MODIBBO ADAMA UNIVERSITY, YOLA

DEPARTMENT OF COMPUTER SCIENCE

CSC202: COMPUTER PROGRAMMING II

(3 Units)

Textbooks:

- Introduction to Programming, from Virtual University of Pakistan.
- A Computer Science Tapestry, 2nd Edition by Owen L. Astrachan.
- Think Python: How to Think Like a Computer Scientist by Allen Downey
- Computer Programming for Beginners by Tim Wired.

Topics to be cover:

Principles of good programming, Structured programming concepts, Debugging and testing, String processing, Internal searching and sorting, Recursion. The programming language to be used Is Python.

WHAT IS COMPUTER PROGRAMMING

The question most of the people ask is **why should we learn to program** when there are so many application software and code generators available to do the task for us.

“The answer consists of two parts. First, it is indeed true that traditional forms of programming are useful for just a few people. But programming as we the authors understand it is useful for everyone: the administrative secretary who uses spreadsheets as well as the high-tech programmer. In other words, we have a broader

Hence learning to program is important because **it develops analytical and problem-solving abilities.** It is a creative activity and provides us a mean to express abstract ideas. Thus, programming is fun and is much more than a vocational skill. By designing programs, we learn many skills that are important for all professions.

These skills can be summarized as:

- ***Critical reading***
- ***Analytical thinking***
- ***Creative synthesis***

Programming is an important activity as people’s life and living depends on the programs one make. Hence while programming one should

- Paying attention to detail
- Think about the reusability.
- Think about user interface
- Understand the fact the computers are stupid
- Comment the code liberally

Paying attention to detail

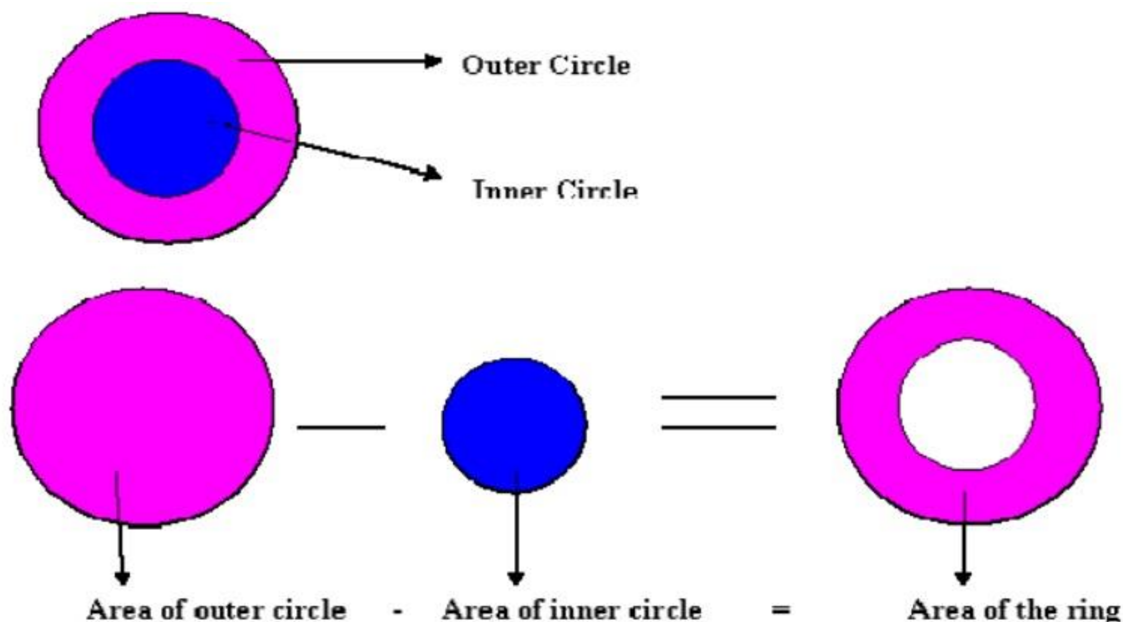
In programming, the details matter. This is a very important skill. A good programmer always analyzes the problem statement very carefully and in detail. You should pay attention to all the aspects of the problem. You can't be vague. You can't describe your program 3/4th of the way, then say, "You know what I mean?'', and have the compiler figure out the rest.

Furthermore, you should pay attention to the calculations involved in the program, its flow, and most importantly, the logic of the program. Sometimes, a grammatically correct sentence does not make any sense. A program can be grammatically correct. It compiles and runs but produces incorrect or absurd results and does not solve the problem. It is very important to pay attention to the logic of the program.

Think about the reusability

Whenever you are writing a program, always keep in mind that it could be reused at some other time. Also, try to write in a way that it can be used to solve some other related problem. A classic example of this is:

Suppose we have to calculate the area of a given circle. We know the area of a circle is (Pi * r2). Now we have written a program which calculates the area of a circle with given radius. At some later time, we are given a problem to find out the area of a ring. The area of the ring can be calculated by subtracting the area of outer circle from the area of the inner circle. Hence, we can use the program that calculates the area of a circle to calculate the area of the ring.



Think about Good user interface

As programmers, we assume that computer users know a lot of things, this is a big mistake. So never assume that the user of your program is computer literate. Always provide an easy to understand and easy to use interface that is self-explanatory.

Understand the fact that computers are stupid

Computers are incredibly stupid. They do exactly what you tell them to do: no more, no less-- unlike human beings. Computers can't think by themselves. In this sense, they differ from human beings. For example, if someone asks you, “What is the time?”, “Time please?” or just, “Time?” you understand anyway that he is asking the time but computer is different. Instructions to the computer should be explicitly stated. Computer will tell you the time only if you ask it in the way you have programmed it.

When you're programming, it helps to be able to "think'' as stupidly as the computer does, so that you are in the right frame of mind for specifying everything in minute detail, and not assuming that the right thing will happen by itself.

Comment the code liberally

Always comment the code liberally. The comment statements do not affect the performance of the program as these are ignored by the compiler and do not take any memory in the computer. Comments are used to explain the functioning of the programs. It helps the other programmers as well as the creator of the program to understand the code.

PRINCIPLES OF GOOD PROGRAMMING

Programming principles will help you over the years to become an efficient better programmer that will be able to produce code which is easier to maintain. By following these coding principles, you can save development and maintenance time, and conquer lots of other bottlenecks which generally arise in later development phases.

Writing good code/programs is essential for creating maintainable, efficient, and reliable software. Here are some of the principles of good programming that can help you produce high-quality code:

1. **Readability and Maintainability:**
    - Write code that is easy to understand by humans. Code is read more often than it's written, so prioritize clarity.
    - Use meaningful variable and function names that convey their purpose.
    - Follow consistent indentation and formatting conventions.
2. **Modularity and Reusability**:
    - Break your code into small, modular components (functions, classes, modules) that perform specific tasks.
    - Encapsulate functionality into well-defined units to promote code reuse and easier testing.
3. **Simplicity and KISS Principle:**
    - Keep your code as simple as possible. Avoid unnecessary complexity and overengineering.
    - Follow the "Keep It Simple, Stupid" (KISS) principle to achieve straightforward solutions.
4. **DRY (Don't Repeat Yourself):**
    - Avoid duplicating code. If you find yourself doing the same thing in multiple places, consider refactoring it into a reusable function or class.

5. **Comments and Documentation**:
    - Write clear comments to explain complex logic, assumptions, and any non-obvious parts of your code.
    - Maintain up-to-date documentation to help other developers (including your future self) understand how to use your code.
6. **Testing and Debugging**:
    - Write automated tests to ensure your code behaves as expected and to catch regressions when making changes.
    - Debug systematically by identifying, isolating, and resolving issues using debugging tools and techniques.
7. **Performance and Efficiency**:
    - Optimize critical sections of code for performance, but only after identifying actual bottlenecks through profiling.
    - Avoid premature optimization; focus on writing clear, correct code first.
8. **Consistency and Conventions**:
    - Follow coding standards and conventions consistent with the language or framework you're using.
    - Adhere to established patterns to make it easier for others to understand your code.
9. **Version Control**:
    - Use version control systems (e.g., Git) to track changes and collaborate with other developers effectively.
    - Commit your code in logical, incremental chunks and write meaningful commit messages.
10. **Security:**
    - Be conscious of security best practices, especially when dealing with user inputs and sensitive data.
    - Keep your libraries and dependencies up to date to address known vulnerabilities.
11. **Scalability and Flexibility**:
    - Design your code to be adaptable to changing requirements and scalable to handle increased usage.
12. **Peer Reviews**:
    - Have your code reviewed by peers to catch potential issues, improve code quality, and learn from others.

Remember that these principles are not rigid rules but guidelines to help you make informed decisions while writing code. Balancing these principles according to the context of your project and team dynamics is crucial for producing effective and maintainable software.

WHAT IS A PROGRAM

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

The details look different in different languages, but a few basic instructions appear in just about every language:

- **input**: Get data from the keyboard, a file, the network, or some other device.
- **output**: Display data on the screen, save it in a file, send it over the network, etc.
- **math**: Perform basic mathematical operations like addition and multiplication.
- **conditional execution**: Check for certain conditions and run the appropriate code.
- **repetition**: Perform some action repeatedly, usually with some variation.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

Program Design Recipe

In order to design a program effectively and properly we must have a recipe to follow. In the book name 'How to design programs' by Matthias Felleisen and the co-worker, the idea of design recipe has been stated very elegantly as;

"Learning to design programs is like learning to play soccer. A player must learn to trap a ball, to dribble with a ball, to pass, and to shoot a ball. Once the player knows those basic skills, the next goals are to learn to play a position, to play certain strategies, to choose among feasible strategies, and, on occasion, to create variations of a strategy because none fits. "

The author then continues to say that:

"A programmer is also very much like an architect, a composer, or a writer. They are creative people who start with ideas in their heads and blank pieces of paper. They conceive of an idea, form a mental outline, and refine it on paper until their writings reflect their mental image as much as possible. As they bring their ideas to paper, they employ basic drawing, writing, and playing music to express certain style elements of a building, to describe a person's character, or to formulate portions of a melody. They can practice their trade because they have honed their basic skills for a long time and can use them on an instinctive level.

Programmers also form outlines, translate them into first designs, and iteratively refine them until they truly match the initial idea. Indeed, the best programmers edit and rewrite their programs many times until they meet certain aesthetic standards. And just like soccer players, architects, composers, or writers, programmers must practice the basic skills of their trade for a long time before they can be truly creative. Design recipes are the equivalent of soccer ball handling techniques, writing techniques, arrangements, and drawing skills. "

Hence to design a program properly, we must:

- Analyze a problem statement, typically expressed as a word problem.
- Express its essence, abstractly and with examples.
- Formulate statements and comments in a precise language.
- Evaluate and revise the activities in light of checks and tests and
- Pay attention to detail.

All of these are activities that are useful, not only for a programmer but also for a businessman, a lawyer, a journalist, a scientist, an engineer, and many others.

Let us take an example to demonstrate the use of design recipe:

Suppose we have to develop a payroll system of a company. The company has permanent staff, contractual staff, hourly based employees and per unit making employees. Moreover, there are different deductions and benefits for permanent employees and there is a bonus for per unit making employees and overtime for contractual employees.

We need to analyze the above problem statement. The company has four categories of employees; i.e.; Permanent staff, Contractual staff, hourly based employees and per unit making employees. Further, permanent staff has benefits and deductions depending upon their designation. Bonus will be given to per unit making employees if they make more than 10 pieces a day. Contractual employee will get overtime if they stay after office hours.

Now divide the problem into small segments and calculations. Also include examples in all segments. In this problem, we should take an employee with his details from each category. Let's say, Mr. Ahmad is a permanent employee working as Finance Manager. His salary is ₦200,000 and benefits of medical, car allowance and house rent are ₦40,000 and there is a deduction of ₦12,000. Similarly, we should consider employees from other categories. This will help us in checking and testing the program later on.

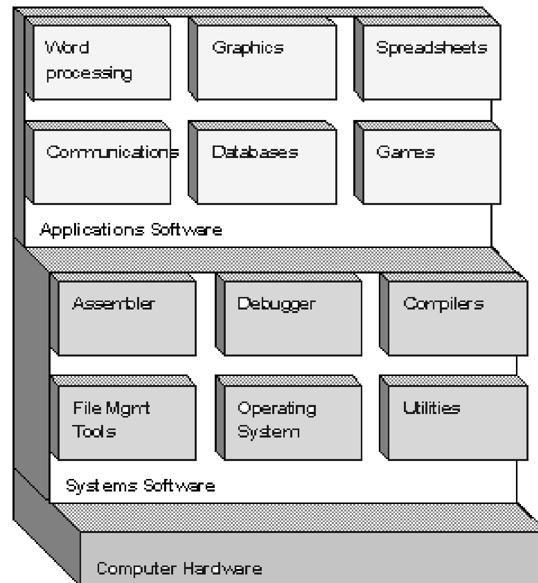
The next step is to formulate these statements in a precise language, i.e. we can use the pseudo code and flowcharting. which will be then used to develop the program using computer language.

Then the program should be evaluated by testing and checking. If there are some changes identified, we revise the activities and repeat the process. Thus, repeating the cycle, we achieve a refined solution.

SOFTWARE CATEGORIES

Software is categorized into two main categories

- System Software
- Application Software



System Software

The system software controls the computer. It communicates with computer's hardware (key board, mouse, modem, sound card etc) and controls different aspects of operations. Sub categories of system software are:

o Operating system

o Device drivers

o Utilities

<u>Operating system</u>

An operating system (sometimes abbreviated as "OS") is the program that manages all the other programs in a computer. It is a integrated collection of routines that service the sequencing and processing of programs by a **computer**. Note: An operating **system** may provide many services, such as resource allocation, scheduling, **input/output control**, and **data management**.

<u>Device drivers</u>

The device driver software is used to communicate between the devices and the computer. We have monitor, keyboard and mouse attached to almost all PC's; if we look at the properties of these devices, we will see that the operating system has installed special software to control these devices. This piece of software is called device driver software. When we attach a new device with the computer, we need software to communicate with this device. These kinds of software are known as

device drivers e.g. CD Rom driver, Sound Card driver and Modem driver. Normally manufacturer of the device provides the device driver software with the device. For scanners to work properly with the computers we install the device driver of the scanner. Nowadays if you have seen a scanner, it comes with TWAIN Drivers. TWAIN stands for Technology Without an Interesting Name.

Utility Software

Utility software is a program that performs a very specific task, usually related to managing system resources. You would have noticed a utility of Disk. Whenever you write a file and save it to the disk, Compression Utility compresses the file (reduce the file size) and write it to the disk and when you request this file from the disk, the compression utility uncompressed the file and shows its contents. Similarly, there is another utility, Disk Defragmentation which is used to defragment the disk. The data is stored on the disks in chunks, so if we are using several files and are making changes to these files then the different portions of file are saved on different locations on the disk. These chunks are linked and the operating system knows how to read the contents of file from the disk combining all the chunks.

Note: **Compilers** and **Interpreters** also belong to the System Software category.

Application software

A program or group of programs designed for end users. For example, a program for Accounting, Payroll, Inventory Control System, and guided system for planes. GPS (global positioning system), another application software, is being used in vehicles, which through satellite determines the geographical position of the vehicle.

PYTHON PROGRAMMING LANGUAGE

Python incorporates the principles of the philosophy that complex tasks can be done in simple ways.

Python was created to have an extremely fast and simple learning curve and development process. As a result, it is considered the most general-purpose programming language since users can work in almost any study domain and still be able to find a useful piece of code for themselves.

In Python, a group of programs for performing various tasks makes up a module (package). At the time of writing, there are over 117,181 modules that have been submitted by an even larger number of developers around the world.

Python is a ***multipurpose, portable, object-oriented, high-level programming*** language that enables an **interactive environment** to code in a minimalistic way.

High-Level Programming: provides facilities like library functions for performing the low-level tasks and also provides ways to define the code in a form readable

to humans, which is then translated into machine language to be fed to a processor. A low-level language is one where users directly feed the machine code to a processor for obtaining results.

Interactive environment: To a large extent, Python derives its philosophy from the ABC language. The syntax structure was largely derived from C and UNIX's Bourne shell environments. The interpretive nature means that Python presents a REPL-based interactive environment to a developer. The interactive shell of the Python programming language is commonly known as REPL (Read-Evaluate-Print-Loops) because it;

- reads what a user types,
- evaluates what it reads,
- prints out the return value after evaluation, and
- loops back and does it all over again.

Object Orientation: Most primitive programming languages were procedural in nature. An object-oriented programming (OOP) language deals with data as an object on which different methods act to produce a desired result. Everything computable is treated as an object. Its nature is defined as its properties. These properties can be probed by functions, which are called methods.

Multipurpose Nature: Python enables developers from different walks of life to use and enrich the language in their fields of expertise. Virtually all fields of computations have used Python. You can define a module specific for one kind of problem. In fact, Python modules exist for specific fields of studies, as shown in Table below; It is impossible to list all the modules for a given application as the modules are being created on a daily basis.

Minimalistic Design: In Python means that it emphasizes code readability. It also provides a syntax structure that allows programmers to express concepts in fewer lines of code than in languages such as C++ and Java.

The core philosophy of the language is summarized by Tim Peters in the document The Zen of Python, which includes the following aphorisms:

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.

13. There should be one— and preferably only one —obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than right now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea—let's do more of those!

Portability: Since Python belongs to an open-source community, it has been ported (that is, adapted to work on) to many platforms so that Python code written on one platform can run without modification on others (except system-dependent features).

Extensibility: Rather than providing all functionalities in its core program, Python's creators designed it to be highly extensible. Users can thus choose to have functionality as per their requirements. For example, if a user needs to work on differential equations, then that user can use a module for differential equations rather than all users having that functionality but never using it.

BRIEF HISTORY OF PYTHON

The development of the Python programming language dates back to the 1980s. Guido van Rossum at CWI in the Netherlands began implementing it in December 1989. This was an era when computing devices were becoming increasingly powerful and reliable with every advancing day. Van Rossum named the language after the BBC TV show Monty Python's Flying Circus. Python 1.0 was released to the public in 1994, Python 2.0 in 2000, and Python 3.0 in 2008. However, Python 3 was not created to be backward compatible with Python 2, which made it less practical for users who were already developing with Python 2. As a result, a lot of developers have continued using Python 2, even now. Nonetheless, the future belongs to Python 3, which has been developed in a more efficient manner. Hence, we will discuss Python-3-based codes in this Course.

TOOLS FOR THE TRADE OF PROGRAMMING

Editors (Text Editor)

First of all, we need a tool for writing the code of a program. For this purpose, we used Editors in which we write our code. We can use word processor too for this, but word processors have many other features like bold the text, italic, coloring the text etc., so when we save a file written in a word processor, lot of other information including the text is saved on the disk. For programming purposes, we don't need these things we only need simple text. Text editors are such editors which save only the text which we type. So, for programming we will be using a text editor.

Compiler and Interpreter

As we write the code in English and we know that computers can understand only ***0***s and ***1***s. We need a translator which translates the code of our program into machine language. There are two kinds of translators which are known as Interpreter and Compilers. These translators translate our program which is written in Python into Machine language. Interpreters translates the program line by line meaning it reads one line of program and translates it, then it reads second line, translate it and so on. The benefit of it is that we get the errors as we go along and it is very easy to correct the errors. The drawback of the interpreter is that the program executes slowly as the interpreter translates the program line by line. Another drawback is that as interpreters are reading the program line by line so they cannot get the overall picture of the program hence cannot optimize the program making it efficient.

Compilers also translate the English like language (Code written in e.g., Java) into a language (Machine language) which computer can understand. The Compiler read the whole program and translates it into machine language completely. **The difference** between interpreter and compiler is that compiler will stop translating if it finds an error and there will be no executable code generated whereas Interpreter will execute all the lines before error and will stop at the line which contains the error. So, Compiler needs syntactically correct program to produce an executable code.

Debugger

Another important tool is Debugger. Every programmer should be familiar with it. Debugger is used to debug the program i.e., to correct the logical errors. Using debugger, we can control our program while it is running. We can stop the execution of our program at some point and can check the values in different variables, can change these values etc. In this way we can trace the logical errors in our program and can see whether our program is producing the correct results.

RUNNING PYTHON

Students should Install **PyCharm Community Edition** for any PC with an OS or Install **Pydroid 3, QPython 3L or Python Interpreter** from the Google Play-store for any Android phone or device. There are many other development environment and ways for running python, so chose what you are much comfortable with.

Our first program: **print ('Hello, Welcome to CSC202 Class')**

Arithmetic operators

Python provides operators, which are special symbols that represent computations.

The operators +, -, and * perform addition, subtraction, and multiplication, as in the following examples: **print (40+20)**, **print (40*20), print (40-20)**, **print (40/20), print (4**2)** // 4 to the power 2.

PHASES OF PROGRAM DEVELOPMENT (PROGRAMMING)

The process of producing a computer program (software) may be divided into eight phases or stages:

[1]. Problem definition/Analysis
[2]. Selection or development of an algorithm
[3]. Designing the program
[4]. Coding the programming statements
[5]. Compiling/Compilation stage
[6]. Testing/Running and Debugging the program
[7]. Documentation.
[8]. Maintenance

[1]. ***Problem Definition/Analysis Stage:*** There is need to understand the problem that requires a solution. The need to determine the data to be processed, form or type of the data, volume of the data, what to be done to the data to produce the expected/required output.

[2]. ***Selection or development of an algorithm:*** An algorithm is the set of steps required to solve a problem written down in English language.

[3]. ***Designing the program:*** In order to minimize the amount of time to be spent in developing the software, the programmer makes use of flowchart. Flowchart is the pictorial representation of the algorithm developed in step 2 above. Pseudocode IPO chart (input processing output) and HIPO (Hierarchical Input Processing and Output) chart may be used in place of flowchart or to supplement flowchart.

[4]. ***Coding the statement:*** This involves writing the program statements. The programmer uses the program flow chart as a guide for coding the steps the computer will follow.

[5]. ***Compiling:*** There is need to translate the program from the source code to the machine or object code if it is not written in machine language. A computer program is fed into the computer first, then as the source program is entered, a translated equivalent (object program) is created and stored in the memory.

[6]. ***Running, Testing and Debugging:*** When the computer is activated to run a program, it may find it difficult to run it because errors (syntax, semantics or logic, or runtime) might have been committed. Manuals are used to debug the errors. A program that is error free is tested using some test data. If the program works as intended, real life data are then loaded.

[7]. ***Documentation:*** This is the last stage in software development. This involves keeping written records that describe the program, explain its purposes, define the amount, types and sources of input data required to run it. List the Departments and people who use its output and trace the logic the program follows.

[8]. ***Maintenance****:* All the activities that occur after the completion of the program come under the program maintenance. Program maintenance includes the following: Finding and correcting the errors; Modifying the program to enhance it – i.e., adapting to some new concepts or when there is a change in the hardware or operating system; Update the documentation; Add new features and functions; Remove useless or redundant parts of code.

NOTE: An **SDLC** (software development life cycle) is a big-picture breakdown of all the steps involved in software creation. This is a higher level of abstraction from the phases for program development.

STRUCTURED PROGRAMMING CONCEPT

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines in contrast to using simple tests and jumps such as the go to statement, which can lead to "spaghetti code" that is potentially difficult to follow and maintain.

Key concepts and principles of structured programming include:

Sequence: Programs are structured as a sequence of statements that are executed in the order they appear. This helps in maintaining a clear and logical flow of control. Sequences allow you to store multiple values in an organized and efficient fashion. There are seven sequence types: strings, bytes, lists, tuples, bytes arrays, buffers, and range objects. Dictionaries and sets are containers for sequential data.

Len (String)

```
>>> len(fruit) = 6
```

String slices

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

```
>>> fruit = 'banana'
>>> fruit[:3] = 'ban'
>>> fruit[3:] = 'ana'
```

Lists

```
# List slicing in Python
my_list = ['p','r','o','g','r','a','m','i','z']
# elements 3rd to 5th
print(my_list[2:5])
```

Dictionaries

```
# get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}
# Output: Jack
print(my_dict['name'])
# Output: 26
print(my_dict.get('age'))
```

Students should learn how to access a file, Open, Read and Write.

Access a Database using Python.

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

```
# Open a file
fo = open("foo.txt", "wb")
print ("Name of the file: ", fo.name)f
print ("Closed or not : ", fo.closed)
print ("Opening mode : ", fo.mode)
fo.close()
Answer: foo.txt, False, wb.
```

Tuples

The important difference is that tuples are **immutable.**

```
>>> 15ulia = ("Julia", "Roberts", 1967, "Duplicity", 2009,
"Actress", "Atlanta, Georgia")
```

```
>>> 15ulia[2] = 1967.
```

Slicing

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[ :4]
['a', 'b', 'c', 'd']
>>> t[3: ]
['d', 'e', 'f']
>>> t[ : ]
['a', 'b', 'c', 'd', 'e', 'f']
```

Adding elements

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

Sorting elements

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Merging two List

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

Deleting element

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

Deleting element cont.

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

Deleting Sliced elements

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

Spliting string into words

```
>>> s = 'pining for the jobs'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'jobs']
```

When you know the element

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

Converting String to List

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Note: All these can work on Strings and List.

Selection (Conditional Statements): Structured programming introduces structured conditional statements like "if-else" and "switch" statements, which allow different paths of execution based on specified conditions. This helps in making the code more readable and understandable.

Boolean expressions

```
x != y # x is not equal to y
x == y # x is equal to y
x > y # x is greater than y
x < y # x is less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```

Floor:

```
>>> minutes = 105
>>> remainder = minutes % 60
>>> remainder
45
```

Logical operators

(n%2 == 0 or n%3 == 0) and, or, not

Conditional and Alternative execution

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

Nested conditionals

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Iteration (Loops): Structured loops, such as "for" and "while" loops, are used to repeat a block of code until a certain condition is met. These loops enhance code readability and prevent the need for duplicated code segments.

Iteration

The **while** statement:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n – 1
    print('Blastoff!')
```

For Loop

```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
```

Ans = **1.73, 1.68, 1.71, 1.89**

Break

```
var = 10
while var > 0:
  print 'Current value :', var
  var = var -1
  if var == 5:
    break

print "Good bye!"
```

Modularity (Functions/Procedures): The concept of breaking down a program into smaller, reusable modules is a fundamental principle of structured programming. These modules, often referred to as functions or procedures, encapsulate specific tasks and can be easily understood, tested, and maintained.

Functions

This is a named sequence of statements that performs a computation. We have already seen one example of a function call:

```
>>> type(42)
<class 'int'>
```

float converts integers and strings to floating-point numbers:

```
>>> float(32)
```

A module is a file that contains a collection of related functions.

```
import math
def my_function():
  print(pow(2, 4))
  print("Hello from a function")
  print(sqrt(x))
my_function()
```

Functions with Arguments: (Need to understand *Local and Global Variable*)

```
def triarea(base, height):
    area = 0.5 * base * height
    return area
triarea(4, 5) = 3
```

Variables Scope:

Local Variables: These are variables which are declared inside the function, compound statement (or block).

Global Variables: The variables declared outside any function are called global variables. They are not limited to any function. Any function can access and modify global variables. Global variables are automatically initialized to 0 at the time of declaration. Global variables are generally written before main() function.

Static Variables: A Static variable is able to retain its value between different function calls. The static variable is only initialized once, if it is not initialized, then it is automatically initialized to 0. Here is how to declare a static variable.

Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Single Entry, Single Exit (SESE) Principle: Each function or module should have a single-entry point (starting point) and a single exit point (ending point). This reduces complexity and makes it easier to reason about the flow of data and control.

Goto Statement Elimination: The "***goto***" statement, which was common in older programming styles, is largely avoided in structured programming. Instead, control structures like loops and conditionals are used to achieve the desired flow of execution. *Used in C and BASIC Programming*.

Clear and Meaningful Names: Variables, functions, and modules should be given meaningful and descriptive names, making the code more self-documenting and easier for other programmers to understand.

Variable names

Programmers generally choose names for their variables that are meaningful—they document what the variable is used for. The normally start with small letters and for variable with more than one name should be joined with an underscore. Example:

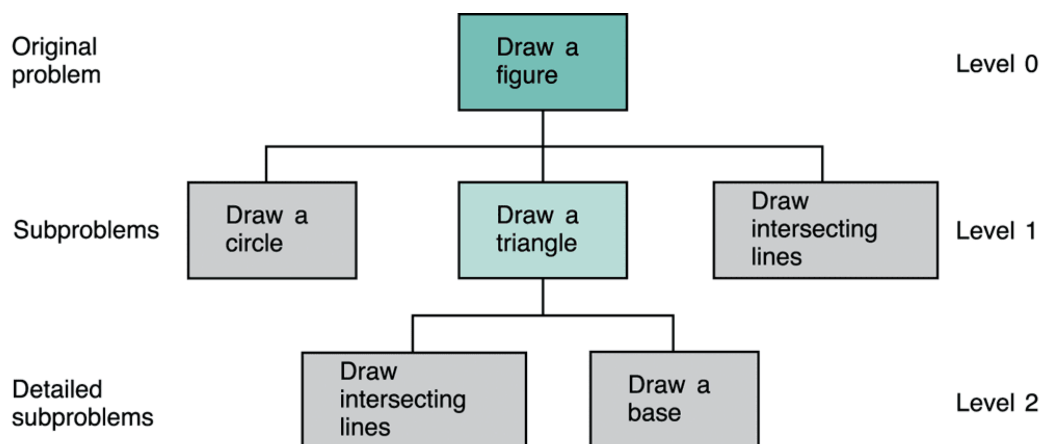
school_boys, entry_value, first_name, address, house_number etc

Note: Do not use Python **Keywords**

Python 3 has these keywords:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Top-Down Design: Structured programming encourages a top-down approach to designing programs, where you start with a high-level overview of the problem and then progressively break it down into smaller, manageable sub-problems. Each sub-problem can then be further decomposed until you reach a level where the implementation becomes straightforward.



Data Abstraction: Structured programming promotes the use of abstract data types and structures to organize and manipulate data. This helps in improving code organization, reusability, and maintainability.

What is ADT in Python?

The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of primitive data types, but operation logics are hidden.

Some examples of ADT are ***Stack***, ***Queue***, ***List*** etc.

Sub-examples of ***Stack*** ADT are; size(), pop(), push() etc in python.

Structured programming has had a significant impact on software development, leading to the development of clearer, more maintainable, and less error-prone code. It laid the foundation for later programming paradigms, such as object-oriented programming (OOP) and procedural programming.

Python String Processing

Although we have worked with strings in previous topics such as List and Tuple, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use single quotes or double quotes to represent a string in Python. For example,

```
# create string type variables
name = "Python"
print(name)
message = "I love Python."
print(message)
```

Output:

Python
I love Python.

Multiline String:	**Compare Two Strings**	**Join Two or More Strings**

message = """
Never gonna give you up
Never gonna let you down
"""
print(message)
 	
Output:
Never gonna give you up
Never gonna let you down
	str1 = "Hello, world!"
	str2 = "I love Python."
	str3 = "Hello, world!"
	
	# compare str1 and str2
	print(str1 == str2)
	
	# compare str1 and str3
	print(str1 == str3)
	Output:
	False
	True
		greet = "Hello, "
		name = "Jack"
		
		# using + operator
		result = greet + name
		print(result)
		
		Output:
		Hello, Jack

Methods of Python String

Besides those mentioned above, there are various string methods present in Python. Here are some of those methods:

Methods	Description
upper()	converts the string to uppercase
lower()	converts the string to lowercase
partition()	returns a tuple
replace()	replaces substring inside
find()	returns the index of first occurrence of substring
rstrip()	removes trailing characters
split()	splits string from left
startswith()	checks if string starts with the specified string
isnumeric()	checks numeric characters
index()	returns index of substring

Recursion in Python

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Syntax:

```
def func(): <--
           |
           | (recursive call)
           |
    func() ----
```

Example:

```
# Program to print factorial of a number recursively.
# Recursive function
def recursive_factorial(n):
  if n == 1:
      return n
  else:
      return n * recursive_factorial(n-1)

# user input
num = 6

# check if the input is valid or not
if num < 0:
  print("Invalid input ! Please enter a positive number.")
elif num == 0:
  print("Factorial of number 0 is 1")
else:
  print("Factorial of number", num, "=", recursive_factorial(num))
```

Output:

Factorial of number 6 = 720

Tail-Recursion:

A unique type of recursion where the last procedure of a function is a recursive call. The recursion may be automated away by performing the request in the current stack frame and returning the output instead of generating a new stack frame. The tail-recursion may be optimized by the compiler which makes it better than non-tail recursive functions.

Values and types

A value is one of the basic things a program works with, like a letter or a number. Some values we have seen so far are 2, 42.0, and 'Hello, World!'.

These values belong to different types: 2 is an integer, 42.0 is a floating-point number, and 'Hello, World!' is a string,

print(type('2')) /*Shows type of value*/

Assignment statements

An assignment statement creates a new variable and gives it a value:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
```

Expressions and statements

An **expression** is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions:

```
>>> 42
42
>>> n
17
>>> n + 25
```

A **statement** is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

Order of operations

Python follows mathematical convention using the acronym **PEMDAS**.

Parentheses, Exponentiation, Multiplication, Division**, **Addition** and **Subtraction. E.g. 2*(3-1) = 4.

DEBUGGING AND TESTING

Debugging and testing are crucial components of the software development process. In Python, these practices play a vital role in ensuring the reliability, functionality, and quality of your code. In this blog, we will delve into the world of debugging and testing in Python, exploring effective strategies, tools, and

techniques to identify and fix bugs, as well as validate the correctness of your code through comprehensive testing.

Debugging:

Programmers make mistakes. For whimsical reasons, programming errors are called bugs and the process of tracking them down is called debugging.

Programming, and especially debugging, sometimes brings out strong emotions. If you are struggling with a difficult bug, you might feel angry, despondent, or embarrassed.

Types of errors for debugging:

Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. This error occurs when the rules and structure are wrong.

Runtime error: The error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

Semantic error: related to meaning. Program will run without generating error messages, but not the desired result will display.

Debugging is the process of identifying and resolving issues, or bugs, within your code. These bugs can cause unexpected behaviors, crashes, or incorrect results. Python provides powerful debugging tools and techniques that help developers trace and analyze code execution, isolate problems, and rectify errors efficiently.

Debugging tools and techniques;

- **Printing and Debugging Statements**

  One of the simplest and most effective debugging techniques is using print statements to display variable values, control flow, and intermediate results during program execution. By strategically placing print statements at critical points in your code, you can gain insights into its behavior and identify potential issues.

- **Debugging with pdb**:

  Python's built-in debugger, pdb, provides a more advanced approach to debugging. It allows you to set breakpoints, step through code execution, inspect variables, and analyze stack traces. By utilizing pdb, you can interactively debug your code, gaining a deeper understanding of its behavior and resolving complex issues.

- **Logging**:

  Logging is another powerful debugging technique that allows you to capture and record runtime information for analysis. The logging module in Python provides a flexible and configurable logging framework, enabling you to log

messages with various levels of severity. By strategically logging relevant information, you can track the execution flow and pinpoint problematic areas in your code.

Testing:

Testing is a critical process in software development that aims to validate the correctness and reliability of your code. Python offers a variety of testing frameworks and tools that simplify the testing process and promote good testing practices.

- ***Unit Testing with unittest***:
  The unittest module in Python provides a comprehensive framework for writing and running unit tests. Unit tests focus on testing individual units of code, such as functions or methods, in isolation. By creating unit tests, you can verify the behavior and correctness of specific code units, ensuring they produce the expected results.
- **Test Driven Development (TDD):**
  Test Driven Development is an iterative development process that involves writing tests before writing the actual code. With TDD, you start by creating a test that describes the desired functionality, run the test (which fails initially), and then write the code to make the test pass. This approach promotes better code design, modularity, and test coverage.
- **Integration and System Testing**:
  In addition to unit tests, integration and system tests are crucial for verifying the interactions and behavior of components within a larger system. Integration tests focus on testing the integration points between different modules or components, while system tests verify the functionality and behavior of the complete system as a whole.
- **Test Coverage Analysis:**
  Test coverage analysis measures the extent to which your tests cover the codebase. It helps identify areas of your code that are not adequately tested, highlighting potential gaps in test coverage. By analyzing test coverage metrics, you can ensure that critical sections of your code are thoroughly tested, reducing the risk of undetected bugs.
- **Test Automation:**
  Automating your tests using frameworks like pytest or nose can significantly streamline the testing process. Automation allows you to run tests in a repeatable and consistent manner, saving time and effort. Additionally, it facilitates continuous integration and deployment practices, ensuring that your code remains reliable and bug-free throughout the development cycle.

GROUP CLASS EXERCISE

AREA OF CIRCLE

```
from math import pi
radius = float(input ("Input the radius of the circle : "))
print ("The area of the circle with radius " + str(radius) + " is: "
+ str(pi * radius**2))
```

PYTHON PROGRAM TO CALCULATE GRADE OF STUDENT

```
print("Enter Marks Obtained: ")
attendanceMark = int(input())
assignmentMark = int(input())
testMark = int(input())
examMark = int(input())

total = attendanceMark + assignmentMark + testMark + examMark
# The Total cannot be higher than 100.
If total>=70 and total<=100:
    print("Your Grade is A")
elif total>=60 and total<70:
    print("Your Grade is B")
elif total>=50 and total<60:
    print("Your Grade is C")
elif total>=45 and total<50:
    print("Your Grade is D")
elif total>=0 and total<45:
    print("Your Grade is F")
else:
    print("Invalid Input!")
```

FIND AVERAGE OF N NUMBERS USING FOR LOOP

```
print("Enter the Value of n: ") # Enter number of Values
n = int(input())
print("Enter " +str(n)+ " Numbers: ") #Enter Values
nums = []
for i in range(n):
    nums.insert(i, int(input()))

sum = 0
for i in range(n):
    sum = sum+nums[i]

avg = sum/n
print("\nAverage = ", avg)
```

PROGRAMMING PARADIGMS

A programming paradigm provides for the programmer the means structure for the execution of a program. A concept by which the methodology of a programming language adheres to. Paradigms are important because they define a programming language and how it works. A great way to think about a paradigm is as a set of ideas that a programming language can use to perform tasks in terms of machine-code at a much higher level.

Common Language paradigms are:

1. Imperative paradigm/languages.
2. Declarative paradigm/languages.
3. Functional paradigm/languages.
4. Object Oriented paradigm/languages.
5. Procedural paradigm/languages.
6. Logic paradigm/languages.
7. Rule- Based paradigm/languages.

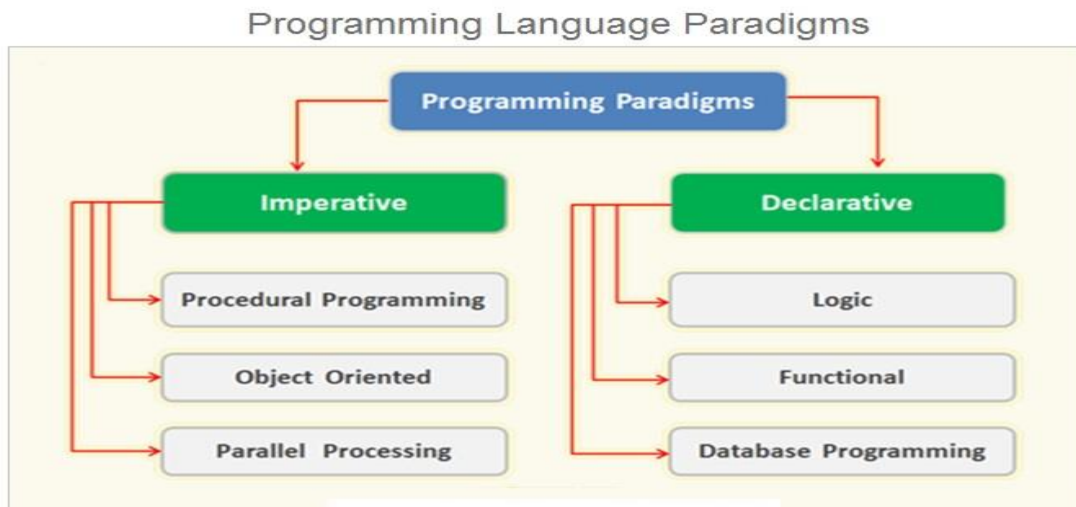


***Figure 2**: Common programming language Paradigms.*

<u>Object Oriented Programming?</u>

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors 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.

An **Object** is an instance of a class.

__init__() sets the **initial state** of the object by assigning the values of the object's properties. That is, .__init__() initializes each new instance of the class.

It is common for the parameters of __init__ to have the **same names as the attributes.** The statement e.g. ***self.hour = hour*** stores the value of the parameter hour as an attribute of **self**.

```
class Dog:
    species = "Canis family"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```
>>> miles = Dog("Miles", 4)
>>> miles.description()
'Miles is 4 years old'
>>> miles.speak("Woof Woof")
'Miles says Woof Woof'
>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

The Four (4) Pillers of Object-oriented programming (OOP)

There are 4 major principles that make a language Object Oriented. These are;

1. *Encapsulation,*
2. *Abstraction,*
3. *Polymorphism and,*
4. *Inheritance.*

ENCAPSULATION: In an Object-Oriented program, you can restrict access to methods and attributes of a certain class. This prevents accidental modification of data and unwanted changes and is known as encapsulation.

In Python, we can create private attributes and methods using __. These methods are not accessible outside of the class (partly true - read more here). We can test to see if a method is private by trying to access it:

```
Class PrivateMethod:
    def __init__(self):
        self.__attr = 1
    def __printhi(self):
        print("hi")
    Def printhello(self):
        print("hello")

a = PrivateMethod()
a.printhello()
a.__printhi()
a.__attr
```

Output:

```
Hello
AttributeError: 'PrivateMethod' object has no attribute '__printhi'
AttributeError: 'PrivateMethod' object has no attribute '__attr'
```

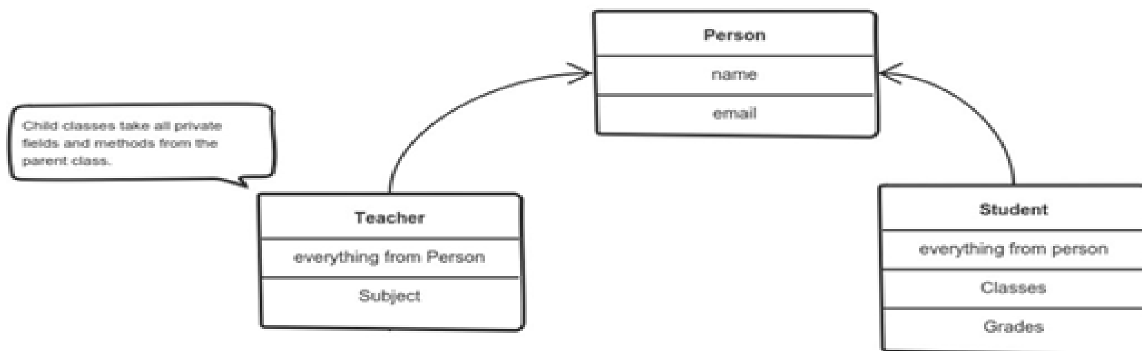
ABSTRACTION: This principle is less physical (programmable) but more logical. Abstraction can be thought of as a natural extension of encapsulation.

At some point, our programs become very large, with various different classes that are using each other. In order to keep our program as logical and simple as possible—we apply abstraction to our classes. This means that each class should only expose its mechanisms that need to be public.

Let’s take a look at an object you’re probably familiar with: A Blendar. All we need to do is press a button, and voilà! The grinded stuff comes out. We aren’t interested in all the noises and processes happening in the back of the machine, we just want our coffee.

Same thing with a class! All the 'behind the scenes' mechanisms should be encapsulated and left private, thus creating less confusion when dealing with many classes at once.

INHERITANCE: A lot of the times our classes will be interacting with each other, and we might even want a specific class to be a subclass of some other class.




```
# parent class
class Bird:

    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
        print("Bird")

    def swim(self):
        print("Swim faster")

# child class
class Penguin(Bird):

    def __init__(self):
        # call super() function
        super().__init__()
    print("Penguin is ready")

    def whoisThis(self):
        print("Penguin")

    def run(self):
        print("Run faster")
```

Continues…

```
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

OUTPUT

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

Additionally, we use the **super() function** inside the __init__() method. This allows us to run the __init__() method of the parent class inside the child class.

POLYMORPHISM: This is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However, we could use the same method to color any shape. This concept is called Polymorphism.

class **Parrot:**
    def fly(self):
        print("Parrot can fly")
    def swim(self):
        print("Parrot can't swim")
 	
class **Penguin:**
    def fly(self):
        print("Penguin can't fly")
    def swim(self):
        print("Penguin can swim")
	# common interface
	def flying_test(bird):
	    bird.fly()
	
	#instantiate objects
	blu = Parrot()
	peggy = Penguin()
	
	# passing the object
	flying_test(blu)
	flying_test(peggy)
	
	OUTPUT

	Parrot can fly Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a **common fly() method**. However, their functions are different.

To use polymorphism, we created a **common interface i.e flying_test()** function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying_test() function, it ran effectively.