Unit 1 notes

Introduction to programming language:

A programming language is a set of instructions and syntax used to create software programs. Some of the key features of programming languages include:

1. Syntax: The specific rules and structure used to write code in a programming language.
2. Data Types: The type of values that can be stored in a program, such as numbers, strings, and Booleans.
3. Variables: Named memory locations that can store values.
4. Operators: Symbols used to perform operations on values, such as addition, subtraction, and comparison.
5. Control Structures: Statements used to control the flow of a program, such as if-else statements, loops, and function calls.
6. Libraries and Frameworks: Collections of pre-written code that can be used to perform common tasks and speed up development.
7. Paradigms: The programming style or philosophy used in the language, such as procedural, object-oriented, or functional.

A programming language is a formal language that specifies a set of instructions for a computer to perform specific tasks. It's used to write software programs and applications, and to control and manipulate computer systems. There are many different programming languages, each with its own syntax, structure, and set of commands. Some of the most commonly used programming languages include Java, Python, C++, JavaScript, and C#. The choice of programming language depends on the specific requirements of a project, including the platform being used, the intended audience, and the desired outcome.

Characteristics of a programming Language –

- A programming language must be simple, easy to learn and use, have good readability, and be human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which the ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.

- Programming language's efficiency must be high so that it can be easily converted into a machine code and its execution consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for the development, debugging, testing, maintenance of a program must be provided by a programming language.
- A programming language should provide a single environment known as Integrated Development Environment(IDE).
- A programming language must be consistent in terms of syntax and semantics.

Advantages of programming languages:

1. **Increased Productivity:** Programming languages provide a set of abstractions that allow developers to write code more quickly and efficiently.
2. **Portability:** Programs written in a high-level programming language can run on many different operating systems and platforms.
3. **Readability**: Well-designed programming languages can make code more readable and easier to understand for both the original author and other developers.
4. **Large Community:** Many programming languages have large communities of users and developers, which can provide support, libraries, and tools.

Disadvantages of programming languages:

1. **Complexity**: Some programming languages can be complex and difficult to learn, especially for beginners.
2. **Performance**: Programs written in high-level programming languages can run slower than programs written in lower-level languages.
3. **Limited Functionality**: Some programming languages may not have built-in support for certain types of tasks or may require additional libraries to perform certain functions.
4. **Fragmentation:** There are many different programming languages, which can lead to fragmentation and make it difficult to share code and collaborate with other developers.

Elements of programming language:

There are four essential ingredients. These are, Variables, Conditionals, Loops and Functions.

Variables are used to store data. They are simply labels used to reference and keep track of a directly-inserted value or some expression that evaluates to a value. All that matters is that the value in question is used to represent data. They can be primary and composite.

Primitives are elementary values. They are used to express quantities and qualities of things and most often, their literals have no underlying structure. For all our purposes we'll simply need, Numbers, Strings and Booleans.

Composites are containers which can have for elements, primitives and even other composites. Lists and Associations (Dictionaries and Mappings) are to me all you need in this category. They implement different element-identification schemes, index-based and key-value pairs, respectively.

Conditionals and Loops help us alter the execution flow from its top-bottom default and steer it through complex decision trees and iterative tasks. These are pretty standard construction in all languages and being acquainted with if-, switch-, while- and for-statement is sufficient for all uses.

Functions package a feature that applies a certain transformation to data or manages a certain resource (like a file or the screen) to transmit data through it. A function allows us to bundle a set of instructions into a callable block. It is the realization of an IPO (Input-Processing-Output) type of structure which consumes data transform it and then returns the result.

Programming language theory:

Programming language theory (PLT) is a branch of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features. It falls within the discipline of computer science, both depending on and affecting mathematics, software engineering and linguistics. It is a well-recognized branch of computer science, and an active research area, with results published in numerous journals dedicated to PLT, as well as in general computer science and engineering publications.

In some ways, the history of programming language theory predates even the development of programming languages themselves. The lambda calculus, developed by Alonzo Church and Stephen Cole Kleene in the 1930s, is considered by some to be the world's first programming language, even though it was intended to model computation rather than being a means for programmers to describe algorithms to a computer system.

The lowercase Greek letter λ (lambda) is an unofficial symbol of the field of programming-language theory. Lambda calculus is based on minimal construction of abstractions(functions) and substitution(application)

Bohm jacopini theorem:

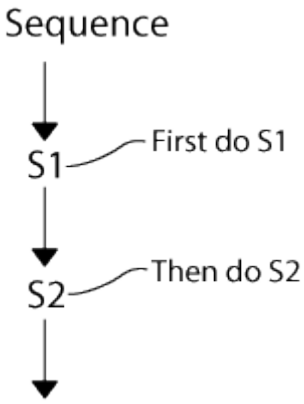
The types of programs we've written so far for the Turing machine are what are usually called non-structured. This type of code is generally very difficult to read and understand. It really has very little to do with how we program in a modern high level (further abstracted from the machine) programming language. Corrado Böhm and Giuseppe Jacopini showed in 1966 that any non-structured program can be rewritten by combining three techniques: sequence, selection, and repetition (or loop). This is something we're familiar with and is how we've been writing our algorithms since we started, but now we can see the connection between the Turing machine and what we've been doing.

The structured program theorem, also called the Böhm–Jacopini theorem,[1][2] is a result in programming language theory. It states that a class of control-flow graphs (historically called flowcharts in this context) can compute any computable function if it combines subprograms in only three specific ways (control structures). These are

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a boolean expression (selection)
3. Repeatedly executing a subprogram as long as a boolean expression is true (iteration)

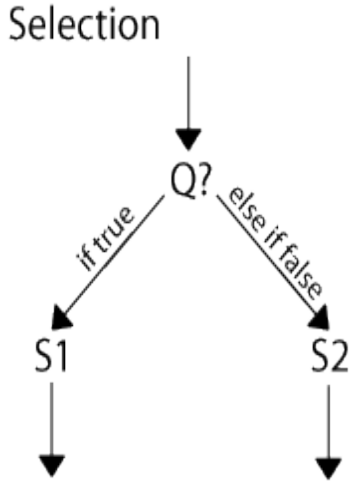
The structured chart subject to these constraints may however use additional variables in the form of bits (stored in an extra integer variable in the original proof) in order to keep track of information that the original program represents by the program location. The construction was based on Böhm's programming language P″.

To review, sequence performs some operations S1; S2 meaning perform S1; then perform S2.



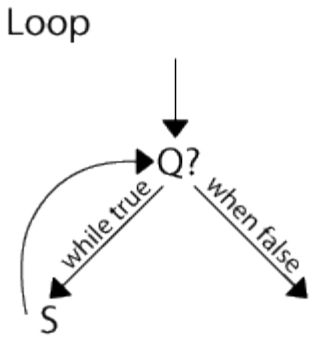

```
def addtwo1():
x = int(raw_input("Enter an integer: "))
ans = x + 2
return ans
```

Selection says, if Q is true, then perform S1, else perform S2.




```
def gtltzero():
x = int(raw_input("Enter an integer: "))
if x < 0:
print "Negative number"
elif x == 0:
print "Zero"
else:
print "Positive number"
return
```

Loop says while Q is true, do S.




```
def square(size):
for x in range(size):
#we wrote this as "for each x" before
for y in range(size):
print "*",
#The comma means not to add a newline after the printing
print "\n",
#\n lets you print a newline
return
```

Multiple programming paradigm:

A multi-paradigm programming language is one that is *equally well-suited* in more than one programming paradigm.

Python – A Multi Paradigm Language-Python is a multi-paradigm programming language. Meaning it supports different styles of writing code. One can write Python code in procedural, object oriented, functional or imperative manner. For this reason, Python is considered a “swiss army knife” in the developers toolbox.

Characteristics of python:

Python is Interpreted: Python is processed at runtime by the interpreter, it does not require compiling the whole program before executing it. The python interpreter directly executes the program, line by line translating each statement into a sequence of subroutines and then into machine code, this makes python more flexible than other programming language by providing smaller executable program size.

Python is Interactive :Python being an Interpreted language, enables users to interact with the python interpreter directly to write the programs, just by sitting at a python prompt. The Python prompt gives various detailed messages for any type of error or for any specific command being executed, it also supports interactive testing and debugging of snippets of code.

Python is Object-Oriented: Python is a Multi-Paradigm programming language Hence, it supports object-oriented style, rules and techniques of programming that encapsulates code within objects. Also, each and everything written in the python source code is in the form of classes and objects.

Python is a Beginner's Language: Python is the best option for Initial or beginner-level programmers. It supports development for a wide range of applications ranging from simple-text processing to world Wide Web Browsers to games.

Python is convenient to write and is quite readable: Python is a strongly typed language, it has few keywords, simple structure and a clearly defined syntax. This allows any amateur individual to pick up the language quickly. Python's code is clearly defined and visible to the eyes, as it uses whitespaces to indicate the beginning and end of a block, i.e. it purely works on indentation.

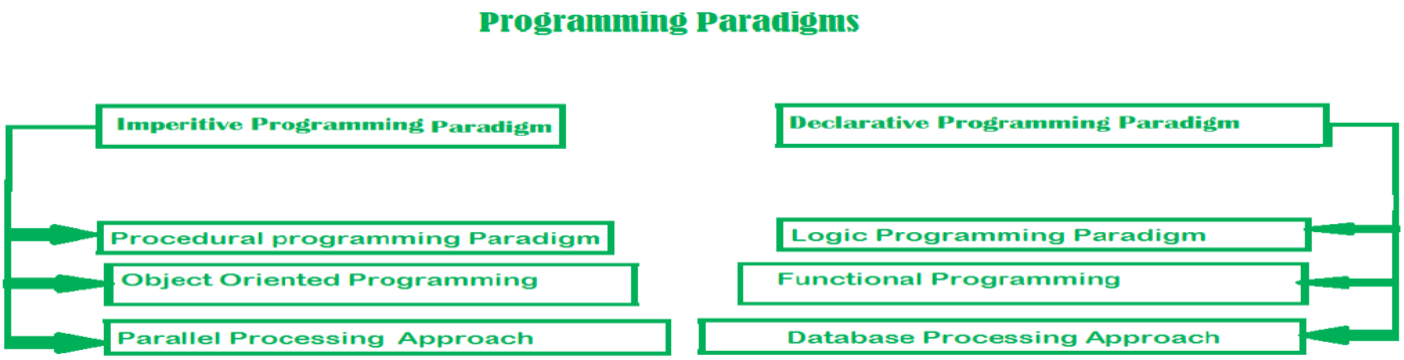
Python is Portable: Python has the capability to run on a very wide variety of hardware platforms with the same interface. It seamlessly runs on almost all operating systems such as Windows, Linux, UNIX, Amigo, Mac OS etc, without any changes.

Python is Extendable: Python is so versatile and flexible programming language with a broad standard Library that it allows the programmers to add existing or create new low-level/High-Level modules and packages to the python interpreter. These modules and tool-packages enables the developers with very portable and cross-platform development possibilities which helps them to create and customize their programs, applications or software tools to be more accurate and efficient.

Programming paradigm:

Programming paradigms are different ways or styles in which a given program or programming language can be organized. Each paradigm consists of certain structures, features, and opinions about how common programming problems

should be tackled. There are lots of programming languages that are well-known but all of them need to follow some strategy when they are implemented. And that strategy is a paradigm. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand. They are discussed below:



1. Imperative programming- Imperative programming is a software development paradigm where functions are implicitly coded in every step required to solve a problem. In imperative programming, every operation is coded and the code itself specifies how the problem is to be solved, which means that pre-coded models are not called on.
   Imperative programming requires an understanding of the functions necessary to solve a problem, rather than a reliance on models that are able to solve it. The focus of imperative programming is how the problem should be solved, which requires a detailed step-by-step guide. Because the written code performs the functions instead of models, the programmer must code each step. Therefore, the source code for imperative languages is a series of commands, which specify what the computer has to do – and when – in order to achieve a desired result. Values used in variables are changed at program runtime. To control the commands, control structures such as loops or branches are integrated into the code.
   Imperative programming languages are very specific, and operation is system-oriented. On the one hand, the code is easy to understand; on the other hand, many lines of source text are required to describe what can be achieved with a fraction of the commands using declarative programming languages.

```
sample_characters = ['p','y','t','h','o','n']
```

```
sample_string = ''
for c in sample_characters:
    sample_string = sample_string + c
print(sample_string)
```

2. Procedural programming paradigm: The procedural programming paradigm is a subtype of imperative programming in which statements are structured into procedures (also known as subroutines or functions). The program composition is more of a procedure call where the programs might reside somewhere in the universe and the execution is sequential, thus becoming a bottleneck for resource utilization. Like the imperative programming paradigm, procedural programming follows the stateful model. The procedural programming paradigm facilitates the practice of good program design and allows modules to be reused in the form of code libraries.

   This modularized form of development is a very old development style. The different modules in a program can have no relationship with each other and can be located in different locations, but having a multitude of modules creates hardships for many developers, as it not only leads to duplication of logic but also a lot of overhead in terms of finding and making the right calls. Note that in the following implementation, the method stringify could be defined anywhere in the universe and, to do its trick, only require the proper call with the desired arguments.

```
def stringify(characters):
    string = ''
    for c in characters:
        string = string + c
    return stringify

sample_characters = ['p','y','t','h','o','n']

stringify(sample_characters)
```

3. Object-oriented programming paradigm:The object-oriented programming paradigm considers basic entities as objects whose instance can contain both data and the corresponding methods to modify that data. The different principles of object-oriented design help code reusability, data hiding, etc., but it is a complex beast, and writing the same logic an in object-oriented method is tricky. For example:

```
class StringOps:
    def __init__(self, characters):
        self.characters = characters
    def stringify(self):
        self.string = ''.join(self.characters)
sample_characters = ['p','y','t','h','o','n']
sample_string = StringOps(sample_characters)
sample_string.stringify()
sample_string.string
```

4. Parallel processing approach – Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function. Parallel processing is a mode of operation where the task is executed simultaneously in multiple processors in the same computer. It is meant to reduce the overall processing time.

   However, there is usually a bit of overhead when communicating between processes which can actually increase the overall time taken for small tasks instead of decreasing it. In python, the multiprocessing module is used to run independent parallel processes by using subprocesses (instead of threads).

   The maximum number of processes you can run at a time is limited by the number of processors in your computer. If you don’t know how many

processors are present in the machine, the cpu_count() function in multiprocessing will show it.

```
import multiprocessing as mp
print("Number of processors: ", mp.cpu_count())
```

5. Declarative Programming Languages-Declarative Programming Languages focus on on describing what should be computed - and avoid mentioning how that computation should be performed. In practice this means avoiding expressions of control flow: loops and conditional statements are removed and replaced with higher level constructs that describe the logic of what needs to be computed.

   The usual example of a declarative programming language is SQL. It lets you define what data you want computed - and translates that efficiently onto the database schema. This lets you avoid having to specify details of how to execute the query, and instead lets the query optimizer figure out the best index and query plan on a case by case basis.

```
mylist = [1,2,3,4,5]
total = sum(mylist)
print(total)
```

6. Functional programming paradigm: The functional programming paradigm treats program computation as the evaluation of mathematical functions based on lambda calculus. Lambda calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It follows the "what-to-solve" approach—that is, it expresses logic without describing its control flow—hence it is also classified as the declarative programming model.

   The functional programming paradigm promotes stateless functions, but it's important to note that Python's implementation of functional programming deviates from standard implementation. Python is said to be an *impure* functional language because it is possible to maintain state and create side effects if you are not careful. That said, functional programming is handy for parallel processing and is super-efficient for tasks requiring recursion and concurrent execution.

```
sample_characters = ['p','y','t','h','o','n']
import functools
sample_string = functools.reduce(lambda s,c:
s + c, sample_characters)
sample_string
```

7. Logical programming- Logic programming is a programming paradigm that is based on logic. This means that a logic programming language has sentences that follow logic, so that they express facts and rules. Computation using logic programming is done by making logical inferences based on all available data. In order for computer programs to make use of logic programming, there must be a base of existing logic, called predicates. Predicates are used to build atomic formulas or atoms, which state true facts. Predicates and atoms are used to create formulas and perform queries. Logic languages most often rely on queries in order to display relevant data. These queries can exist as part of machine learning, which can be run without the need for manual intervention.

8. Database/Data driven programming approach – This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

```
CREATE DATABASE databaseAddress;

CREATE TABLE Addr (

PersonID int,

LastName varchar(200),

FirstName varchar(200),
```

Address varchar(200),

City varchar(200),

State varchar(200)

);

Machine codes for procedural programming:

Python Functions

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in Python programming:

- Standard library functions - These are built-in functions in Python that are available to use.
- User-defined functions - We can create our own functions based on our requirements.

Python Library Functions

In Python, standard library functions are the built-in functions that can be used directly in our program. For example,

- print() - prints the string inside the quotation marks
- sqrt() - returns the square root of a number

- pow() - returns the power of a number

These library functions are defined inside the module. And, to use them we must include the module inside our program.

For example, sqrt() is defined inside the math module.

Python Function Declaration:

The syntax to declare a function is:

```
def function_name(arguments):
    # function body
      return
```

Here,

- def - keyword used to declare a function
- function_name - any name given to the function
- arguments - any value passed to function
- return (optional) - returns value from a function

Let's see an example,

```
def greet():
    print('Hello World!')
```

Here, we have created a function named greet(). It simply prints the text Hello World!.

Calling a Function in Python

In the above example, we have declared a function named greet().

```
def greet():
    print('Hello World!')
```

Now, to use this function, we need to call it.

Here's how we can call the greet() function in Python.

```
# call the function
greet()
```

Machine code:

```
mylist = [10, 20, 30, 40]
def sum_the_list(mylist):
    res = 0
    for val in mylist:
        res += val
    return res
 print(sum_the_list(mylist))
```

Python Function Arguments

A function can also have arguments. An argument is a value that is accepted by a function. For example,

Example 1: Python Function Arguments

```
def add_numbers(a, b):
    sum = a + b
    print('Sum:', sum)
add_numbers(2, 3)
# Output: Sum: 5
```

In the above example, the function add_numbers() takes two parameters: a and b. Notice the line,

```
add_numbers(2, 3)
```

Here, add_numbers(2, 3) specifies that parameters a and b will get values **2** and **3** respectively.

Function Argument with Default Values

In Python, we can provide default values to function arguments.

We use the = operator to provide default values. For example,

```
def add_numbers( a = 7,  b = 8):
    sum = a + b
    print('Sum:', sum)
# function call with two arguments
add_numbers(2, 3)
#  function call with one argument
add_numbers(a = 2)
# function call with no arguments
add_numbers()
```

Here, we have provided default values **7** and **8** for parameters a and b respectively. Here's how this program works

1. add_number(2, 3)

Both values are passed during the function call. Hence, these values are used instead of the default values.

2. add_number(2)

Only one value is passed during the function call. So, according to the positional argument **2** is assigned to argument a, and the default value is used for parameter b.

3. add_number()

No value is passed during the function call. Hence, default value is used for both parameters a and b.

Python Function With Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python.

Arbitrary arguments allow us to pass a varying number of values during a function call.

We use an asterisk (*) before the parameter name to denote this kind of argument. For example,

```
# program to find sum of multiple numbers

def find_sum(*numbers):
    result = 0
    for num in numbers:
        result = result + num
    print("Sum = ", result)

# function call with 3 arguments

find_sum(1, 2, 3)

# function call with 2 arguments

find_sum(4, 9)
```

Output

```
Sum =  6

Sum =  13
```

Positional Arguments in Python

During a function call, values passed through arguments should be in the order of parameters in the function definition. This is called positional arguments.

Keyword arguments should follow positional arguments only.

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

The above function can be called in two ways:

First, during the function call, all arguments are given as positional arguments. Values passed through arguments are passed to parameters by their position. 10 is assigned to a, 20 is assigned to b and 30 is assigned to c.

```
print (add(10,20,30))
```

Keyword Arguments in Python

Functions can also be called using keyword arguments of the form kwarg=value.

During a function call, values passed through arguments don't need to be in the order of parameters in the function definition. This can be achieved by keyword arguments. But all the keyword arguments should match the parameters in the function definition.

```
def add(a,b=5,c=10):
    return (a+b+c)
```

Calling the function add by giving keyword arguments

All parameters are given as keyword arguments, so there's no need to maintain the same order.

```
print (add(b=10,c=15,a=20))

#Output:45
```

During a function call, only giving a mandatory argument as a keyword argument. Optional default arguments are skipped.

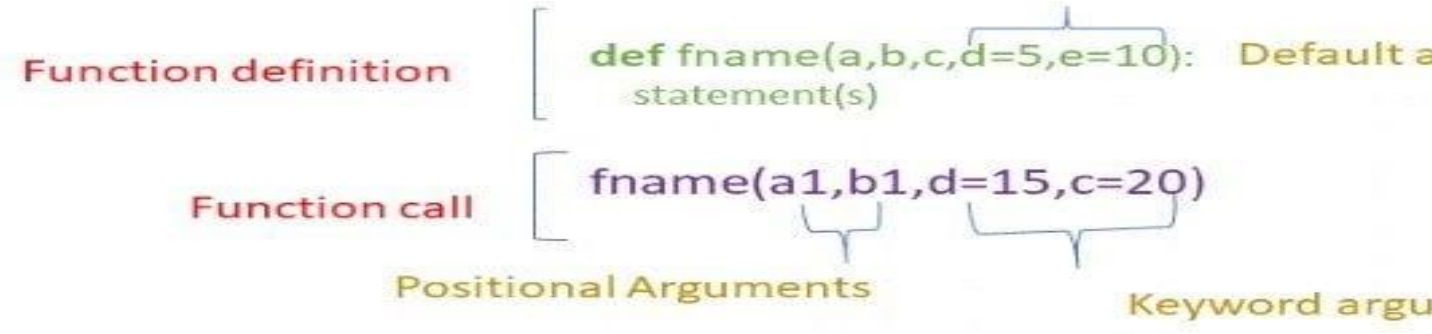
```
print (add(a=10))

#Output:25
```

Arbitrary keyword arguments in python

For arbitrary positional argument, a double asterisk (**) is placed before a parameter in a function which can hold keyword variable-length arguments.

```
def fn(**a):
    for i in a.items():
        print (i)
fn(numbers=5,colors="blue",fruits="apple")
'''
Output:
('numbers', 5)
('colors', 'blue')
('fruits', 'apple')
'''
```

Default vs positional vs keyword arguments



Machine code-object oriented programming:

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with properties like a name, age, and address and behaviors such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called methods, which identify the behaviors and actions that an object created from the class can perform with its data.

In this tutorial, you'll create a Dog class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how something should be defined. It doesn't actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the Dog class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information. Just like many people can fill out the same form with their own unique information, many instances can be created from a single class.

Let's define a class named Book for a bookseller's sales software.

```
class Book:
    def __init__(self, title, quantity, author, price):
        self.title = title
        self.quantity = quantity
        self.author = author
```

```
        self.price = price
```

The __init__ special method, also known as a **Constructor**, is used to initialize the Book class with attributes such as title, quantity, author, and price.

In Python, built-in classes are named in lower case, but user-defined classes are named in Camel or Snake case, with the first letter capitalized.

This class can be instantiated to any number of objects. Three books are instantiated in the following example code:

```
book1 = Book('Book 1', 12, 'Author 1', 120)
book2 = Book('Book 2', 18, 'Author 2', 220)
book3 = Book('Book 3', 28, 'Author 3', 320)
```

book1, book2 and book3 are distinct objects of the class *Book*. The term **self** in the attributes refers to the corresponding instances (objects).

```
print(book1)
print(book2)
print(book3)
```

Object Methods:

Objects can also contain methods. Methods in objects are functions that belong to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

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

  def myfunc(self):
    print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)
p1.myfunc()
```

Python Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 2: Use of Inheritance in Python

```
# base class
class Animal:

    def eat(self):
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# derived class
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

# Create object of the Dog class
dog1 = Dog()

# Calling members of the base class
dog1.eat()
dog1.sleep()

# Calling member of the derived class
dog1.bark();
```

Python Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class. It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve **data hiding**.

In Python, we denote private attributes using underscore as the prefix i.e single _ or double __. For example,

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

Polymorphism

Polymorphism is another important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Let's see an example,

```
class Polygon:
    # method to render a shape
    def render(self):
        print("Rendering Polygon...")

class Square(Polygon):
    # renders Square
    def render(self):
        print("Rendering Square...")

class Circle(Polygon):
    # renders circle
    def render(self):
        print("Rendering Circle...")

# create an object of Square
s1 = Square()
s1.render()

# create an object of Circle
c1 = Circle()
c1.render()
```

Method call overhead:

Efficient function calls in Python are the secret sauce to writing code that is both streamlined and optimized. In the vast world of Python programming, understanding how to make your function calls efficient can significantly impact the performance and readability of your code. By employing smart techniques and leveraging Python's built-in features, you can minimize overhead, reduce redundancy, and maximize the speed and efficiency of your program.

The benchmarks are variations of comparing the execution time of:

```
for i in range(n):
    pass
```

and

```
def f():
    pass
for i in range(n):
    f()
```

for some suitably large n

Lets take another example, if we define

```
def a():
    return 29

def b():
    return a()

def c():
    return b()
```

then by timing how much longer b() takes than a() and c() than b() and so on then can attribute that increase to function call overhead

When we discuss the overhead of a Python function call, we're really talking about three things:

Creating the stack frame: Python needs a place to keep track of the function's variables and what it's supposed to do next after the function call. This is called a stack frame. It's like setting up a dedicated workspace each time you start a new task.

Storing the variables: Python has to store the function's local variables somewhere. This storage costs space and time.

Jumping to and from the function: Finally, Python has to take a leap of faith, jumping to the function and back again. Like any athletic feat, this requires energy (i.e., computational resources).

These three steps happen every single time a function is called in Python. The cost may seem negligible at first, but when your code involves thousands or even millions of function calls, it quickly adds up.

Dynamic memory allocation for message and object storage:

Memory management is automatically handled by the interpreter in the realm of Python.

Static Memory Allocation in Python**:** Static memory allocation refers to declaring fixed-size data structures or allocating memory for data structures at the time of compilation. Even though Python doesn't explicitly allocate static memory, there are some situations in which static-like behavior is preferred. Let's investigate these situations:

Example

```
def static_memory_allocation():
# Create a fixed-size tuple
my_tuple = (1, 2, 3, 4, 5)
    # Print the tuple
    print("Tuple with static-like memory allocation:")
    print(my_tuple)
    # Attempt to modify the tuple (this will raise an error)
    try:
        my_tuple[0] = 10
    except TypeError as e:
        print(f"Error: {e}")
static_memory_allocation()
```

Strings, tuples, and frozensets are examples of immutable objects in Python that behave similarly to static memory allocation. Once they've been created, their values can't be changed. Python reduces memory use by recycling immutable objects as often as feasible. For instance, if you make two strings with the same value, they will both refer to the same location in memory and use less memory.

Stack Allocation: The Stack data structure is used to store the static memory. It is only needed inside the particular function or method call. The function is added in program's call stack whenever we call it. Variable assignment inside the function is temporarily stored in the function call stack; the function returns the value, and the call stack moves to the text task.

Dynamic Memory Allocation in Python: By allocating memory at runtime, dynamic memory allocation offers flexibility in memory utilization. Dynamic memory allocation in Python usually works through making use of pre-built data structures like lists, dictionaries, and objects.

Example

```
def dynamic_memory_allocation():
    # Create an empty list
    my_list = []
    # Add elements to the list dynamically
    for i in range(1, 6):
        my_list.append(i)
    # Print the list
    print("List after dynamic allocation:")
    print(my_list)
    # Remove an element from the list dynamically
    my_list.remove(3)
    # Print the list after removing an element
    print("List after removing an element:")
    print(my_list)
dynamic_memory_allocation()
```

Python lists are dynamically allocated and can expand or contract as necessary. A block of memory is allocated to accommodate the list's initial size when it is initialized. If more space is needed for the list to grow, new memory is allocated and the current components are copied over to it. Large datasets may be handled and effective memory use is ensured by this dynamic scaling.

Heap Memory Allocation: Heap data structure is used for dynamic memory which is not related to naming counterparts. It is type of memory that uses outside the program at the global space. One of the best advantages of heap memory is to it freed up the memory space if the object is no longer in use or the node is deleted.

Python Garbage Collector: Python removes those objects that are no longer in use or can say that it frees up the memory space. This process of vanish the unnecessary object's memory space is called the Garbage Collector. The Python garbage collector initiates its execution with the program and is activated if the reference count falls to zero. When we assign the new name or placed it in containers such as a dictionary or tuple, the reference count increases its value. If we reassign the reference to an object, the reference counts decreases its value if. It also decreases its value when the object's reference goes out of scope or an object is deleted.

Python Objects in Memory: As we know, everything in Python is object. The object can either be simple (containing numbers, strings, etc.) or containers (dictionary, lists, or user defined classes). In Python, we don't need to declare the variables or their types before using them in a program.

Let's understand the following example.

Example -

```
a= 10

print(a)

del a

print(a)
```

Output:

```
10

Traceback (most recent call last):

  File "", line 1, in

    print(x)

NameError : name 'a' is not defined
```

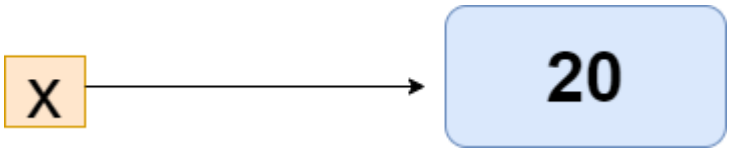
As we can see in the above output, we assigned the value to object x and printed it. When we remove the object x and try to access in further code, there will be an error that claims that the variable x is not defined.

Reference Counting in Python

Reference counting states that how many times other objects reference an object. When a reference of the object is assigned, the count of object is incremented one. When references of an object are removed or deleted, the count of object is decremented. The Python memory manager performs the de-allocation when the reference count becomes zero. Let's make it simple to understand.

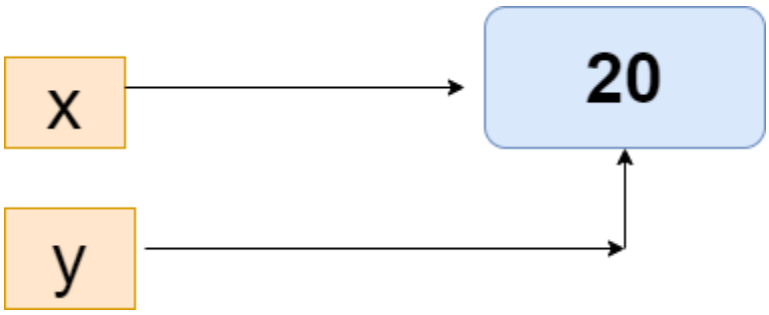
Example -Suppose, there is two or more variable that contains the same value, so the Python virtual machine rather creating another object of the same value in the private heap. It actually makes the second variable point to that the originally existing value in the private heap. This is highly beneficial to preserve the memory, which can be used by another variable.

```
x = 20
```



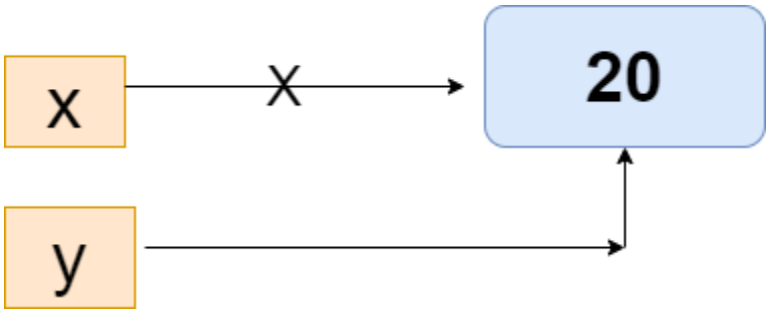
When we assign the value to the x. the integer object 10 is created in the Heap memory and its reference is assigned to x.

```
x = 20

y = x

if id(x) == id(y):

    print("The variables x and y are referring  to the same object")
```



In the above code, we have assigned y = x, which means the y object will refer to the same object because Python allocated the same object reference to new variable if the object is already exists with the same value.

Now, see another example.



Example -

```
x = 20

y = x

x += 1

If id(x) == id(y):

    print("x and y do not refer to  the same object")
```

Output:

x and y do not refer to the same object

The variables x and y are not referring the same object because x is incremented by one, x creates the new reference object and y still referring to 10.

Dynamically dispatched message calls and direct procedure call overheads:

In computer science, dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time. It is commonly employed in, and considered a prime characteristic of, object-oriented programming (OOP) languages and systems.

Dynamic dispatch contrasts with static dispatch, in which the implementation of a polymorphic operation is selected at compile time. The purpose of dynamic dispatch is to defer the selection of an appropriate implementation until the run time type of a parameter (or multiple parameters) is known.

Dynamic dispatch is different from late binding (also known as dynamic binding). Name binding associates a name with an operation. A polymorphic operation has several implementations, all associated with the same name. Bindings can be made at compile time or (with late binding) at run time. With dynamic dispatch, one particular implementation of an operation is chosen at run time. While dynamic dispatch does not imply late binding, late binding does imply dynamic dispatch, since the implementation of a late-bound operation is not known until run time.

A language may be implemented with different dynamic dispatch mechanisms. The choices of the dynamic dispatch mechanism offered by a language to a large extent alter the programming paradigms that are available or are most natural to use within a given language.

Normally, in a typed language, the dispatch mechanism will be performed based on the type of the arguments (most commonly based on the type of the receiver of a message). Languages with weak or no typing systems often carry a dispatch table as part of the object data for each object. This allows instance behaviour as each instance may map a given message to a separate method.Some languages offer a hybrid approach.

Dynamic dispatch will always incur an overhead so some languages offer static dispatch for particular methods.

```
class Cat:
    def speak(self):
        print("Meow")
class Dog:
    def speak(self):
```

```
        print("Woof")

def speak(pet):
    # Dynamically dispatches the speak method
    # pet can either be an instance of Cat or Dog
    pet.speak()

cat = Cat()

speak(cat)

dog = Dog()

speak(dog)
```

Object serialization: Serialization refers to the process of converting a data object (e.g., Python objects, Tensorflow models) into a format that allows us to store or transmit the data and then recreate the object when needed using the reverse process of deserialization.

There are different formats for the serialization of data, such as JSON, XML, HDF5, and Python's pickle, for different purposes. JSON, for instance, returns a human-readable string form, while Python's pickle library can return a byte array.

Serialization is the process of converting the object into a format that can be stored or transmitted. After transmitting or storing the serialized data, we are able to reconstruct the object later and obtain the exact same structure/object, which makes it really convenient for us to continue using the stored object later on instead of reconstructing the object from scratch.

In Python, there are many different formats for serialization available. One common example of hash maps (Python dictionaries) that works across many languages is the JSON file format which is human-readable and allows us to store the dictionary and recreate it with the same structure. But JSON can only store basic structures such as a list and dictionary, and it can only keep strings and numbers. We cannot ask JSON to remember the data type (e.g., numpy float32 vs. float64). It also cannot distinguish between Python tuples and lists.

Object serialization is the process of converting state of an object into byte stream. This byte stream can further be stored in any file-like object such as a disk file or memory stream. It can also be transmitted via sockets etc. Deserialization is the process of reconstructing the object from the byte stream.

Python refers to serialization and deserialization by terms pickling and unpickling respectively. The 'pickle' module bundled with Python's standard library defines functions for serialization (dump() and dumps()) and deserialization (load() and loads()).

The data format of pickle module is very Python specific. Hence, programs not written in Python may not be able to deserialize the encoded (pickled) data properly. In fact it is not considered to be secure to unpickle data from unauthenticated source.

The pickle module is part of the Python standard library and implements methods to serialize (pickling) and deserialize (unpickling) Python objects. To get started with pickle, import it in Python:

Following program pickle a dictionary object into a binary file.

```
import pickle

f=open("pickled.txt","wb")

dct={"name":"Rajeev", "age":23, "Gender":"Male","marks":75}

pickle.dump(dct,f)

f.close()
```

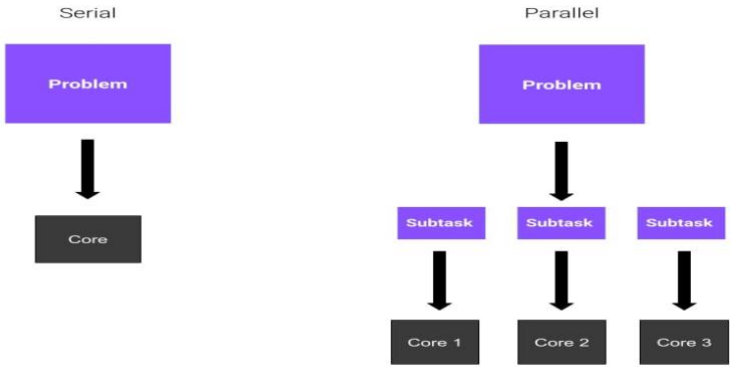
On the other hand load() function unpickles or deserializes data from binary file back to Python dictionary.

```
import pickle

f=open("pickled.txt","rb")

d=pickle.load(f)

print (d)

f.close()
```

Parallel Computing: Imagine you have a huge problem to solve, and you're alone. You need to calculate the square root of eight different numbers. What do you do? Well, you don't have many options. You start with the first number, and you calculate the result. Then, you go on with the others.

What if you have three friends good at math willing to help you? Each of them will calculate the square root of two numbers, and your job will be easier because the workload is distributed equally between your friends. This means that your problem will be solved faster.

Okay, so all clear? In these examples, each friend represents a core of the CPU. In the first example, the entire task is solved sequentially by you. This is called serial computing. In the second example, since you're working with four cores in total, you're using parallel computing. Parallel computing involves the usage of parallel processes or processes that are divided among multiple cores in a processor.



There are mainly three models in parallel computing:

- Perfectly parallel : The tasks can be run independently, and they don't need to communicate with each other.
- Shared memory parallelism: Processes (or threads) need to communicate, so they share a global address space.
- Message passing: Processes need to share messages when needed.

There are two main ways to handle parallel programs:

- **Shared Memory**

In shared memory, the sub-units can communicate with each other through the same memory space. The advantage is that you don't need to handle the communication explicitly because this approach is sufficient to read or write from the shared memory. But the problem arises when multiple process access and change the same memory location at the same time. This conflict can be avoided using synchronization techniques.

- **Distributed memory**

In distributed memory, each process is totally separated and has its own memory space. In this scenario, communication is handled explicitly between the processes. Since the communication happens through a network interface, it is costlier compared to shared memory.

Benefits of Using Multiprocessing:

Here are a few benefits of multiprocessing:

- better usage of the CPU when dealing with high CPU-intensive tasks
- more control over a child compared with threads
- easy to code

The first advantage is related to performance. Since multiprocessing creates new processes, you can make much better use of the computational power of your CPU by dividing your tasks among the other cores. Most processors are multi-core processors nowadays, and if you optimize your code you can save time by solving calculations in parallel.

The second advantage looks at an alternative to multiprocessing, which is multithreading. Threads are not processes though, and this has its consequences. If you create a thread, it's dangerous to kill it or even interrupt it as you would do with a normal process. Since the comparison between multiprocessing and multithreading isn't in the scope of this article, I encourage you to do some further reading on it.

The third advantage of multiprocessing is that it's quite easy to implement, given that the task you're trying to handle is suited for parallel programming.

Program to implement the concept of parallel computing using multiprocessing module in python

```
from multiprocessing import Process
  def cube(x):
    for x in my_numbers:
      print( x**3)
    if __name__ == '__main__':
      my_numbers = [3, 4, 5, 6, 7, 8]
      p = Process(target=cube, args=('x',))
      p.start()
p.join
print ("Done")
```