo()
# error
print(local_var)
```

inside the function

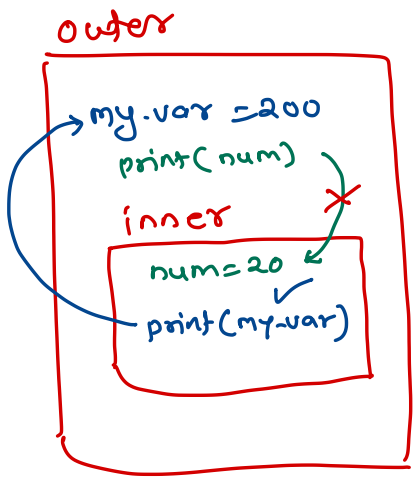
Global Keyword

- 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**
  - When we create a variable inside a function, it is local by default
  - When we define a variable outside of a function, it is 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

Nested Function

- Function within a function is called as nested function or inner function
- E.g.

```
def outer():
    print("inside outer")
    def inner():
        print("inside inner")
    inner()

outer()
# error
inner()
```



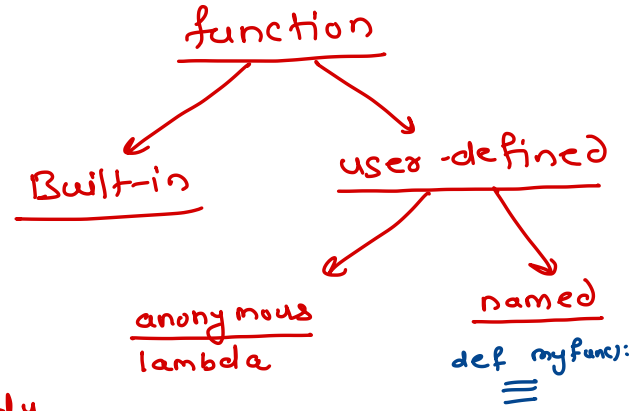
Anonymous/Lambda Function

- In Python, an anonymous function is a function that is defined without a name
- While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword
- Hence, anonymous functions are also called lambda functions
- Syntax
  - lambda arguments: expression → evaluated
- Characteristics
  - It can only contain expressions and can't include statements in its body
  - It is written as a single line of execution
  - It does not support type annotations
  - It can be immediately invoked (self execution)

lambda cannot have multi-line body

no type hinitng is allowed

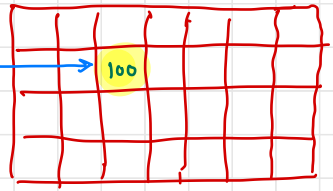
- lambda can not be parameterless

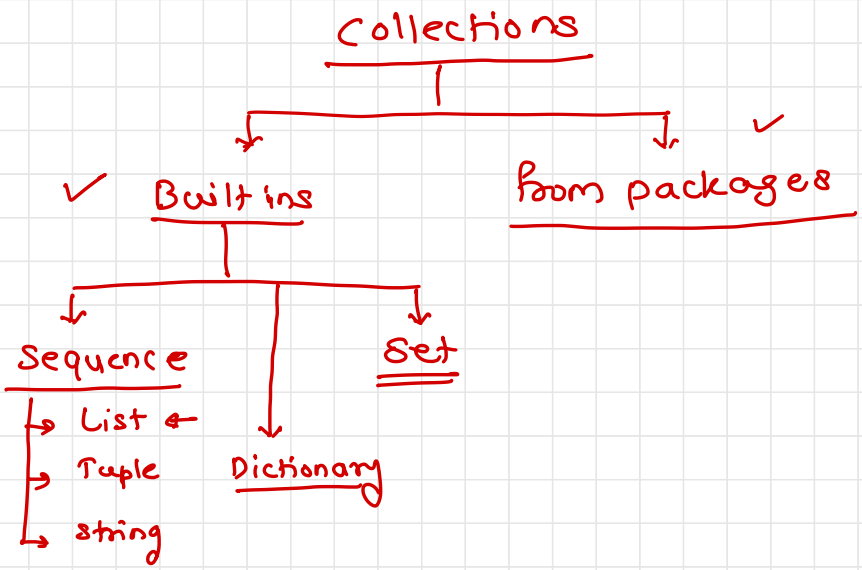


Python Programming

Collections

num = 100


100


Collections
Built ins
from packages
Sequence
Set
List
Tuple
string
Dictionary

List

- Python lists are one of the most versatile data types that allow us to work with multiple elements at once
- For example
  - # a list of programming languages
  - ['Python', 'C++', 'JavaScript']
- List is created by placing elements inside square brackets [], separated by commas
- List is **mutable**

List - functions

- append()
  - adds an element to the end of the list
- extend()
  - adds all elements of a list to another list
- insert()
  - inserts an item at the defined index
- remove()
  - removes an item from the list
- pop()
  - returns and removes an element at the given index
- clear()
  - removes all items from the list
- index()
  - returns the index of the first matched item
- count()
  - returns the count of the number of items passed as an argument
- sort()
  - sort items in a list in ascending order
- reverse()
  - reverse the order of items in the list
- copy()
  - returns a shallow copy of the list

Accessing List Members

- We can use the index operator [] to access an item in a list
- In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4
- Trying to access indexes other than these will raise an **IndexError**
- The index must be an integer. We can't use float or other types, this will result in **TypeError**
- **Negative indexing**
  - Python allows negative indexing for its sequences
  - The index of -1 refers to the last item, -2 to the second last item and so on

List Slicing

- We can access a range of items in a list by using the slicing operator

[start : stop : increment-value]

```
my_list = ['p','r','o','g','r','a','m','i','z']

# elements from index 2 to index 4
print(my_list[2:5])

# elements from index 5 to end
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

Tuple

- A tuple is a collection of objects which ordered and immutable
- Tuples are sequences, just like lists
- The differences between tuples and lists are
  - Tuples cannot be changed unlike lists
  - Tuples use parentheses, whereas lists use square brackets

[once created you cannot modify tuple]

Tuple Operations

- Creating Tuples
- Concatenation of Tuples
- Nesting of Tuples (Tuple of Tuples)
- Repetition of Tuples
- Packing and Unpacking
- Indexing in Tuple
- Slicing in Tuple

} same as list

Set

- A Python set is the collection of the unordered items
- Each element in the set must be unique
- Sets are mutable which means we can modify it after its creation
- Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index
- However, we can print them all together, or we can get the list of elements by looping through the set

Set Operations

- Union of two Sets
- Intersection of two sets
- Difference between the two sets
- Frozenset
- Symmetric Difference of two sets

Dictionary

- Python Dictionary is used to store the data in a key-value pair format
- The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key
- It is the mutable data-structure
- The dictionary is defined into element Keys and values
- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.
- In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object

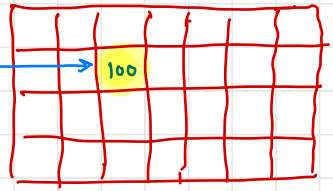
Dictionary Operations

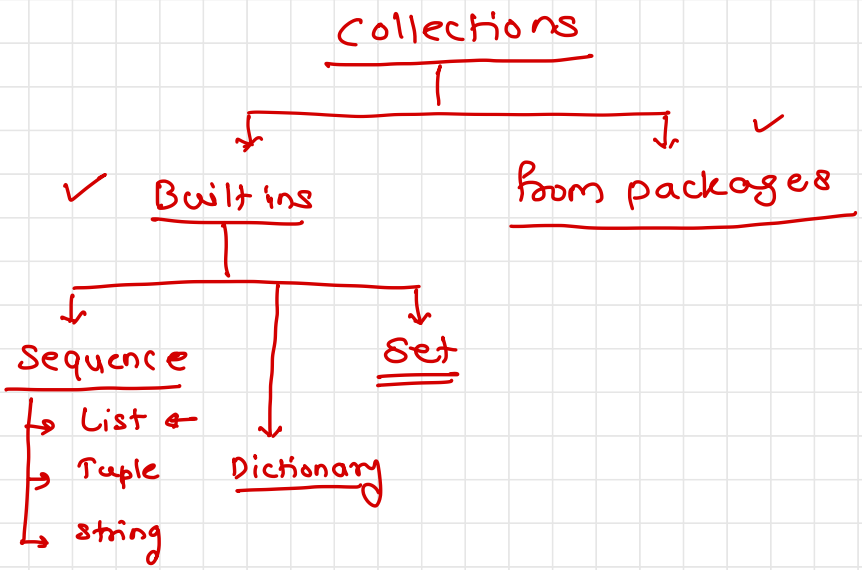
- Creating the dictionary
- Accessing the dictionary values
- Adding dictionary values
- Deleting elements using del keyword

Python Programming

Collections

num = 100


100


Collections
Built ins
from packages
Sequence
Set
List
Tuple
string
Dictionary

List

- Python lists are one of the most versatile data types that allow us to work with multiple elements at once
- For example
  - # a list of programming languages
  - ['Python', 'C++', 'JavaScript']
- List is created by placing elements inside square brackets [], separated by commas
- List is **mutable**

List - functions

- append()
  - adds an element to the end of the list
- extend()
  - adds all elements of a list to another list
- insert()
  - inserts an item at the defined index
- remove()
  - removes an item from the list
- pop()
  - returns and removes an element at the given index
- clear()
  - removes all items from the list
- index()
  - returns the index of the first matched item
- count()
  - returns the count of the number of items passed as an argument
- sort()
  - sort items in a list in ascending order
- reverse()
  - reverse the order of items in the list
- copy()
  - returns a shallow copy of the list

Accessing List Members

- We can use the index operator [] to access an item in a list
- In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4
- Trying to access indexes other than these will raise an **IndexError**
- The index must be an integer. We can't use float or other types, this will result in **TypeError**
- **Negative indexing**
  - Python allows negative indexing for its sequences
  - The index of -1 refers to the last item, -2 to the second last item and so on

List Slicing

- We can access a range of items in a list by using the slicing operator

[start : stop : increment-value]

```
my_list = ['p','r','o','g','r','a','m','i','z']

# elements from index 2 to index 4
print(my_list[2:5])

# elements from index 5 to end
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

Tuple

- A tuple is a collection of objects which ordered and immutable
- Tuples are sequences, just like lists
- The differences between tuples and lists are
  - Tuples cannot be changed unlike lists
  - Tuples use parentheses, whereas lists use square brackets

[once created you cannot modify tuple]

Tuple Operations

- Creating Tuples
- Concatenation of Tuples
- Nesting of Tuples (Tuple of Tuples)
- Repetition of Tuples
- Packing and Unpacking
- Indexing in Tuple
- Slicing in Tuple

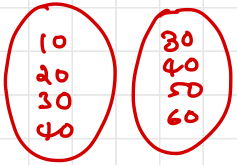
} same as list

Set

- A Python set is the collection of the unordered items
- Each element in the set must be unique
- Sets are mutable which means we can modify it after its creation
- Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index
- However, we can print them all together, or we can get the list of elements by looping through the set

Set Operations

- Union of two Sets
- Intersection of two sets
- Difference between the two sets
- Frozenset
- Symmetric Difference of two sets



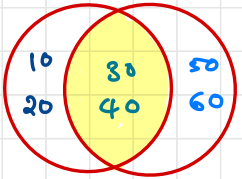
S1 S2

① union

S1 ∪ S2

{ 10, 20, 30, 40, 50, 60 }

② intersection



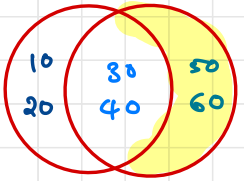
S1 ∩ S2

{ 30, 40 }

③ Difference

S1 - S2

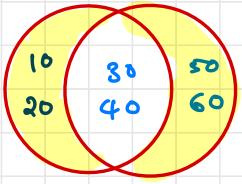
{ 10, 20 }



S2 - S1

{ 50, 60 }

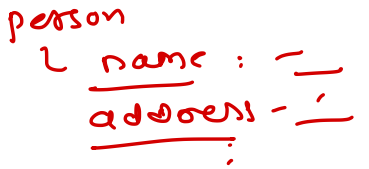
④ symmetric difference



{ 10, 20, 50, 60 }

Dictionary

- Python Dictionary is used to store the data in a key-value pair format
- The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key
- It is the mutable data-structure
- The dictionary is defined into element Keys and values
- Keys must be a single element
- Value can be any type such as list, tuple, integer, etc.
- In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object



Dictionary Operations

- Creating the dictionary
- Accessing the dictionary values
- Adding dictionary values
- Deleting elements using del keyword

Python Programming

python

- scripting
- procedure oriented language
  - function
- object oriented language ←
- functional programming
- aspect programming

Object Oriented Progamming

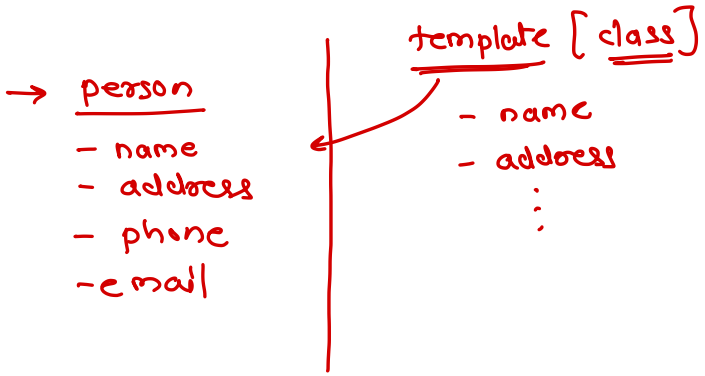
Everything in python is an object

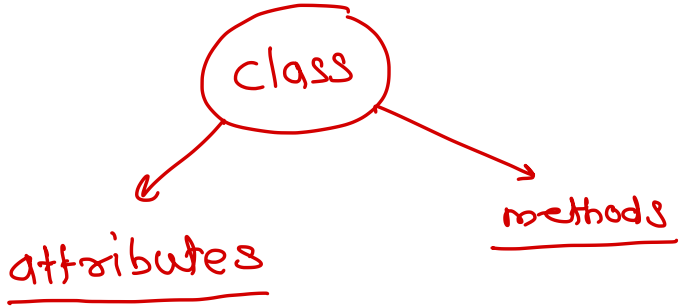
Introduction

- In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming
- It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming
- The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data
- Main concepts
  - Class
  - Objects
  - Polymorphism
  - Encapsulation
  - Inheritance

Class

- A class is a collection of objects (attributes)
- A class contains the blueprints or the prototype from which the objects are being created
- It is a logical entity that contains some attributes and methods






```
class Person :
    pass
```

Empty class

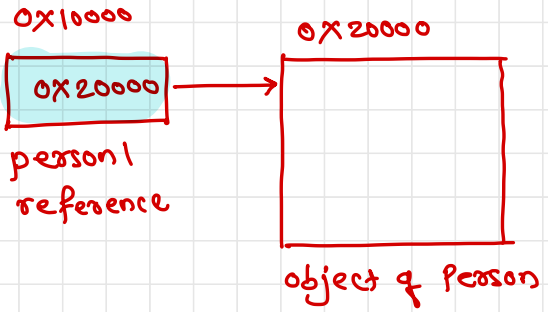
Object

- The object is an entity that has a state and behavior associated with it
- It may be any real-world object like a mouse, keyboard, chair, table, pen, etc.
- Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects
- More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on.
- **An object consists of :**
  - **State**
    - It is represented by the attributes of an object
    - It also reflects the properties of an object
  - **Behavior**
    - It is represented by the methods of an object
    - It also reflects the response of an object to other objects
  - **Identity**
    - It gives a unique name to an object and enables one object to interact with other objects

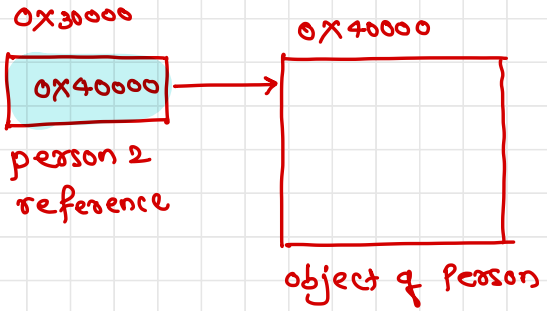
```
class Person:
    pass
```

person1 = Person()

reference — allocate new memory



person2 = Person()

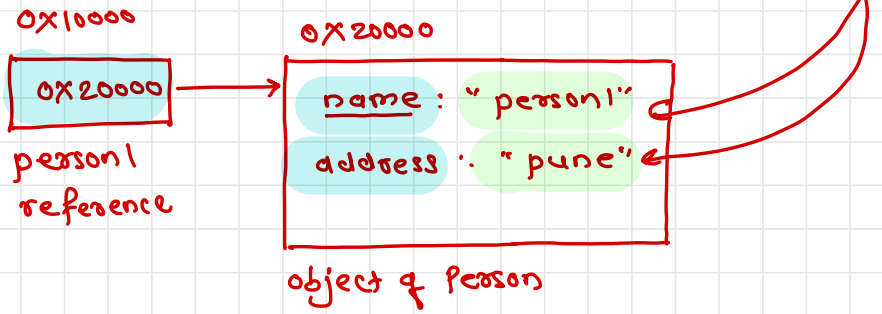


person1 = Person()

reference allocate new memory

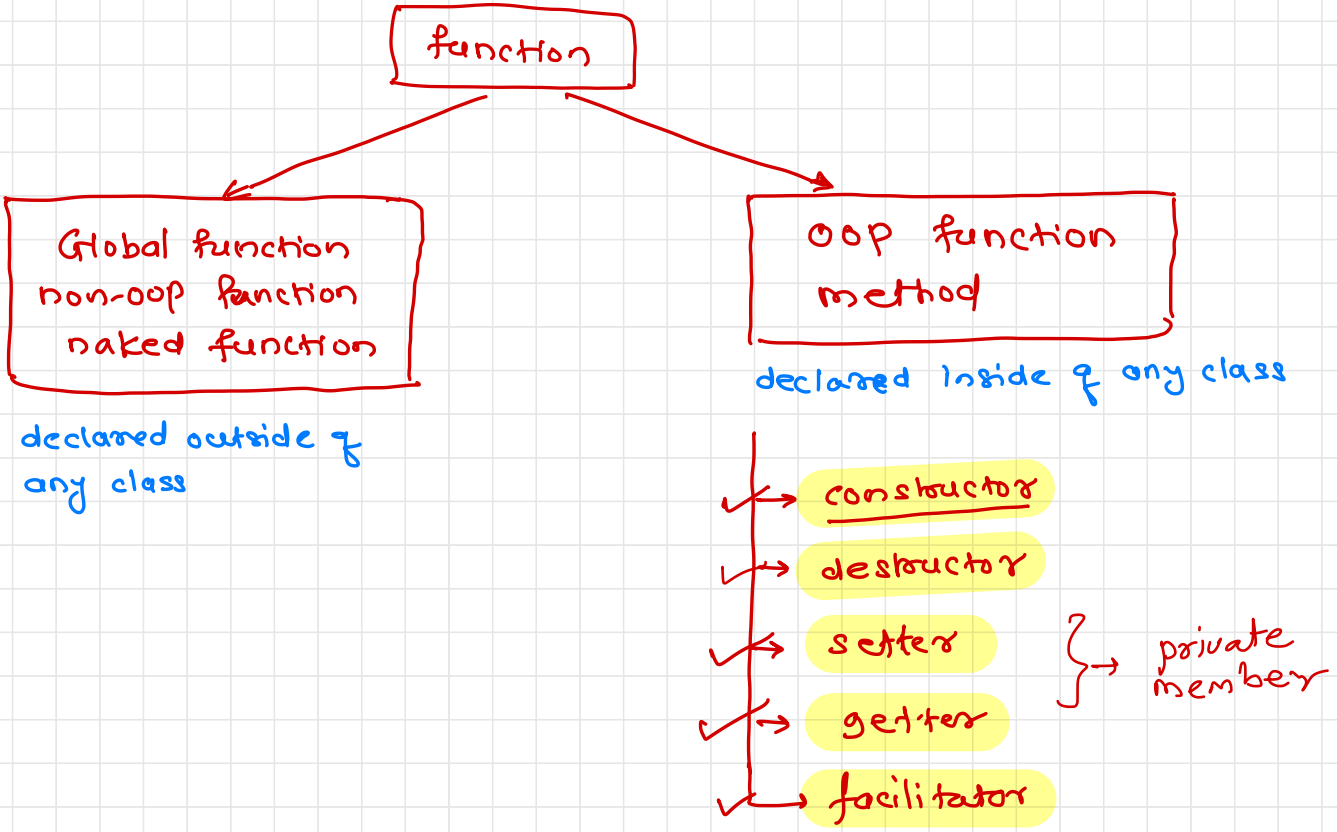
setattr (person1, "name", "person1")

setattr (person1, "address", "pune")



Methods

- Functions defined by a class are called as methods
- Methods can be of following types
  - Initializer
  - Deinitializer (del)
  - Facilitators
  - Setters
  - Getters


function
Global function
non-oop function
naked function
declared outside of any class
OOP function
method
declared inside of any class
constructor
destructor
setter
getter
facilitator
private member

```
class Person:

    # initializer
    def __init__(self):
        print("inside __init__")
```

similar to "this" in c++ and Java

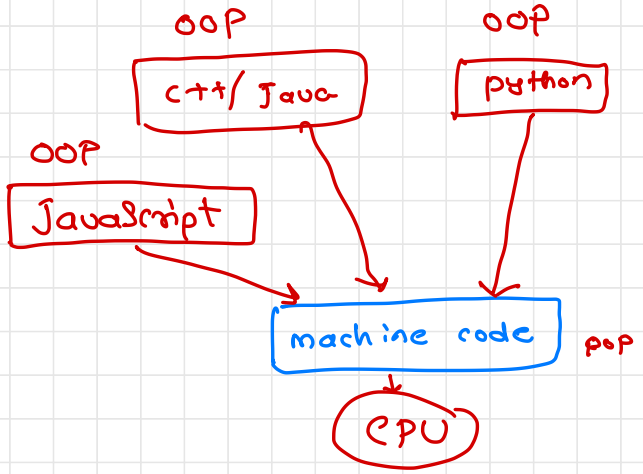
person1 = Person()

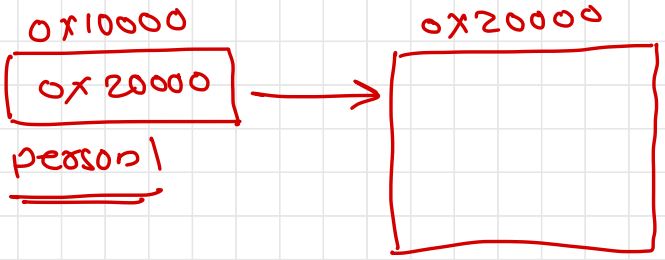
allocate memory (0x20000)

initialize object at 0x20000

person1.__init__()

Person.__init__(person1)





Inheritance

is-a relationship

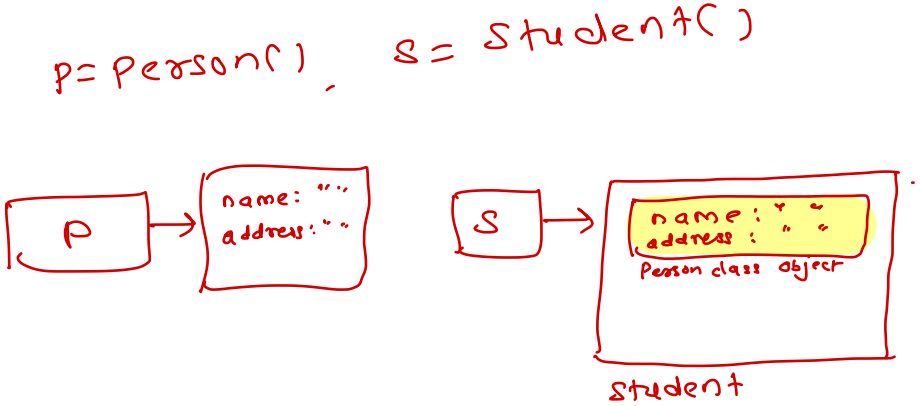
Inheritance

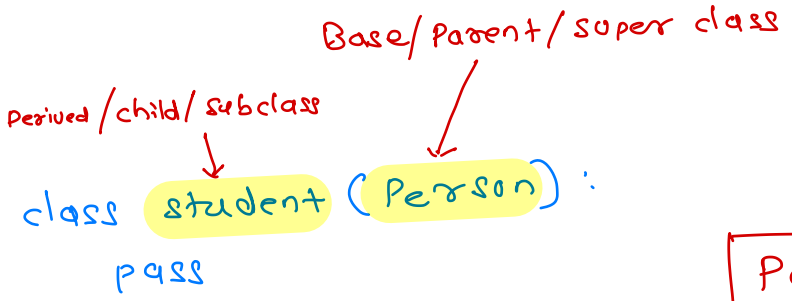
- Inheritance is a powerful feature in object oriented programming
- Inheritance is the capability of one class to derive or inherit the properties from another class
- The benefits of inheritance are:
  - It represents real-world relationships well
  - It provides **reusability** of a code. We don't have to write the same code again and again
  - It allows us to add more features to a class without modifying it
  - It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A
- Types
  - Single inheritance
  - Multilevel inheritance
  - Multiple inheritance
  - Hierarchical inheritance
  - Hybrid inheritance

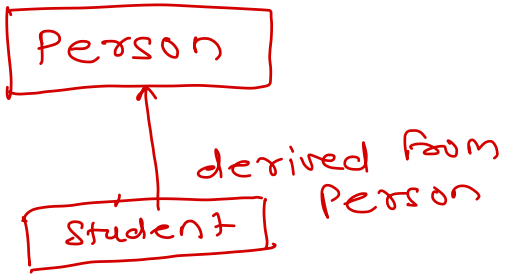
Single Inheritance

- When a child class inherits from only one parent class, it is called single inheritance
- It involves only one parent and one child class

```
class Person:
    pass
```

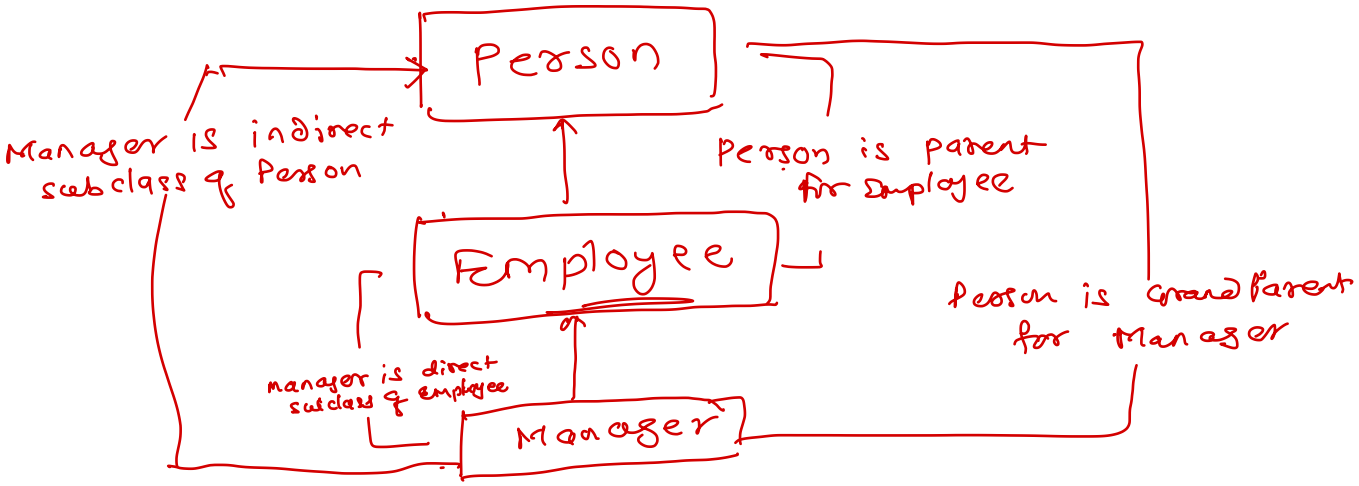


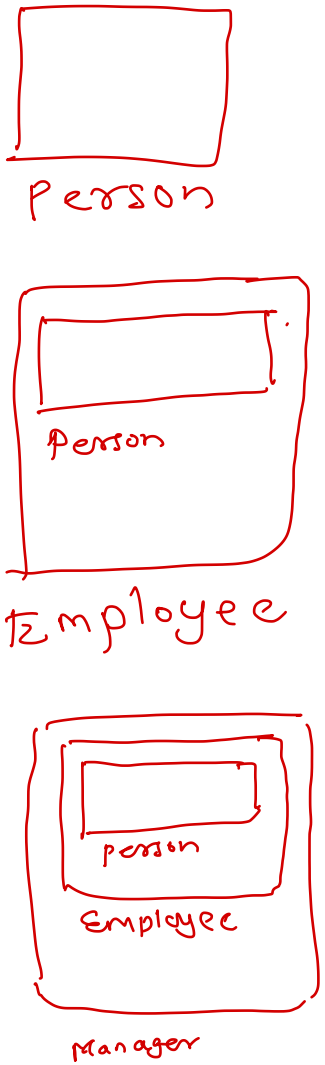




Multilevel inheritance

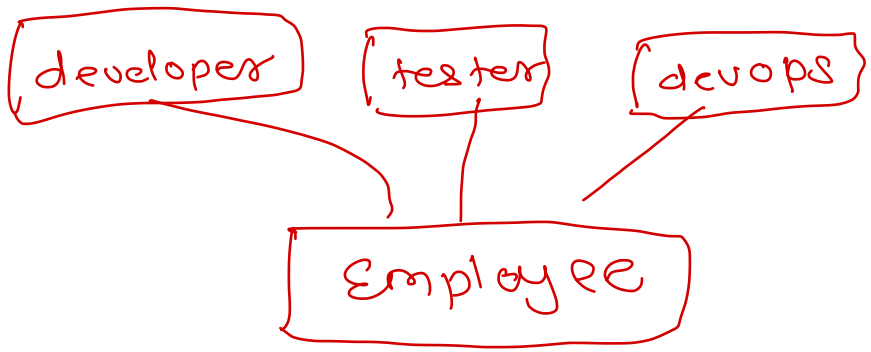
- When we have a child and grandchild relationship

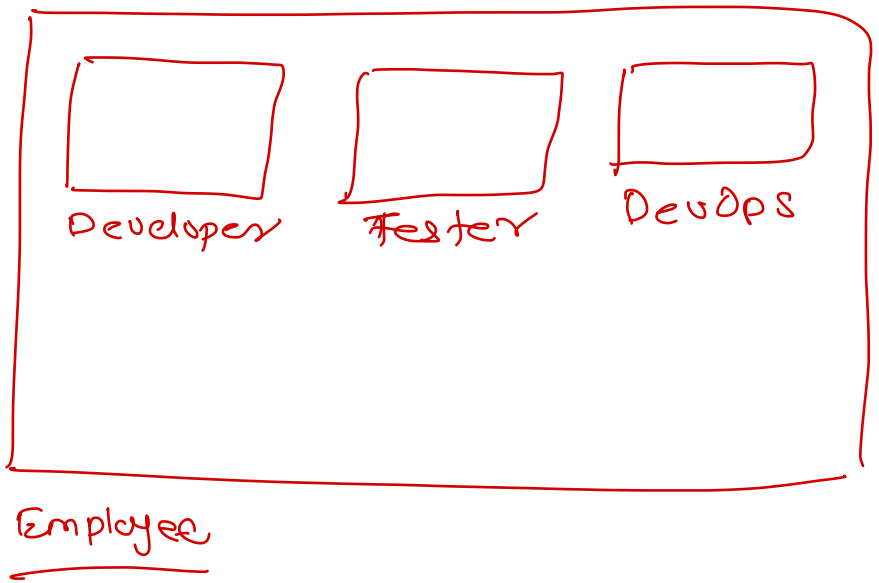




Multiple inheritance

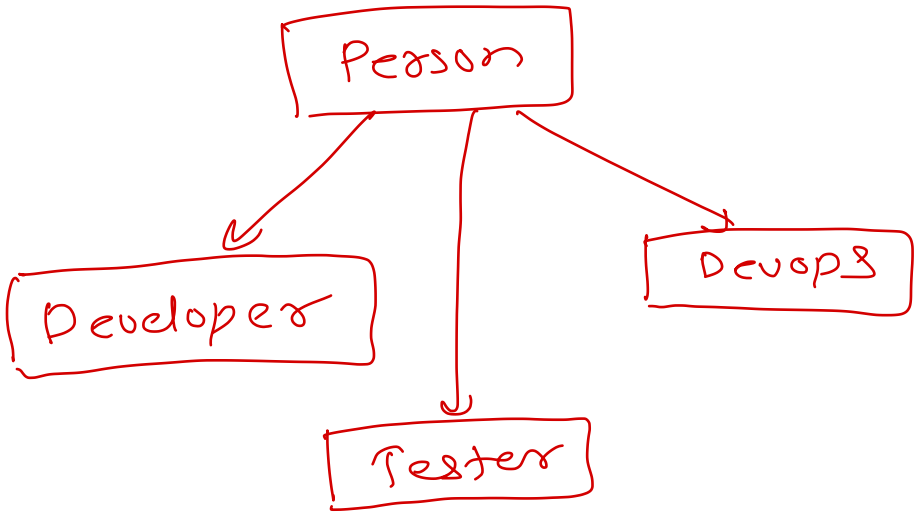
- When a child class inherits from multiple parent classes, it is called multiple inheritance

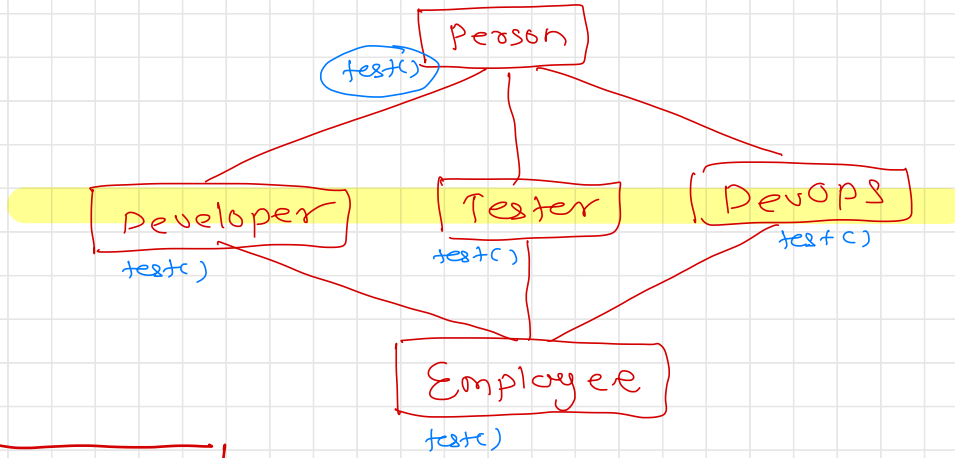


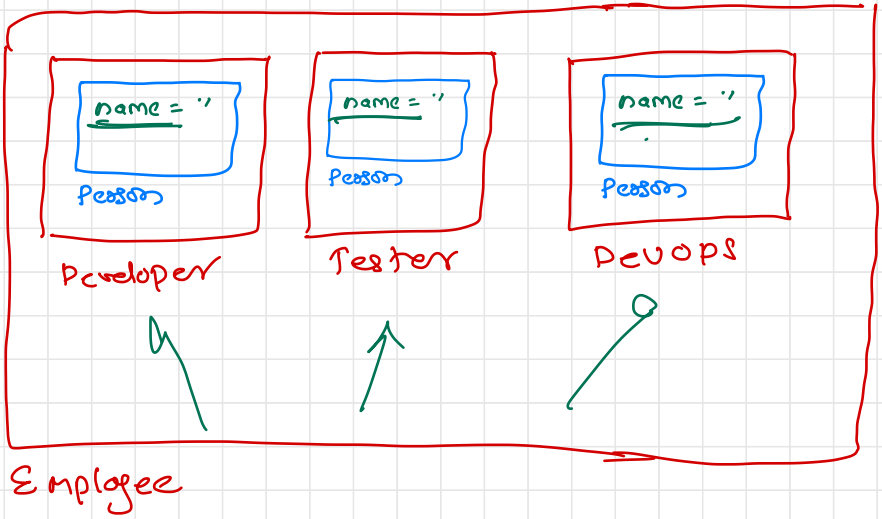


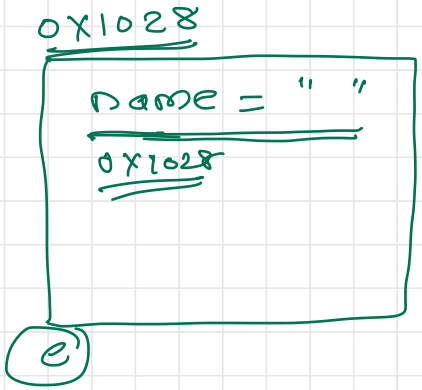
Hierarchical inheritance

- More than one derived classes are created from a single base




Person
test()
Developer
test()
Tester
test()
DevOps
test()
Employee
test()


name = ''
Person
name = ''
Person
name = ''
Person
Developer
Tester
DevOps
Employee


0X1028
name = ''
0X1028
e

Hybrid inheritance

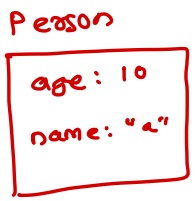
- This form combines more than one form of inheritance
- Basically, it is a blend of more than one type of inheritance

Encapsulation & Abstraction

access specifier
modifiers

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP)
- It describes the idea of wrapping data and the methods that work on data within one unit → class → object
- This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data
- To prevent accidental change, an object's variable can only be changed by an object's method



Protected members

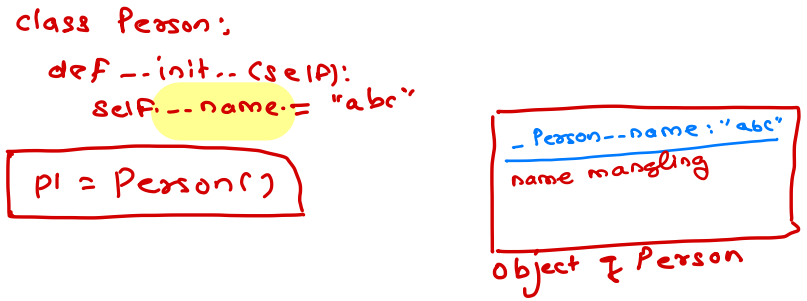
- Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses
- To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**
- Although the protected variable can be accessed out of the class as well as in the derived class(modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body

Private members [convention]

- Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class
- There is no existence of **Private** instance variables that cannot be accessed except inside a class
- However, to define a private member prefix the member name with double underscore "__"
- **Note:** Python's private and protected member can be accessed outside the class through Python Name Mangling

Name Mangling

- In Python, there is something called name mangling, which means that there is limited support for a valid use-case for class-private members basically to avoid name clashes of names with names defined by subclasses
- Any identifier of the form __member (at least two leading underscores or at most one trailing underscore) is replaced with _classname__,member, where classname is the current class name with a leading underscore(s) stripped
- As long as it occurs within the definition of the class, this mangling is done
- This is helpful for letting subclasses override methods without breaking intraclass method calls



Python Programming

python

- scripting
- procedure oriented language
  - function
- object oriented language ←
- functional programming
- aspect programming

Object Oriented Progamming

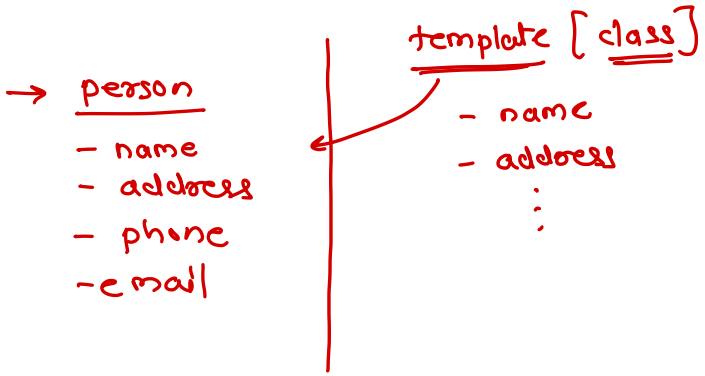
Everything in python is an object

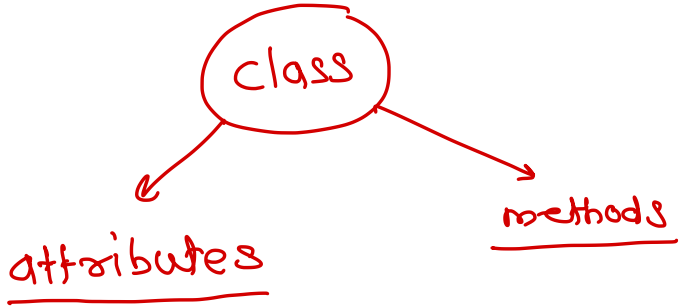
Introduction

- In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming
- It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming
- The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data
- Main concepts
  - Class
  - Objects
  - Polymorphism
  - Encapsulation
  - Inheritance

Class

- A class is a collection of objects (attributes)
- A class contains the blueprints or the prototype from which the objects are being created
- It is a logical entity that contains some attributes and methods






```
class Person :
    pass
```

Empty class

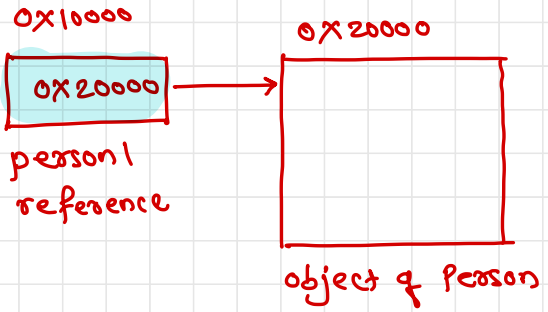
Object

- The object is an entity that has a state and behavior associated with it
- It may be any real-world object like a mouse, keyboard, chair, table, pen, etc.
- Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects
- More specifically, any single integer or any single string is an object. The number 12 is an object, the string “Hello, world” is an object, a list is an object that can hold other objects, and so on.
- **An object consists of :**
  - **State**
    - It is represented by the attributes of an object
    - It also reflects the properties of an object
  - **Behavior**
    - It is represented by the methods of an object
    - It also reflects the response of an object to other objects
  - **Identity**
    - It gives a unique name to an object and enables one object to interact with other objects

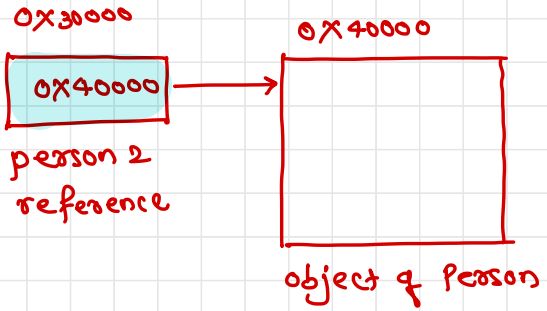
```
class Person:
    pass
```

person1 = Person()

reference — allocate new memory



person2 = Person()

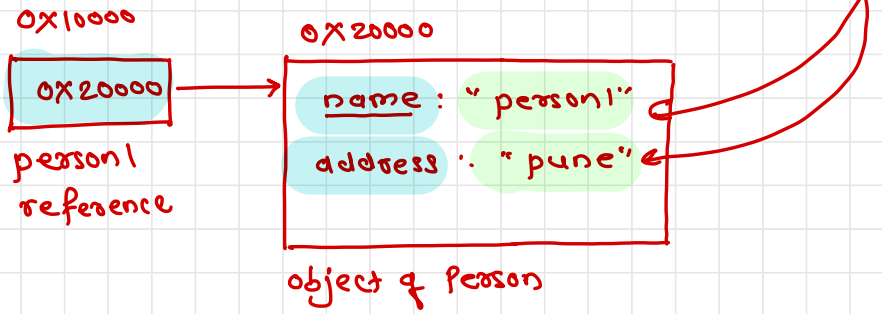


person1 = Person()

reference allocate new memory

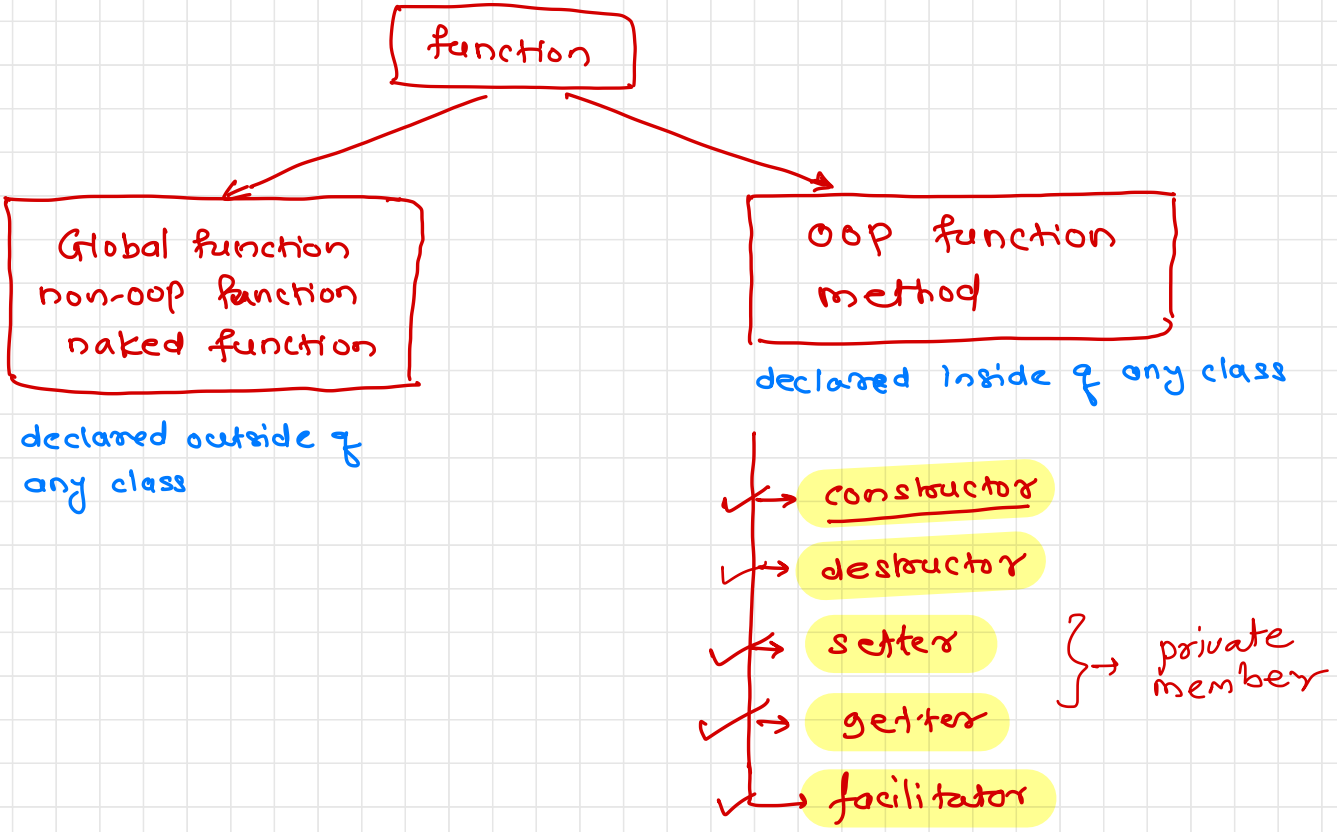
setattr (person1, "name", "person1")

setattr (person1, "address", "pune")



Methods

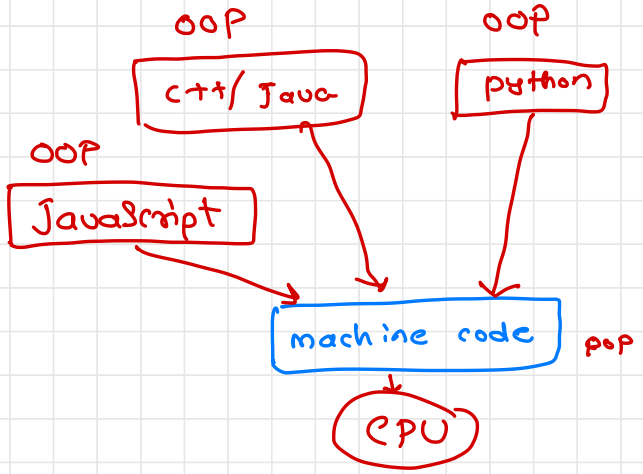
- Functions defined by a class are called as methods
- Methods can be of following types
  - Initializer
  - Deinitializer (del)
  - Facilitators
  - Setters
  - Getters

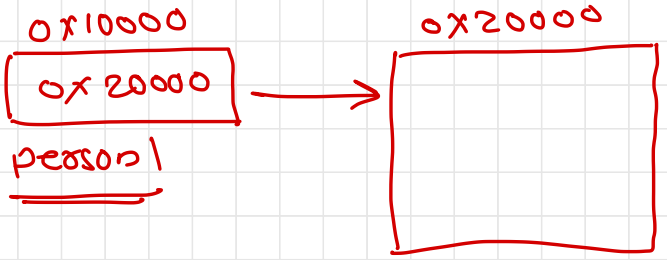

function
Global function
non-oop function
naked function
declared outside of any class
OOP function
method
declared inside of any class
constructor
destructor
setter
getter
facilitator
private member

```
class Person:

    # initializer
    def __init__(self):
        print("inside __init__")
```

similar to "this" in c++ and Java

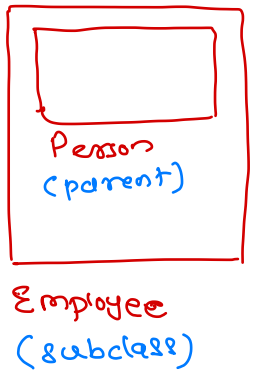
```
person1 = Person()

# allocate memory (0x20000)

# initialize object at 0x20000

# person1.__init__()

# Person.__init__(person1)
```





Student is-a Person

Car is-a vehicle



Inheritance

is-a relationship

Inheritance

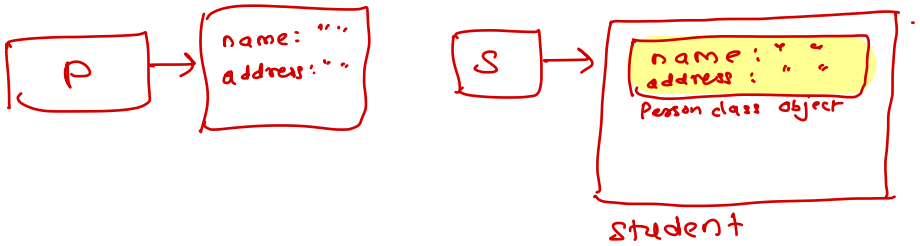
- Inheritance is a powerful feature in object oriented programming
- Inheritance is the capability of one class to derive or inherit the properties from another class
- The benefits of inheritance are:
  - It represents real-world relationships well
  - It provides **reusability** of a code. We don't have to write the same code again and again
  - It allows us to add more features to a class without modifying it
  - It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A
- Types
  - Single inheritance
  - Multilevel inheritance
  - Multiple inheritance
  - Hierarchical inheritance
  - Hybrid inheritance

Single Inheritance

- When a child class inherits from only one parent class, it is called single inheritance
- It involves only one parent and one child class

```
class Person:
    pass
```

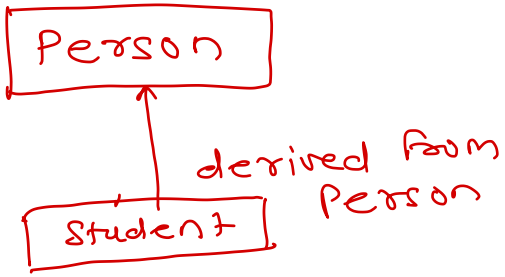
p = Person() s = Student()



Base/Parent/super class

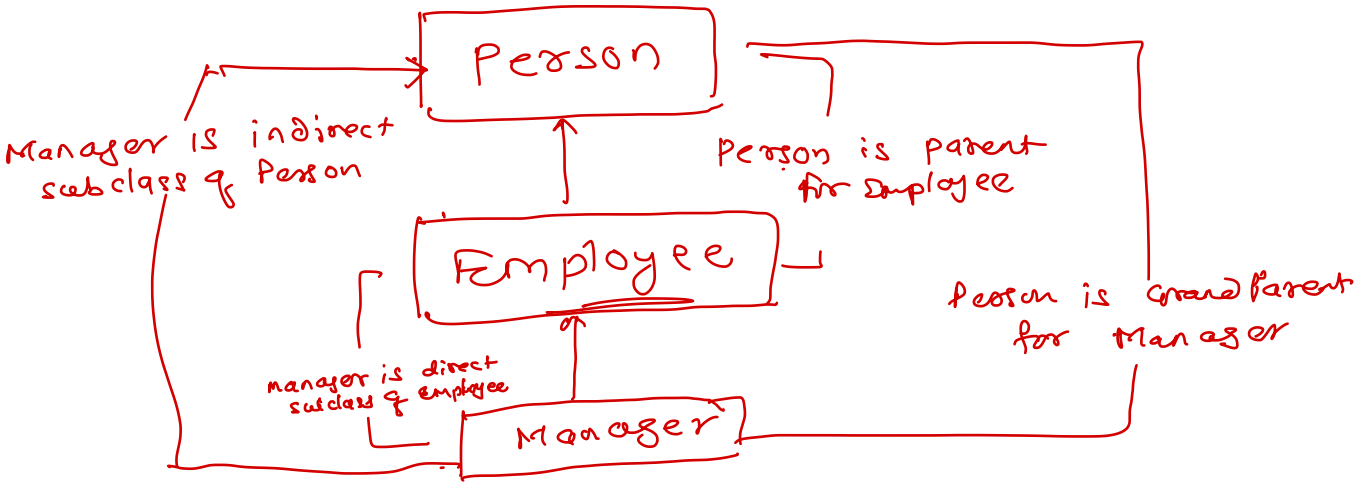
Derived/child/subclass

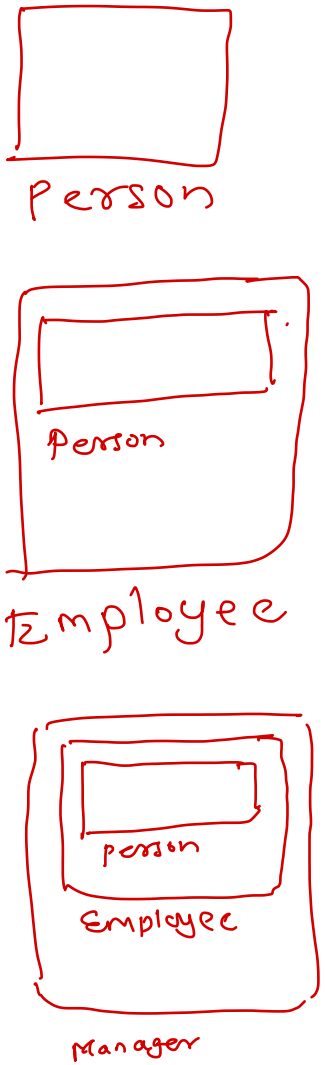
```
class student(Person):
    pass
```



Multilevel inheritance

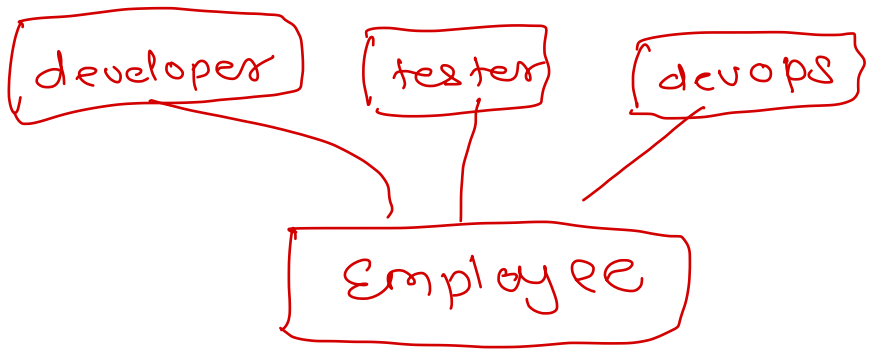
- When we have a child and grandchild relationship

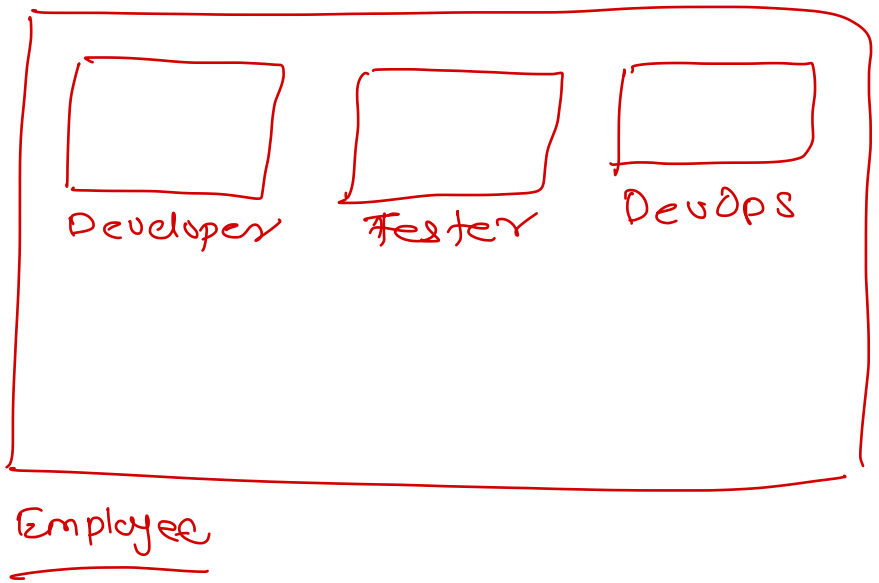




Multiple inheritance

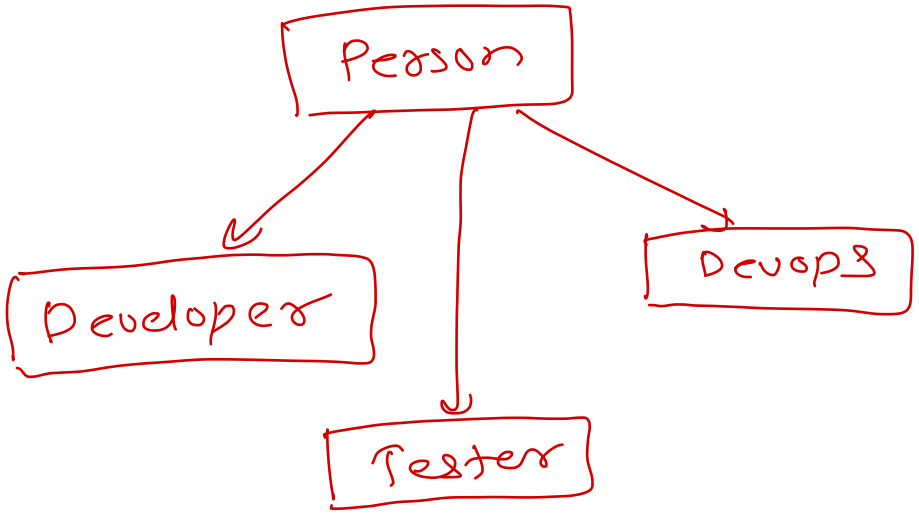
- When a child class inherits from multiple parent classes, it is called multiple inheritance

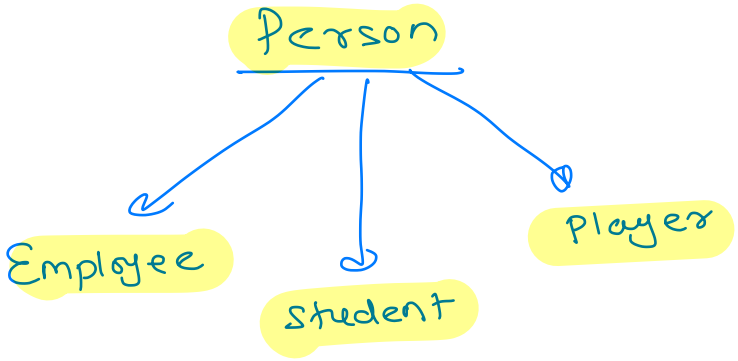


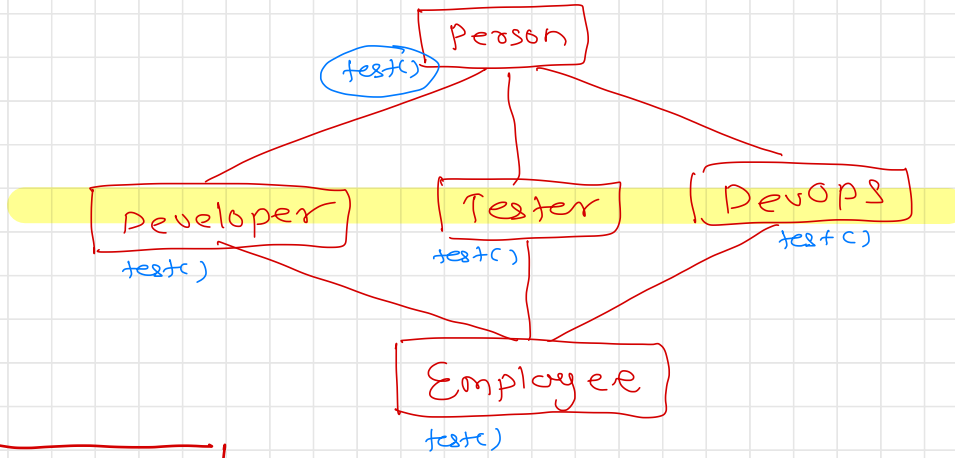


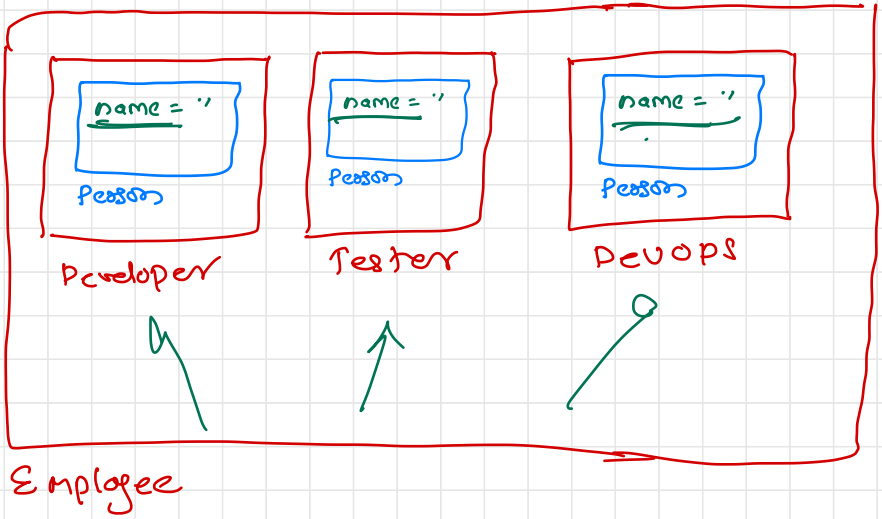
Hierarchical inheritance

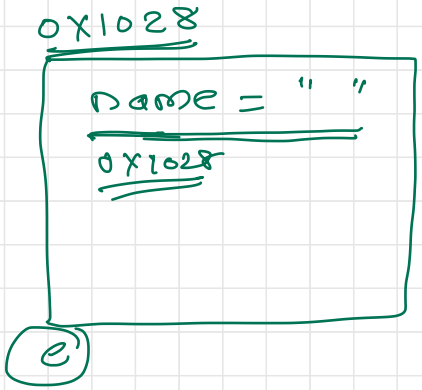
- More than one derived classes are created from a single base






Person
test()
Developer
test()
Tester
test()
DevOps
test()
Employee
test()


name = ''
Person
Developer
name = ''
Person
Tester
name = ''
Person
DevOps
Employee


0X1028
name = ''
0X1028
e

Hybrid inheritance

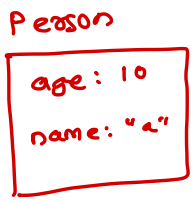
- This form combines more than one form of inheritance
- Basically, it is a blend of more than one type of inheritance

Encapsulation & Abstraction

access specifier
modifiers

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP)
- It describes the idea of wrapping data and the methods that work on data within one unit → class → object
- This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data
- To prevent accidental change, an object's variable can only be changed by an object's method



access specifiers

conventions

① public

- no underscore
- can be accessed outside of class

```
class Person:
    def __init__():
        self.name = ''
```

② protected

- single underscore
- can be accessed
  - in same class
  - in the child class(es)

```
class Person:
    def __init__():
        self._name = ''
```

③ private

- two underscores
- can be accessed only within the same class
- can not be accessed outside of the class

```
class Person:
    def __init__():
        self.__name =
```

Protected members

- Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses
- To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**
- Although the protected variable can be accessed out of the class as well as in the derived class(modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body

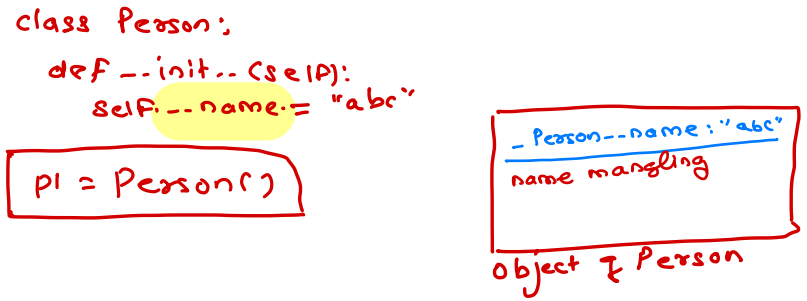
Private members

[convention]

- Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class
- There is no existence of **Private** instance variables that cannot be accessed except inside a class
- However, to define a private member prefix the member name with double underscore "__"
- **Note:** Python's private and protected member can be accessed outside the class through Python Name Mangling

Name Mangling

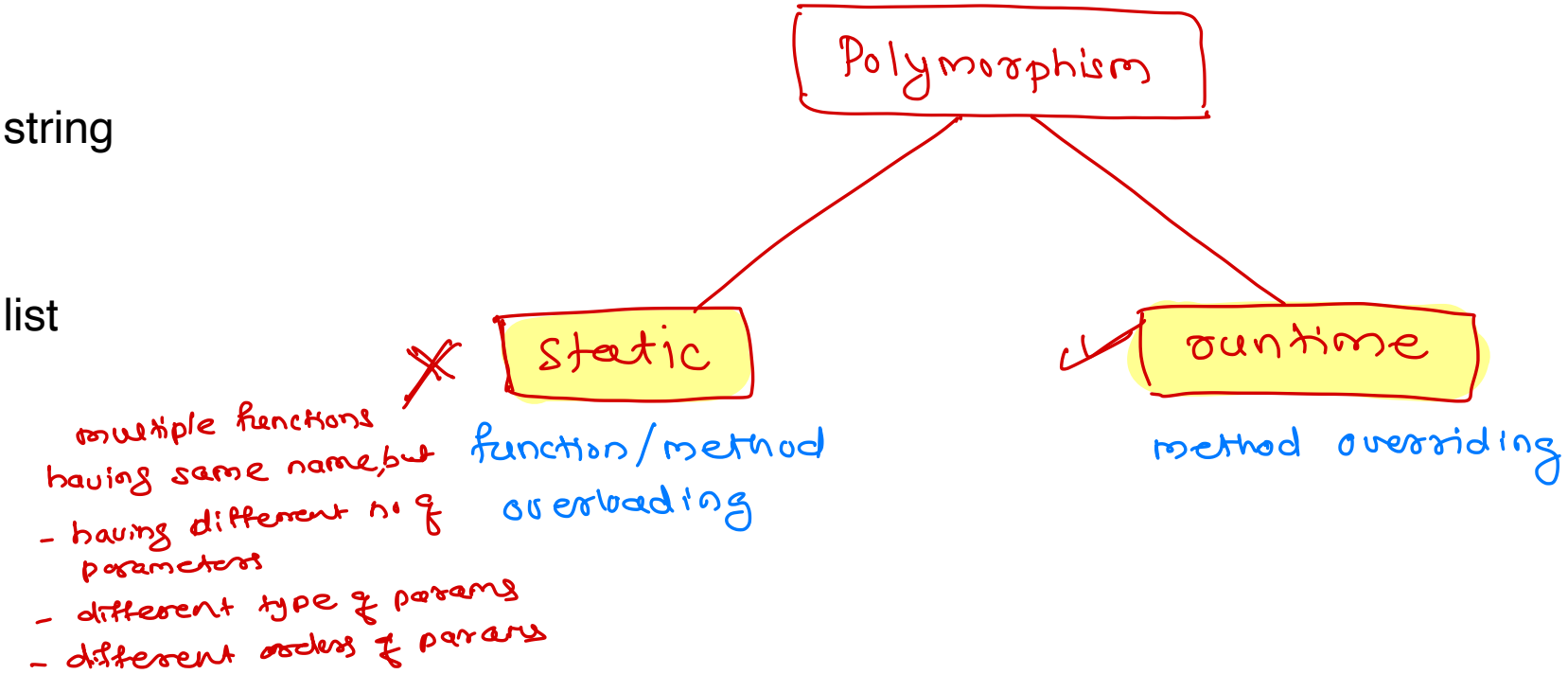
- In Python, there is something called name mangling, which means that there is limited support for a valid use-case for class-private members basically to avoid name clashes of names with names defined by subclasses
- Any identifier of the form __member (at least two leading underscores or at most one trailing underscore) is replaced with _classname__,member, where classname is the current class name with a leading underscore(s) stripped
- As long as it occurs within the definition of the class, this mangling is done
- This is helpful for letting subclasses override methods without breaking intraclass method calls



Polymorphism

- The word polymorphism means having many forms
- In programming, polymorphism means the same function name (but different signatures) being used for different types
- E.g.

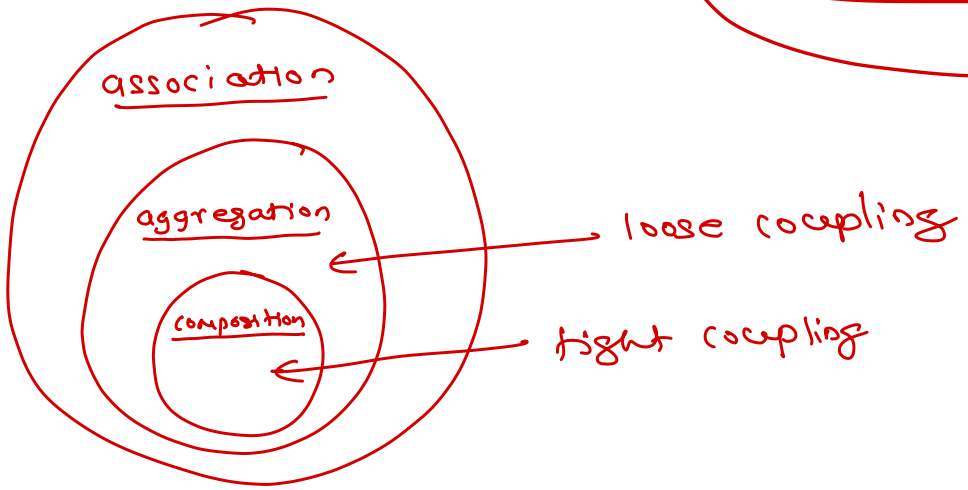
```
# len() being used for a string
print(len("python"))

# len() being used for a list
print(len([10, 20, 30]))
```



Method Overriding

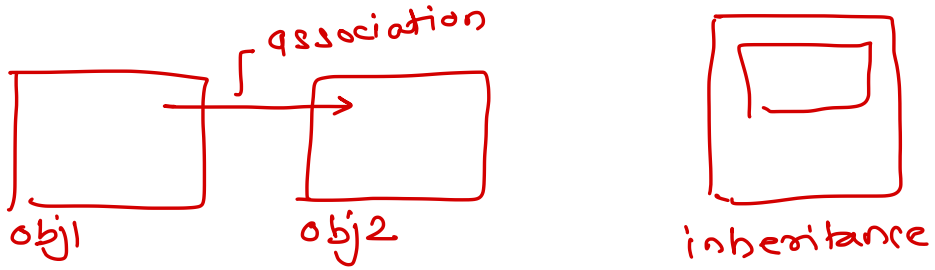
- Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class

Association



Association

- In object-oriented general software design, the relationship between one object's functionality and another's is known as an association
- Note that an association between two objects is not the same thing as inheritance between two classes
- Association means that one object uses another object or a function/method in that other object
- In other words, association is defined as the relationship between objects when one object has one or more references to other objects
-



Association vs Inheritance

- Inheritance implies that two objects are the same type of object. One object just happens to be either a more generalized or more specific version of the other object. Association occurs between two *different* objects. unrelated
- Inheritance is said to be an **IS-A** relationship whereas association is known as a **HAS-A** relationship.
- The modality of inheritance depends on the programming language features. For example, Java does not support multiple inheritance, but C++ does. On the other hand, any language can have one-to-one, one-to-many, and many-to-many associations between objects.

Composition

(strong relationship)
(tight coupling)

[part of, composed of]

- Composition is a form of association that occurs when an object's life is tightly bound to another object's life
- When the main object dies (i.e., is deleted), all the objects that are associated with that object also die
- This means that the referenced object is solely contained by the referring object

* human has-a heart
* car has-a engine
* Books has pages
* room has walls

Aggregation

(loose-coupling / weak relationship)

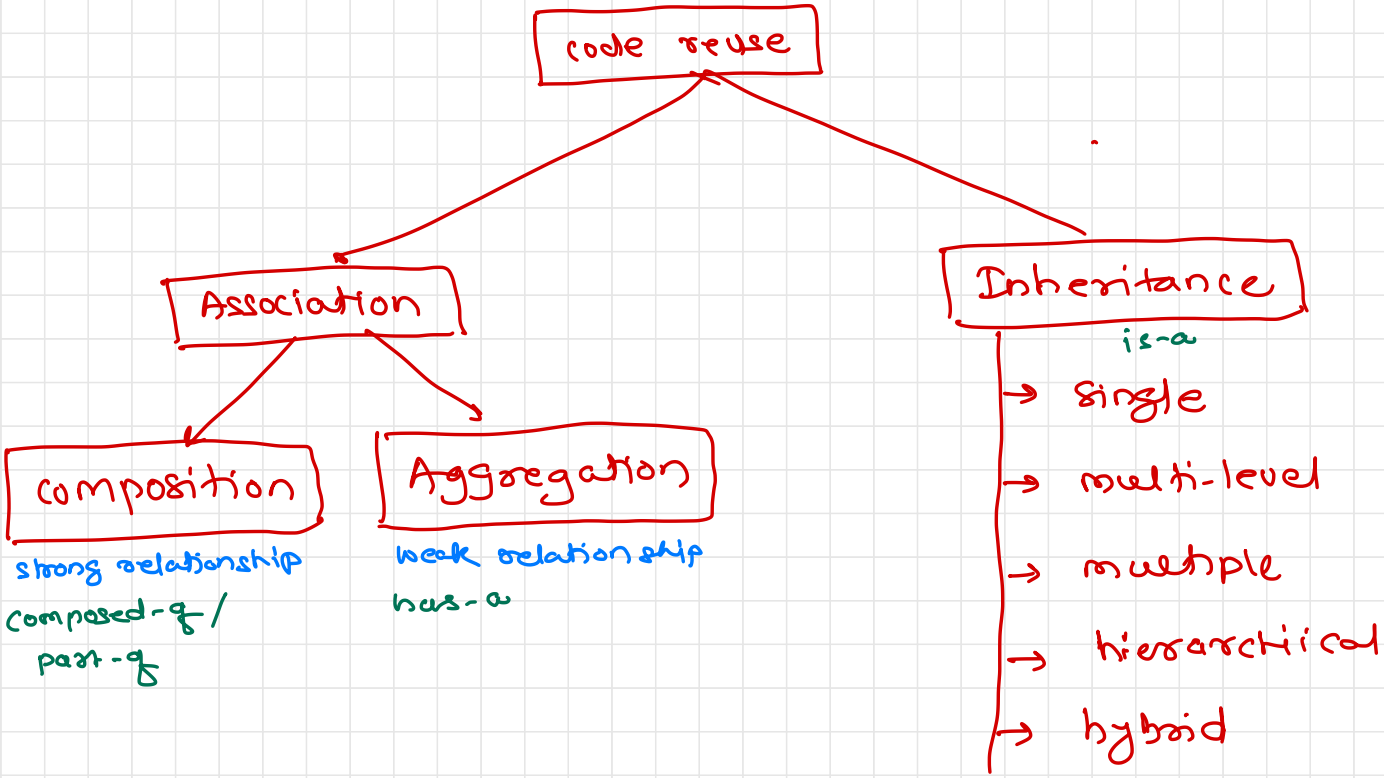
- **Aggregation** is the other form of association and is similar to composition
- In aggregation, a container object again has several references to other objects
- But, Aggregation is looser than composition
- The objects' life cycles are not bound to each other in aggregation
- Thus, the referring object may get destroyed before/after the referenced object

* Company has-a employee
* Department has-a professor

Composition vs Aggregation

- Child will co-exist with the container class
- Part-of relationship
- Stronger form of association
- Can only have one-to-one and many-to-one relationship
- Composited objects cannot be changed from the referrer

- Child can exist Independently
- Has-a relationship
- Weaker form of composition
- Can have a one-to-one, many-to-one, one-to-many and many-to-many relationship
- Aggregated objects can be removed or replaced by another same type of object


code reuse
Association
Inheritance
is-a
Composition
strong relationship
composed-of / part-of
Aggregation
weak relationship
has-a
single
multi-level
multiple
hierarchical
hybrid

Module and Package

Module

- **Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**
- Individual modules can then be cobbled together like building blocks to create a larger application
- The **module** is a simple Python file that contains collections of functions and global variables and withhaving a .py extension file
- There are several advantages to **modularizing** code in a large application
  - Simplicity
  - Maintainability
  - Reusability
  - Scoping

any file with .py extension
module

Package [directory structure for modules]

- **Packages** allow for a hierarchical structuring of the module namespace using **dot notation**
- In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names
- Package is a collection of modules