MODULE 1

HISTORY OF PYTHON

- Python laid its foundation in the late 1980s.
- The implementation of Python began in **December 1989** (during Christmas vacations) by **Guido Van Rossum** at **CWI** (Centrum Wiskunde and Informatica) in the **Netherlands** as a successor to **ABC Language** capable of **exception handling** and **interfacing with the Amoeba Operating System**.

  Note:
  - An **exception** is an **unwanted or unexpected event**, which **occurs during the execution of a program** i.e., at **run time**, that **disrupts the normal flow of the programs instructions**.
  - An **exception handler** is a **code that tells what a program will do when exception occurs**.
- **CWI** (National Research Institute for Mathematics and Computer Science) is a **research center** in the field of **mathematics and theoretical computer science**. It is part of the **Netherlands Organization for Scientific Research** (NWO) and is located at the **Amsterdam Science Park**.
- **ABC** is a **programming language** developed at **CWI**, **Netherlands** by **Leo Geurts**, **Lambert Meertens** and **Steven Pemberton**.
  - **Interactive**, **structure**, **high-level** and intended to be used instead of BASIC, Pascal or AWK.
  - It had a **major influence** on the design of **Python Programming Language**.
  - **Features:**
    - ✓ Only five basic data types.
    - ✓ Does not require variable declarations.
    - ✓ Explicit support for top-down programming.
    - ✓ Statement nesting is indicated by indentation.
    - ✓ Infinite arithmetic precision, unlimited-sized lists and strings.
  - **Disadvantages:**
    - ✓ **Inability** to **adapt** to **new requirements**, such as creating **graphical user interface**.
    - ✓ **Inability** to **directly access** the underlying **file system** and **operating system**.
- **Amoeba** is a **distributed operating system** developed by **Andre S Tanenbaum** and others at the **Vrije Universiteit Amsterdam**.
  - Aim was to build a **timesharing system** that makes an entire **network of computers appear to the user as a single machine**.
  - **Amoeba network consists of a number of workstations** (scientific applications) **connected to a pool of resources**.
  - **Executing** a **program** from a **terminal causes it to run on any of the available processors**, with the operating system **providing load balancing**.
- **Python** was named after the **BBC TV Show Monty Python's Flying Circus** (45 episodes airing over 4 series from 1969-1974).

- In **February 1991**, Van Rossum published the code (labelled **version 0.9.0**)
- In **1994**, **Python 1.0** was released with **new features** like **lamda, map, filter** and **reduce**.
- In **2000**, the **entire Python team moved from CNRI** (Corporation for National Research Initiatives) **to BeOpen.com** (an open source software startup) **to form BeOpen Python apps team**. Before this they were **working** on **version 1.5**.
- **CNRI** requested **that version 1.6** be released, **summarizing the development** to the point where team left.
- Two versions were being developed simultaneously
  - **CNRI – Python 1.6**
  - **BeOpen.com – Python 2.0 (16th Oct 2000)**
- With **Python 2.0**, it became an **open source project**. It was **later moved to source forge giving more access to users** and providing easy way to simulate patches and bugs.
- **Python 2.0** added **new features** like: **list comprehensions, garbage collection system**.
- With **version 2.1**, they moved to **digital creation part of Zope**, an **object oriented web application server rendering program**.
- Major revision was **Python 3.0** (sometimes called Python 3000 or 3k). It **wasn't backward compatible with the Python 2.0**. It was released on **3rd Dec 2008**.

1 WHY SHOULD YOU LEARN TO WRITE PROGRAMS

- Writing programs is a very creative and rewarding activity.
- Reasons to learn programming:
  - Solving or helping in solve a difficult problem
  - To have fun
  - Helping someone solve a problem
  - Improves problem solving and logical thinking
  - To automate repetitive tasks
  - Job prospects are more

1.1 COMPUTER HARDWARE ARCHITECTURE

When you try to look inside a computer you will find the following parts:

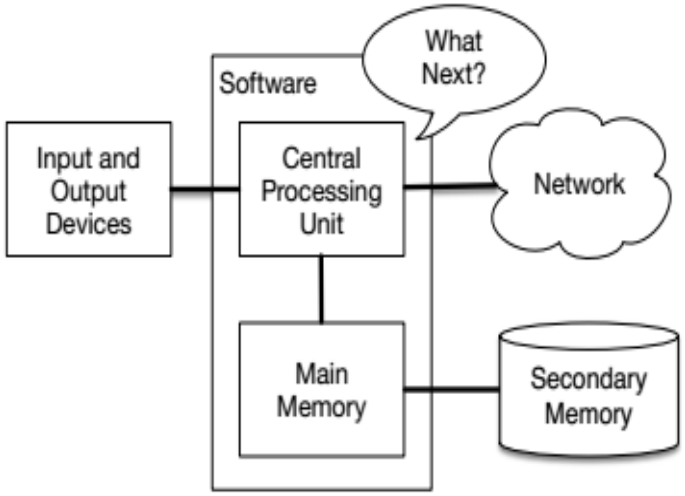


Fig: Hardware Architecture

The high-level definitions of these parts are as follows:

- **Central Processing Unit (CPU)**: is the unit (heart of the computer) which performs most of the processing inside a computer ("what is next?"). It is basically used to control instructions and data flow to and from other parts of the computer.
- **Main Memory**: Also known as Random Access Memory (RAM) or Primary Memory. It is used to store information that the CPU needs in a hurry (currently). The main memory is nearly as fast as CPU. But the information stored in the main memory is lost (volatile) when the computer is turned off.
- **Secondary Memory**: is used to store information permanently. It is much slower than the main memory. The information stored in the secondary memory is not lost (non-volatile) when the computer is turned off. Examples: Disk drives, flash memory (typically found in USB stick and portable music players).

- **Input and Output Devices**: are screen (monitor), keyboard, mouse, microphone, speaker, touchpad, etc. These are all the ways in which we can interact with the computer.
- Most computers also have **Network Connection** to retrieve data/information over a network. We can think of a network as a very slow place that stores and retrieves data that might not always be "up". This means that the network is slower and at times unreliable form of Secondary Memory.

1.2 INTERPRETER AND COMPLIER

- The actual hardware inside the CPU does not understand any of the high-level languages.
- The CPU understands only **machine language**. It's nothing but **zero's (0's)** and **one's (1's)**.

```
00101000111010010010101000000111
11100110000011101010010101101101
...
```

- It is very difficult and complex for the programmers to write in machine language. Instead we use translators that converts the program from high-level language to machine language for actual execution by the CPU.
- Since **machine language** is tied to the computer hardware, machine language is **not portable** across different types of hardware.
- Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.
- The programming language translators can be categorized into two:
  i. **Interpreter**
  ii. **Compiler**

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

- Even though we give commands into Python one line at a time, Python treats them as an ordered sequence of statements with later statements able to retrieve data created in earlier statements.
- The **Python interpreter is written in a high-level language called "C"**.
- **Python is a program itself and is compiled into machine code.**
- When we install Python on the computer, we copy a machine-code copy of the translated Python program onto the system.

1.3 THE BUILDING BLOCKS OF PROGRAMS

- **A program is a sequence of Python statements that is intended to do something.**
- We use some general constructs while writing a program. These constructs are not just for Python programs, they are part of every programming language from machine language to high-level languages.
  - **Input**: Get data from "outside world". It can be reading data from a file or some kind of sensor like a microphone or GPS. Generally input will be given from the user through keyboard.
  - **Output**: Display the result of the program on the screen or store them in a file or write them to a device like a speaker to play music or speak text.
  - **Sequential Execution**: Statements are executed one after the other in the order they appear in the script.
  - **Conditional Execution**: Check for certain conditions and based on that either execute or skip a sequence of statements.
  - **Repeated Execution**: Repeat some set of statements, usually with some variation.
  - **Reuse**: Write a set of instructions/statements once and give them a name, then reuse those instructions in the program whenever needed.

1.4 WHAT COULD POSSIBLY GO WRONG?

As the programs become more sophisticated, we will encounter three general types of errors:

- **Syntax Errors**: A syntax error means that you have violated the "grammar" rules of Python.

OR

Syntax refers to the structure of a program or the rules about that structure.

Python points right at the line and character where the error occurred. Sometimes the mistake that needs to be fixed is actually earlier in the program than where python noticed. So the line and character that Python indicates in a syntax error may just be a starting point of our investigation. The syntax errors are easy to fix.

Examples: missing colon, commas or brackets, misspelling a keyword, incorrect indentation and so on.

- **Logic Errors**: A logic error is when the program is syntactically correct but there is a mistake in the order of statements or mistake in how the statements relate to one another. It causes the program to operate incorrectly. It produces unintended or undesired output. These are difficult to fix.
  Examples: indenting a block to a wrong level, using integer division instead of floating-point division, wrong operator precedence and so on.
- **Semantic Errors**: An error in a program that makes it to do something other than what the programmer intended.

OR

A semantic (meaning) error is when the program is syntactically correct and in the right order, but there is simply a mistake in the program.
Example: Colourless green ideas sleep furiously – This is grammatically correct but cannot be combined into one meaning.

OR

A semantic error is a violation of the rules of meaning of a natural language or a programming language.

Summary

- ✓ **Syntax relate to spelling and grammar.**
  If the code fails to execute due to typos, invalid names, a missing parenthesis or some other grammatical flaw, you have a syntax error.
- ✓ **Logic relate to program flow.**
  If the syntax is correct but a piece of code is (inadvertently) never executed, operations are not done in the correct order, the operation itself is wrong or code is operating on the wrong data, you have a logical error. Using a wrong conditional operator is a common example, so is inadvertently creating an infinite loop or mixing up (valid) names of variables or functions.
- ✓ **Semantics relate to meaning and context.**
  If both your program logic and syntax is correct so the code runs as intended, but the result is still wrong: you likely have a semantic error.

1.5 GLOSSARY

bug	An error in a program.
central processing unit	The heart of any computer. It is what runs the software that we write; also called “CPU” or “the processor”.
compile	To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.
high-level language	A programming language like Python that is designed to be easy for humans to read and write.
interactive	A way of using the Python interpreter by typing commands and expressions at

mode	the prompt.
interpret	To execute a program in a high-level language by translating it one line at a time.
low-level language	A programming language that is designed to be easy for a computer to execute; also called “machine code” or “assembly language”.
machine code	The lowest-level language for software, which is the language that is directly executed by the central processing unit (CPU).
main memory	Stores programs and data. Main memory loses its information when the power is turned off.
parse	To examine a program and analyze the syntactic structure.
portability	A property of a program that can run on more than one kind of computer.
print function	An instruction that causes the Python interpreter to display a value on the screen.
problem solving	The process of formulating a problem, finding a solution, and expressing the solution.
program	A set of instructions that specifies a computation.
prompt	When a program displays a message and pauses for the user to type some input to the program.
secondary memory	Stores programs and data and retains its information even when the power is turned off. Generally slower than main memory. Examples of secondary memory include disk drives and flash memory in USB sticks.
semantics	The meaning of a program.
semantic error	An error in a program that makes it do something other than what the programmer intended.
source code	A program in a high-level language.

1.6 INTRODUCTION TO PYTHON

- **Python** is a **general purpose, dynamic** (at runtime. Executes many common programming behaviour)**, high level, object oriented and interpreted programming language**.
- It supports **Object Oriented programming** (concept of **objects** (variable, a data structure, a function, or a method, and as such, is a location in memory having a value and referenced by an identifier) which may **contain data in the form of fields** (**attributes**) and **code** (**methods**)) and approach to develop applications.
- It is **simple and easy to learn** and **provides lots of high-level data structures**.
- Python is **easy to learn yet powerful and versatile scripting language** which makes it attractive for **Application Development**.

  Note:

  The main difference between the programming language and scripting language is that **the scripting language does not create any binary files (executables) and no memory will be allocated**. For programming languages on compilation make binaries (either executables or libraries). These binaries executes from system's memory. Scripting language is a very limited, high-level language that is application-specific.
- Python's **syntax and dynamic typing** (the property of a language where type checks are performed mostly at run time) with its **interpreted nature**, **makes it an ideal language for scripting and rapid application development**.
- Python supports **multiple programming pattern**, including **object oriented, imperative** and **functional or procedural programming styles**.
- Python is not intended to work on special area such as web programming. That is why it is known as **multipurpose** because it can be used with web, enterprise, 3D CAD etc.
- We **don't need to use data types to declare variable** because it is **dynamically typed** so **we can write a=10 to assign an integer value in an integer variable**.
- Python makes the **development and debugging fast because there is no compilation step included in python development and edit-test-debug cycle is very fast**.

NOTE:

DIFFERENCE BETWEEN LOW LEVEL LANGUAGE AND HIGH LEVEL LANGUAGE

Low Level Language	High Level Language
Faster	Comparatively slower
Memory efficient	Not memory efficient
Requires additional knowledge of the computer architecture	Does not require any additional knowledge of the computer architecture
Machine dependent and are not portable	Machine independent and portable
Less or no abstraction from hardware	High abstraction from the hardware
More error prone	Less error prone
Debugging and maintenance is difficult	Debugging and maintenance is comparatively

	easier
Used for developing system software's (OS) and embedded applications	Used for developing a variety of applications such as – desktop applications, websites, mobile software's

DIFFERENCE BETWEEN SCRIPTING LANGUAGE AND PROGRAMMING LANGUAGE

- All scripting languages are **programming languages**.
- The theoretical difference between the two is that **scripting languages do not require the compilation step and are rather interpreted**. For example, normally, a **C program** needs to be **compiled** before running whereas normally, a **scripting language like JavaScript or PHP need not be compiled**.
- **Compiled programs run faster than interpreted programs because they are first converted native machine code.** Also**, compilers read and analyze the code only once**, and **report the errors collectively that the code might have**, but **the interpreter will read and analyze the code statements each time it meets them and halts at that very instance if there is some error**.
- Another point to be noted is that while classifying a language as scripting language or programming language, the environment on which it would execute must be taken into consideration. The reason why this is important is that **we can design an interpreter for C language and use it as a scripting language**, and at the same time, **we can design a compiler for JavaScript and use it as a programming language**. A live example of this is **V8, the JavaScript engine of Google Chrome, which compiles the JavaScript code into machine code, rather than interpreting** it.
- **Examples:**
  Scripting Languages (without an explicit compilation step) - JavaScript, PHP, Python, VBScript
  Programming Languages (with an explicit compilation step) - C, C++

Applications of Scripting Languages:

- To automate certain tasks in a program
- Extracting information from a data set
- Less code intensive as compared to traditional programming languages

Applications of Programming Languages:

- They typically run inside a parent program like scripts
- More compatible while integrating code with mathematical models
- Languages like JAVA can be compiled and then used on any platform

TYPE CHECKING

- The existence of types is useless without a process of verifying that those types make logical sense in the program so that the program can be executed successfully. This is where type checking comes in.
- **Type checking is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically)**. Type checking is all about ensuring that the program is type-safe, meaning that the possibility of type errors is kept to a minimum.
- A type error is an erroneous program behavior in which an operation occurs (or trys to occur) on a particular data type that it's not meant to occur on.

  For example: There may be a situation where an operation is performed on an integer with the intent that it is a float, or even something such as adding a string and an integer together:

  X = 1 + "2"

  While in many languages both strings and integers can make use of the **+** operator, this would often result in a type error because this expression is usually not meant to handle multiple data types.

STATIC TYPING vs DYNAMIC TYPING

STATIC TYPING

- **Static typed programming languages are those languages in which variables must necessarily be defined before they are used**. This implies that static typing has to do with the **explicit declaration** (or initialization) of variables before they're employed. Java is an example of a static typed language; **C, C++, C#, JADE, Java, Fortran, Haskell, ML, Pascal, and Scala are also static typed languages**.

  Note: In C (and C++ also), variables can be cast into other types, but they don't get converted; you just read them assuming they are another type.
- Static typing does not imply that you have to declare all the variables first, before you use them; variables maybe be initialized anywhere, but developers have to do so before they use those variables anywhere. Consider the following example:

```
/* C code */
int num, sum; // explicit declaration
num = 5; // now use the variables
sum = 10;
sum = sum + num;
```

DYNAMIC TYPING

- **Dynamic typed programming languages are those in which variables need not be defined before they're used**. This implies that dynamic typed languages **do not require the explicit declaration of the variables** before they're used. **Python** is an example of a dynamic typed programming language, and so is **PHP, Groovy, JavaScript, Lisp, Lua, Objective-C, Prolog, Ruby, Smalltalk and Tcl**. Consider the following example:

```
/* Python code */
num = 10 // directly using the variable
```

1.7 PYTHON FEATURES

- Python's features include:
  - **Easy to Learn and Use:** Python is easy to learn and use. It is developer-friendly and high level programming language.
  - **Easy to read**: Python language is more expressive means that it is more understandable and readable.
  - **Interpreted Language**: Python is an interpreted language i.e. interpreter executes the code line by line at a time. This makes debugging easy and thus suitable for beginners.
  - **Cross-platform Language**: Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc. So, we can say that Python is a portable language.
  - **Free and Open Source**: Python language is freely available at official web address. The source-code is also available. Therefore it is open source.
  - **Object-Oriented Language**: Python supports object oriented language and concepts of classes and objects come into existence. With OOP, you are able to divide these complex problems into smaller sets by creating objects.
  - **Interactive Mode** – Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
  - **Extensible**: It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our python code. Suppose an application requires high performance. You can easily combine pieces of C/C++ or other languages with Python code.
  - **Large Standard Library**: Python has a large and broad library and provides rich set of module and functions for rapid application development.
  - **GUI Programming Support**: Graphical user interfaces can be developed using Python.
  - **Databases**: Python provides interfaces to all major commercial databases.
  - **Integrated**: It can be easily integrated with languages like C, C++, JAVA etc.
  - **Scalable**: Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below −

- It supports **functional and structured programming methods as well as OOP**.
- It can be used as a **scripting language or can be compiled to byte-code for building large applications.**
- It provides very **high-level dynamic data types and supports dynamic type checking**.
- It supports **automatic garbage collection**.
- It can be **easily integrated with C, C++, COM, ActiveX, CORBA, and Java**.

1.8 PYTHON APPLICATIONS

➢ **Web Applications**

Python has been used to create a variety of web-frameworks including CherryPy, Django, TurboGears, Bottle, Flask etc. These frameworks provide standard libraries and modules which simplify tasks related to content management, interaction with database and interfacing with different internet protocols such as HTTP, SMTP, XML-RPC, FTP and POP. Plone, a content management system; ERP5, an open source ERP which is used in aerospace, apparel and banking; Odoo – a consolidated suite of business applications; and Google App engine are a few of the popular web applications based on Python.

➢ **Desktop GUI Applications**

Python has simple syntax, modular architecture, rich text processing tools and the ability to work on multiple operating systems which make it a desirable choice for developing desktop-based applications. There are various GUI toolkits like wxPython, PyQt or PyGtk available which help developers create highly functional Graphical User Interface (GUI). The various applications developed using Python includes:

- **Image Processing and Graphic Design Applications**:

  Python has been used to make 2D imaging software such as Inkscape, GIMP, Paint Shop Pro and Scribus. Further, 3D animation packages, like Blender, 3ds Max, Cinema 4D, Houdini, Lightwave and Maya, also use Python in variable proportions.

- **Scientific and Computational Applications**:

  The higher speeds, productivity and availability of tools, such as Scientific Python and Numeric Python, have resulted in Python becoming an integral part of applications involved in computation and processing of scientific data. 3D modeling software, such as FreeCAD, and finite element method software, such as Abaqus, are coded in Python. Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

- **Games**:

  Python has various modules, libraries and platforms that support development of games. For example, PySoy is a 3D game engine supporting Python 3, and PyGame

provides functionality and a library for game development. There have been numerous games built using Python including Civilization-IV, Disney's Toontown Online, Vega Strike etc.

- **Software Development**

  Python's design and module architecture has influenced development of numerous languages. Boo language uses an object model, syntax and indentation, similar to Python. Further, syntax of languages like Apple's Swift, CoffeeScript, Cobra, and OCaml all share similarity with Python.

- **Enterprise and Business Applications**

  With features that include special libraries, extensibility, scalability and easily readable syntax, Python is a suitable coding language for customizing larger applications. Reddit, which was originally written in Common Lips, was rewritten in Python in 2005. Python also contributed in a large part to functionality in YouTube.

  Python is used to build Bussiness applications like ERP and e-commerce systems. Tryton is a high level application platform.

  Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

- **Console Based Application**

  We can use Python to develop console based applications. For example: **IPython**.

- **Audio or Video based Applications**

  Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

- **3D CAD Applications**

  To create CAD application Fandango is a real application which provides full features of CAD.

- **Operating Systems**

  Python is often an integral part of Linux distributions. For instance, Ubuntu's Ubiquity Installer, and Fedora's and Red Hat Enterprise Linux's Anaconda Installer are written in Python. Gentoo Linux makes use of Python for Portage, its package management system.

1.9 PYTHON INSTALLATION

- **In Windows:**

The steps to install Python on Windows machine:

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer *python-XYZ.msi* file where XYZ is the version you need to install.
- To use this installer *python-XYZ.msi*, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Setting path at Windows

To add the Python directory to the path for a particular session in Windows:

At the command prompt – type path %path%;C:\Python and press Enter.
Note – C:\Python is the path of the Python directory

- **In UNIX/LINUX:**

The steps to install Python on Unix/Linux machine:

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the *Modules/Setup* file if you want to customize some options.
- run ./configure script
- make
- make install

This installs Python at standard location */usr/local/bin* and its libraries at */usr/local/lib/pythonXX* where XX is the version of Python.

Setting path at Unix/Linux

To add the Python directory to the path for a particular session in Unix:

- **In the csh shell** – type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- **In the bash shell (Linux)** – type export ATH="$PATH:/usr/local/bin/python" and press Enter.
- **In the sh or ksh shell** – type PATH="$PATH:/usr/local/bin/python" and press Enter.
- **Note** – /usr/local/bin/python is the path of the Python directory

1.10 WRITING A PROGRAM

- **A program is a set of instructions that specifies a computation.**

OR

A program is a sequence of Python statements that does some specific task.

- **Python interpreter is not recommended for solving complex problems**.
- To write a program, we **need a text editor to write the Python instructions into a file, which is called script**.
- **To execute the script, we need to specify the name of the file to the Python interpreter**. In UNIX or Windows command window we need to type **python filename.py**

Note: You can bring up an interactive help system using help().

- **>>> help()**

  Welcome to Python 3.6's help utility!

  If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

  Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

  To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

- **help> keywords**
- **help> symbols**
- **help> topics**

  help> LOOPING → **press q (quit to come out from that particular topic)**

  help> METHODS

  - **help> modules**

    help> math

For Example:

Use help(function) to get the details about that particular function in the python interpreter.

>>> help(print)
Help on built-in function print in module builtins:

print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.

print() in Python

- print() is a function in Python that is used to print values, variables, strings.
- **For Example:**
  - **If u want to print Hello World on the screen.**

```
>>> print('hello world')
hello world
```

  - **If u want to print It's a beautiful day on the screen.**

```
>>> print('It's a beautiful day')
  File "<stdin>", line 1
    print('It's a beautiful day')          → This gives u error
            ^
SyntaxError: invalid syntax
```

Instead use " " to print it.

```
>>> print("It's a beautiful day")
It's a beautiful day
        OR
>>> print('It\'s a beautiful day')        → Use escape character
It's a beautiful day
```

  - **If you want to print any values**

```
>>> print(3)
3
>>> print(23.5)
23.5
>>> print(-12)
-12
```

  - **>>> print('\n')**

```
>>>
```

  - **>>> print(5*'\n')**

```
>>>
```

  - **>>> print(6+4)**
    10

Note: The strings in the print statements are enclosed in quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

2 VARIABLES, EXPRESSIONS AND STATEMENTS

2.1 VALUES AND TYPES

- **A value is one of the basic things a program works with, like a letter/character or a number**.
  Eg: 1, 5.5, "Hello"
- The values belong to different **types**:
  - **1 is an integer**
  - **5.5 is float**
  - **"Hello" is a string** (contains string of letters/characters).
- If you want to know what type a value belongs to, use python interpreter as below:
  - **>>> type(1)**
    <class 'int'> → **integers belong to type int.**
  - **>>> type(5.5)**
    <class 'float'> → **decimal point belong to type float.**
  - **>>> type('Hello')**
    <class 'str'> → **strings belong to type str.**
  - **>>>type('15')**
    <class 'str'> → **These are treated as strings.**
    >>>type('3.2')
    <class 'str'>
- Whenever we write large integers, we generally use commas (,) between a group of three or two digits. For example: 1,000,000. This is **not a valid integer in Python**, but it is valid

i. **>>> print(1,000,000)**
1 0 0
Python interprets 1,000,000 as a comma separated sequence of integers, which it prints with spaces between.

ii. **>>> print(98,058,576,03948)**
File "<stdin>", line 1
print(98,058,576,03948)
^
SyntaxError: invalid token

iii. **>>> print(852,48,170)**
852 48 170

Note: This is an example for semantic error.

2.2 KEYWORDS

- **Keywords are words which are predefined/have fixed meaning and these meanings cannot be changed.**
- The interpreter uses keywords to recognize the structure of the program and they cannot be used as variable names.

- **Python reserves 33 keywords:**

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

2.3 VARIABLES

- One of the most powerful features of a programming language is the ability to manipulate variables.
- **A variable is a name that refers to a value.**

OR

Variables are nothing but reserved memory locations to store values.

- Programmers generally choose variable names that are meaningful and document what the variable is used for.
- The general rules for naming a variable is:
  - **It can contain both letters and numbers, but cannot start with a number.**
  - **We can use uppercase letters, but it is a good practice to begin variable names with lowercase letter.**
  - **The underscore character (_) can appear in the variable name. It is often used whenever we want a variable name with multiple words. For example: my_name, sum_of_no.**
  - **Keywords, whitespaces and special characters are not allowed.**
  - **Variable names can start with an underscore character but we should generally avoid doing this unless we are writing library code for others to use.**
- If we try to violate the above rules of naming a variables, then it will give you syntax error. **For example:**

I.
```
>>> class = 'Python Application Programming'
```
→ **Keyword cannot be used**
```
  File "<stdin>", line 1
    class = 'Python Application Programming'
          ^
SyntaxError: invalid syntax
```

II.
```
>>> sum of = 0
```
→ **No spaces between two names**
```
  File "<stdin>", line 1
    sum of = 0
        ^
SyntaxError: invalid syntax
```

```
>>>
```

III. `>>> shop@ = 'myntra'` → No special characters

```
  File "<stdin>", line 1
    shop@ = 'myntra'
         ^
SyntaxError: invalid syntax
```

IV. `>>> 99Sale = 99` → Cannot start with a number

```
  File "<stdin>", line 1
    99Sale = 99
       ^
SyntaxError: invalid syntax
```

- In Python, **variables do not need explicit declaration** to reserve memory space. **The declaration happens automatically when you assign a value to a variable**. The **equal sign (=) is used to assign values to variables**.
- **An *assignment statement* creates new variables and gives them values.**
- The operand to the left of the '=' operator is the name of the variable and the operand to the right of the '=' operator is the value stored in the variable.
- The below example makes three assignments. The **first assigns a string to a new variable called message, the second assigns integer 10 to n, the third assigns approximate value of pi**.

```
>>> message = 'one has to be odd to be number one'
>>> n = 10
>>> pi = 3.1412
```

- To display the value of a variable, we can use print statement as follows:

```
>>> print(n)
10
>>> print(pi)
3.1412
>>> print(message)
one has to be odd to be number one
```

OR

```
>>> print(n,pi,message)
10 3.1412 one has to be odd to be number one
```

OR

```
>>> print(n,'\n',pi,'\n',message)
10
 3.1412
 one has to be odd to be number one
```

- The type of a variable is the type of the value it refers to.

```
>>> type(message)
<class 'str'>
>>> type(n)
```

```
<class 'int'>
>>> type(pi)
<class 'float'>
```

- **Python allows us to assign a single value to multiple variables simultaneously.**

For example:

```
>>> a = b = c = 5
>>> print(a)
5
>>> print(b)
5
>>> print(c)
5
```

OR

```
>>> a = b = c = 5
>>> print(a, b, c)
5 5 5
```

- **We can assign multiple values to multiple variables.**

For example:

```
>>> x,y,z = 'Awesome',10,-6.4
>>> print(x,y,z)
Awesome 10 -6.4
```

2.4 STATEMENTS

- A statement is a unit of code that the Python interpreter can execute.
- When you type a statement in interactive mode, the interpreter executes it and displays the result.

For example:

```
>>> x = 10
>>> print(x)
10
>>> y = 5.1
>>> print(y)
5.1
```

- A script usually contains a set/sequence of statements. If there is more than one statement, the result appear one at a time as the statement execute.

For example, in the script:

```
print(4)
x = 51          → Assignment statement does not produce any output
print(x)
```

produces the output

```
4
```

```
51
```

2.5 EXPRESSIONS

- **An expression is a combination of values, variables and operators.**

  For example,

  15 → **Note: A value by itself is considered as an expression**

  a

  a + 15

- If we type an expression in an interactive mode, the interpreter evaluates it and displays the result.

  For example,

```
>>> 1 + 5
6
```

2.6 BOOLEAN EXPRESSIONS

- **A Boolean expression is an expression which is either true or false.**
- **The operator == compares two operands and produces True if they are equal and False otherwise.**

  For example,

```
>>> 5 == 5
True
>>> 3 == 1
False
>>> 97 < 64
False
>>> 4 != 4
False
>>> 2 >= 0.97
True
>>> 'Hi' == 'Hi'
True
>>> 'z' == 'q'
False
```

- **True and False are special values that belongs to the class bool, they are not strings.**

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

NOTE: = is assignment operator and == is comparison operator

2.7 ASKING THE USER FOR INPUT

- **Python provides a built-in function called input that gets input from the keyboard**.
- When the input function is called, the program stops and waits for the user to give input.
- **When the user presses enter, the program resumes and input function return what the user had typed as string.**

For example,

I. To take input a string from the user

```
>>> mess = input()
Hi. Good Morning!!!
>>> print(mess)
Hi. Good Morning!!!
```

OR

```
>>> mess = input()
Hi. Good Morning!!!
>>> print("The message entered is", mess)
The message entered is Hi. Good Morning!!!
```

II. To display/prompt a mess to the user before taking input

```
>>> name = input("Enter your name?")
Enter your name?akhilaa
>>> print(name)
akhilaa
```

OR

```
>>> mess = input("Enter your name?\n")
Enter your name?
akhilaa
>>> print("My name is", mess)
My name is Akhilaa
```

Note: '\n' is a new line character that causes a line break

III. To take integer as input from the user

```
>>> age = input("Enter your age?\n")
Enter your age?
28
>>> print(age)
28
>>> type(age)
<class 'str'>
```

→ **If we try to check the type of age it shows as 'str' because input function returns string.**

To avoid taking input an integer as string we can explicitly typecast it to integer as below:

```
>>> age = int(input("Enter your age?\n"))
Enter your age?
28
```

```
>>> print("My age is", age)
My age is 28
>>> type( age)
<class 'int'>
```

IV. To take floating point number as input from the user

```
>>> marks = float(input("Enter marks: "))
Enter marks: 46.5
>>> print(marks)
46.5
>>> type(marks)
<class 'float'>
```

2.8 OPERATORS AND OPERANDS

- **Operators are special symbols that represent computations like mathematical and logical.**
- **The values the operator is applied to are called operands.**
- The different types of operators are:
  1. **Arithmetic Operators**
  2. **Relational (Comparison) Operators**
  3. **Assignment Operators**
  4. **Logical Operators**
  5. **Bitwise Operators**
  6. **Membership Operators**
  7. **Identity Operators**

ARITHMETIC OPERATORS

- **Assume variable a holds 10 and variable b holds 20:**

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	**>>> a + b** **30**
- Subtraction	Subtracts right hand operand from left hand operand.	**>>> a - b** **-10**
*** Multiplication**	Multiplies values on either side of the operator	**>>> a * b** **200**
/ Division	Divides left hand operand by right hand operand	**>>> a / b** **0.5** **>>> b / a** **2.0** **>>> 9 / 2** **4.5**

		>>> 9.4 / 3
		3.1333333333333333
		>>> 3.3 / 3.2
		1.0312499999999998
		>>> -9.3 / 91
		-0.10219780219780221

% Modulus	Divides left hand operand by right hand operand and returns remainder	>>> a % b
		10
		>>> b % a
		0
		>>> -2 % 3
		1
		>>> 2 % 3
		2
		>>> 5.6 % 5.6
		0.0
		>>> 3.2 % 2
		1.2000000000000002
		>>> -5.7 % 2.3
		1.1999999999999993

** Exponent	Performs exponential (power) calculation on operators	>>> a ** b 100000000000000000000

//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)	>>> 9 // 2
		4
		>>> 9.4 // 3
		3.0
		>>> 9 // 2
		4
		>>> 9.0 // 2.0
		4.0
		>>> 9.0 // 2
		4.0
		>>> 11 // 3
		3
		>>> -11 // 3
		-4
		>>> 11.0 // 3
		3.0
		>>> -11.0 // 3
		-4.0

NOTE: There is a change in division operator between Python 2.x and Python 3.x. **In Python 3.x the result of division is a floating point number.**

For example,

- **The division operator in Python 3.x gives floating point result.**

  >>> minute = 59

  >>> minute/60

  0.9833333333333333

- **The division operator in Python 2.x would divide two integers and truncate the result to an integer.**

  >>> minute = 59

  >>> minute/60

  0

  To obtain the same answer in Python 3.x use floored (// integer) division.

  >>> minute = 59

  >>> minute//60

  0

➢ The **modulus operator (%) works on integers and gives the remainder when the first operand is divided by the second.**

For Example:

I. **>>> remainder = 7 % 3**
   >>> print(remainder)
   1

II. **>>> remainder = 2 % 5**
   >>> print(remainder)
   2

III. **>>> remainder = 3.1 % 2**
   >>> print(remainder)
   1.1

IV. **Suppose you want quotient then**
   >>> quotient = 7 / 3
   >>> print(quotient)
   2.3333333333333335
   >>> quotient = 7 // 3 → **Gives you floored integer division**
   >>> print(quotient)
   2

➢ The modulus operator is useful:

- **To check if a number is divisible by another number or not.**

  For example,

  If x % y is zero, then x is divisible by y

- **To extract the right most digit or digits from a number.**

  For example,

  I. **x % 10 gives the right most digit of x (in base 10).**

II. x % 100 gives the last two digits

RELATIONAL (COMPARISION) OPERATORS

- **These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.**
- **Assume variable a holds 10 and variable b holds 20:**

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	>>> a == b False
!=	If values of two operands are not equal, then condition becomes true.	>>> a != b True
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	>>> a > b False
<	If the value of left operand is less than the value of right operand, then condition becomes true.	>>> a < b True
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	>>> a >= b False
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	>>> a <= b True

ASSIGNMENT OPERATORS

- Assume variable **a** holds **10** and variable **b** holds **20**

Operator	Description	Example

=	Assigns values from right side operands to left side operand	>>> c = a + b
		>>> print(c)
		30

+=
Add AND
	It adds right operand to the left operand and assign the result to left operand	>>> a += b
		>>> print(a)
		30

-=
Subtract AND
	It subtracts right operand from the left operand and assign the result to left operand	>>> a -= b
		>>> print(a)
		-10

*=
Multiply AND
	It multiplies right operand with the left operand and assign the result to left operand	>>> a *= b
		>>> print(a)
		200

/= **Divide AND**	It divides left operand with the right operand and assign the result to left operand	**>>> a /= b** **>>> print(a)** **0.5**
%= **Modulus AND**	It takes modulus using two operands and assign the result to left operand	**>>> a %= b** **>>> print(a)** **10**
****=** **Exponent AND**	Performs exponential (power) calculation on operators and assign value to the left operand	**>>> a **= b** **>>> print(a)** **10000000000000000000**
//= **Floor Division**	It performs floor division on operators and assign value to the left operand	**>>> a //= b** **>>> print(a)** **0**

LOGICAL OPERATORS

- There are three logical operators: **and, or and not**.

  For example,

  I. **x > 0 and x < 10**

  is true only if x is greater than 0 *and* less than 10.

  II. **n%2 == 0 or n%3 == 0**

  is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

  III. **not (x > y)**

  is true if x > y is false; that is, if x is less than or equal to y.

- **Any non-zero number is treated as "True".**

  For example,

  >>> 14 and True

  True

Operator	Description	Example
and	Logical AND	**>>> 252.7 and True** **True** **>>> 5 > 2 and 3 < 46** **True** **>>> 5 > 3 and -1 > 3** **False**
or	Logical OR	**>>> 3 >= 2 or 98 -2** **True** **>>> 3 <= 2 or 98 < 9** **False**

not	LOGICAL NOT	**>>> not(False)** **True** **>>> not(True)** **False** **>>> not (5<3)** **True** **>>> not (5>3)** **False** **>>> not (5>3 and -1<3)** **False** **>>> not (5<3 or -1>3)** **True**

BITWISE OPERATORS

- Bitwise operator works on bits and performs bit by bit operation. Assume if **a = 60**; and **b = 13**; Now in binary format they will be as follows –

a = 0011 1100
b = 0000 1101
----
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

Operator	Description	Example
& **Bitwise AND**	Operator copies a bit to the result if it exists in both operands	**>>> a & b** **12 → (means 0000 1100)**
| **Bitwise OR**	It copies a bit if it exists in either operand.	**>>> a | b** **61 → (means 0011 1101)**
^ **Bitwise XOR**	It copies the bit if it is set in one operand but not both.	**>>> a ^ b** **49 → (means 0011 0001)**
~ **Bitwise Not** **(Ones Complement)**	It is unary and has the effect of 'flipping' bits.	**>>> ~a** **-61 → (means 1100 0011 in 2's complement form due to a signed binary number.**
<< **Bitwise Left Shift**	The left operands value is moved left by the number of bits specified by the right operand.	**>>> a << 2** **240 → (means 1111 0000)**
>> **Bitwise Right Shift**	The left operands value is moved right by the number of bits specified by the right operand.	**>>> a >> 2** **15 → (means 0000 1111)**

MEMBERSHIP OPERATORS

- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

IDENTITY OPERATORS

- Identity operators compare the memory locations of two objects.

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

2.9 ORDER OF OPERATIONS

- **When we have more than operator in an expression, the order of evaluation depends on the rules of precedence.**
- The acronym ***PEMDAS*** is used to remember the rules:
  - **Parenthesis** has the **highest precedence**. **The expression in the parenthesis are evaluated first**. It can also be used to make the expression easy to read.
    For example:
    I. **2 * (3 – 1)** → **Gives result 4**
    II. **(1 + 1) ** (5 – 2)** → **Gives result 8**
  - **Exponentiation** has the next highest precedence.
    For example:
    I. **2 ** 1 + 1** → **Gives result 3 and not 4**
    II. **3 * 1 ** 3** → **Gives result 3 and not 27**

- **Multiplication** and **Division** have the same precedence, which is higher than **Addition** and **Subtraction**, which also have the same precedence.
  For example,
- **Operators with the same precedence are evaluated from left to right.**
  For example,
  The expression 5 - 3 - 1 gives the result 1, not 3, because 5 - 3 is evaluated first and then 1 is subtracted from 2.

NOTE:

- ✓ Always put parenthesis in your expressions to make sure that the computations are performed in the order you want.
- ✓ Python statements always end with a new line, but it also allows multiline statement using "\" character at the end of the line as shown below:

```
>>> x = (3+5)\
... * 8
>>> print(x)
64
```

2.10 OPERATOR PRECEDENCE AND ASSOCIATIVITY

The operator precedence in Python are listed in the following table. It is in descending order, upper group has higher precedence than the lower ones.

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Positive, Negative, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparison, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Associativity of Python Operators

- We can see in the above table that more than one operator exists in the same group. These operators have the same precedence.
- **When two operators have the same precedence, associativity helps to determine which the order of operations**.
- **Associativity is the order in which an expression is evaluated that has multiple operator of the same precedence. Almost all the operators have left-to-right associativity**.
  For example, multiplication and floor division have the same precedence. Hence, if both of them are present in an expression, left one is evaluates first.
- **Exponent operator ** has right-to-left associativity in Python**.
  For example:

```
>>> print(2**3**2)
512
>>> print((2**3)**2)
64
```

- Some operators like assignment operators and comparison operators do not have associativity in Python.
  For example:
  x < y < z neither means (x < y) < z nor x < (y < z). x < y < z is equivalent to x < y and y < z, and is evaluates from left-to-right.

NOTE: From Python documentation on operator precedence

Operator	Description
()	Parentheses (grouping)
***f*(args...)**	Function call
***x*[index:index]**	Slicing
***x*[index]**	Subscription
x.attribute	Attribute reference
**	Exponentiation
~*x*	Bitwise not
+*x*, -*x*	Positive, negative
*, /, %	Multiplication, division, remainder
+, -	Addition, subtraction
<<, >>	Bitwise shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

in, not in **is, is not** **<, <=, >, >=, !=, ==**	Membership Comparisons Identity
not *x*	Boolean NOT
and	Boolean AND
or	Boolean OR
lambda	Lambda expression

2.11 STRING OPERATIONS

- **The + operator is also used with string.**
- **It performs concatenation, meaning joining two strings.**

For example,

I.
```
>>> first = 10
>>> second = 20
>>> print(first+second)        → Performs addition
30
```

II.
```
>>> first = '10'
>>> second = '20'
>>> print(first+second)        → Performs concatenation
1020
```

2.12 COMMENTS

- As the programs become bigger and more complicated, they become very difficult to read.
- It is a good practice **to add notes to the program that briefly tells what the program is does**. **These are called comments** and in **Python they start with # symbol**.

For example,

```
#compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

OR

```
percentage = (minute * 100) / 60        # percentage of an hour
```

- **In python, multi-line comments can be written using “””.**

```
"""
To display hello world on the screen
Author: Akhilaa
"""
print('hello world')
```

- **In python, multi-line comments can be written using ‘’’.**

```
'''
```

```
To display hello world on the screen
Author: Akhilaa
'''
print('hello world')
```

2.13 EXERCISES

1. width = 17
   height = 12.0
   For each of the following expressions, write the value of the expression and the type (of the value of the expression).
   1. width//2
   2. width/2.0
   3. height/3
   4. 1 + 2 * 5

   Use the Python interpreter to check your answers.
2. Write a python program which prompts the user for a Celsius temperature, convert the temperature to Fahrenheit, and print out the converted temperature.
   F = ((9/5)*c) + 32 **or** F = 1.8 C +32
   C = (5/9)*(F-32)
3. Write a python program to compute simple interest.
4. Write a python program to compute the area and circumference of a circle.
   $A = \pi r^2$ $C = \pi r$
5. Write a python program to swap two numbers.
6. Write a python program to compute sum and average of 5 numbers.
7. Write a python program to demonstrate arithmetic operators(calculator program).
8. Write a python program to compute area and perimeter of a rectangle.
   A = l * b P = 2 (l+b)

2.14 GLOSSARY

assignment	A statement that assigns a value to a variable.
concatenate	To join two operands end to end.
comment	Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.
evaluate	To simplify an expression by performing the operations in order to yield a single value.
expression	A combination of variables, operators, and values that represents a single result value.
floating point	A type that represents numbers with fractional parts.
integer	A type that represents whole numbers.
keyword	A reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names.

mnemonic	A memory aid. We often give variables mnemonic names to help us remember what is stored in the variable.
modulus operator	An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.
operand	One of the values on which an operator operates.
operator	A special symbol that represents a simple computation like addition, multiplication, or string concatenation.
rules of precedence	The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.
statement	A section of code that represents a command or action. So far, the statements we have seen are assignments and print expression statement.
string	A type that represents sequences of characters.
type	A category of values. The types we have seen so far are integers (type int), floating-point numbers (type float), and strings (type str).
value	One of the basic units of data, like a number or string, which a program manipulates.
variable	A name that refers to a value.

3 CONDITIONAL EXECUTION

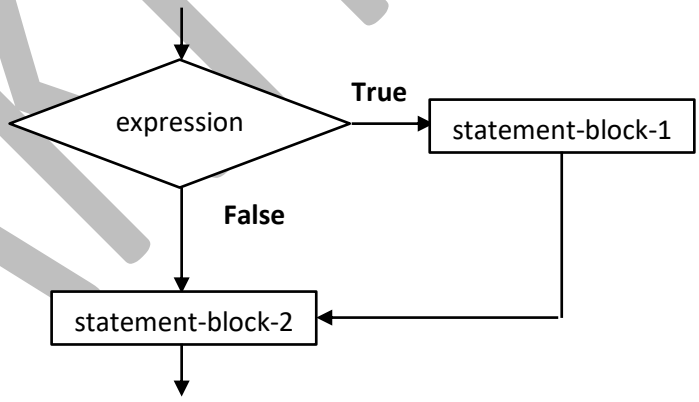
3.1 IF STATEMENT (CONDITIONAL EXECUTION)

- **Conditional statements gives the ability to check conditions change the behaviour of the program accordingly**.
- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- The syntax of if-statement is:

```
if expression :
      statement-block-1
statement-block-2
```

The boolean expression after the if statement is called the *condition*. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.

- **The expression is evaluated to either True or False**.
  - **If True then the intended statements (statement-block-1) get executed** and the control comes outside the if statement and the execution of further statements (statement-block-2) continues if any.
  - **If the expression is evaluated to be false, then intended block (statement-block-1) is skipped.**
- **There is no limit on the number of statements that can appear in the body, but there must be at least one.**
- **Note: Parenthesis can be used for the expression in the if statement but it is optional.**
- The flowchart is shown below:



- **Python interprets non-zero values as True. None and 0 are interpreted as False.**
- **# Python program to check if a given number is positive.**

```
n = int(input('Enter a number: '))

if n > 0 :
   print('Positive')

print('Bye')
```

Output:

I. **Enter a number: 10**
Positive
Bye

II. **Enter a number: -74**
Bye

III. **Enter a number: 0**
Bye

- **Sometimes, it is useful to have a body with no statements (usually as a place holder for code you haven't written yet). In that case, you can use the pass statement, which does nothing.**
For example:

```
if n < 0 :
    pass    # need to handle negative values
```

- If you enter an if statement in the Python interpreter, the prompt will change from three chevrons to three dots to indicate you are in the middle of a block of statements, as shown below:

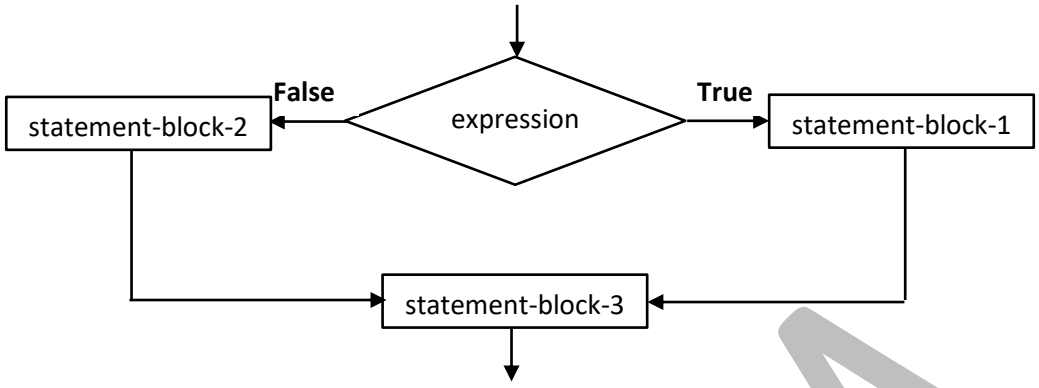
```
>> x = 3
>>> if x < 10 :
...     print('Small')
...
Small
```

3.2 IF-ELSE STATEMENT (ALTERNATIVE EXECUTION)

- It is two-way selection statement. It is used when we have to choose between two alternatives.
- The syntax of if-else statement is:

```
if expression :
        statement-block-1
else :
        statement-block-2
statement-block-3
```

- The expression is evaluated to true or false.
  - **If the expression is evaluated to true, then statement-block-1 is executed** and the control comes outside the if-else and statement-block-3 is executed.
  - **If the expression is evaluated to false, then statement-block-2 is executed** and the control comes outside the if-else and statement-block-3 is executed.
- **Since the condition must either be true or false, exactly one of the alternatives will be executed. The alternatives are called *branches*.**
- The flowchart is shown below:



- **# Python program to check if a number is positive or negative.**

```
n = float(input('Enter a number: '))

if n > 0 :
   print('Positive')
else :
   print('Negative')

print('Bye')
```

Output:

I. **Enter a number: -972**
   Negative
   Bye
II. **Enter a number: 0**
   Negative
   Bye
III. **Enter a number: 76.359**
   Positive
   Bye
IV. **Enter a number: 0.036**
   Positive
   Bye

3.3 ElSE-IF LADDER STATEMENT (CHAINED CONDITIONALS)

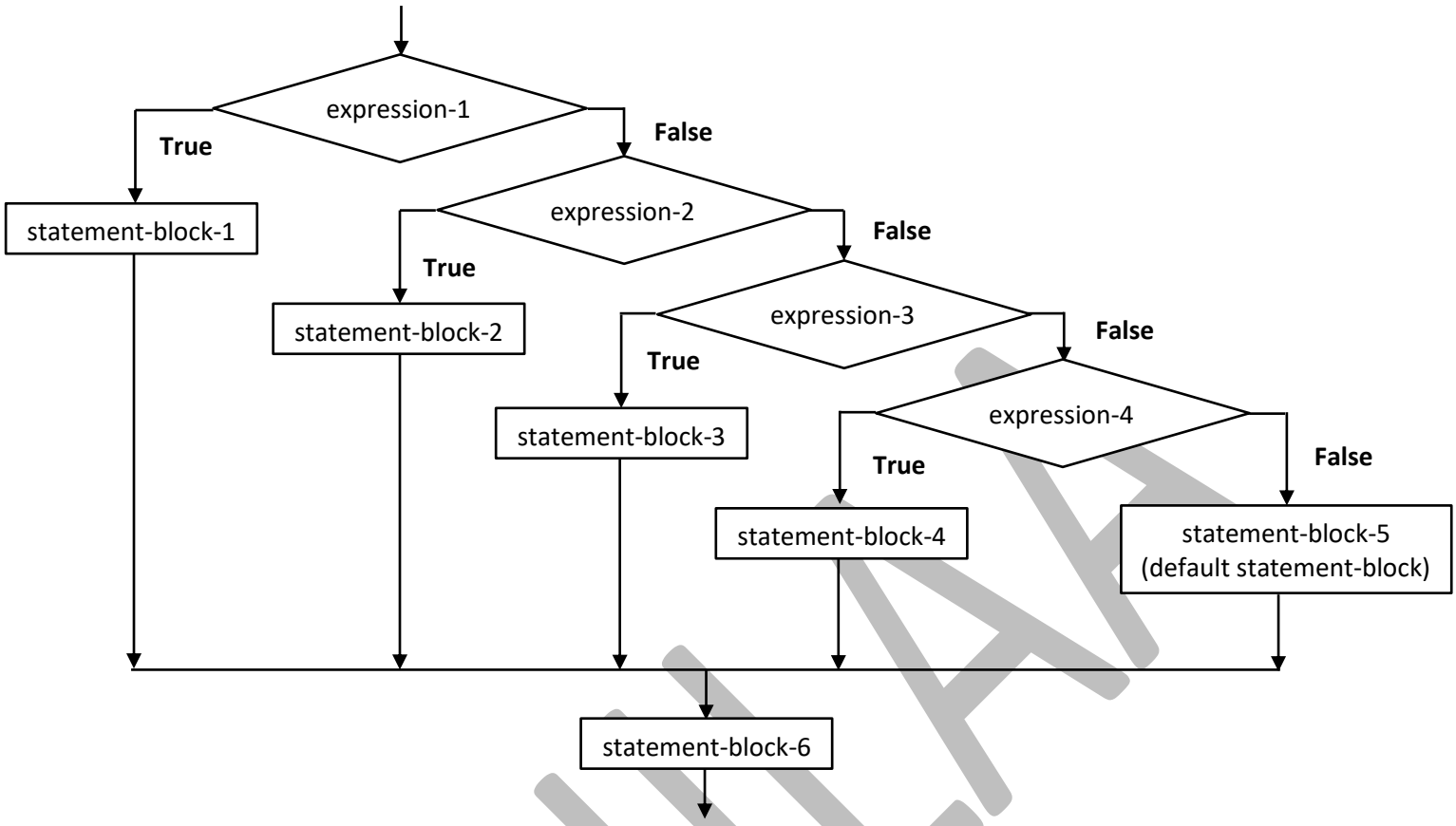
- It is multi-way selection statement. **It is used when we have to make a choice among many alternatives**. One way to express a computation like that is a *chained conditional.*
- The syntax of else-if statement is:

```
if expression-1 :
```

```
        statement-block-1
    elif expression-2 :
        statement-block-2
    elif expression-3 :
        statement-block-3
    elif expression-4 :
        statement-block-4
    elif expression-5 :
        statement-block-5
    statement-block-6
```

Note: elif is an abbreviation of "else if"

- **The expression is evaluated in top to bottom order. If an expression is evaluated to true, then the statement-block associated with that expression is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.**
- **There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.**
- For example:
    - If expression-1 is evaluated to true, then statement-block-1 is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.
    - If expression-1 is evaluated to false, then expression-2 is evaluated. If expression-2 is evaluated to true then statement-block-2 is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.
    - If all the expressions are evaluated to false, then the last statement-block-5 (default) is executed and the control comes out of the entire else if ladder and continues execution from statement-block-6 if any.
- The flowchart is shown below:



➢ **# Python program to check if a number is positive, negative or zero.**

```
n = int(input('Enter the number: '))

if n > 0 :
  print('The number',n,'is Positive')
elif n < 0 :
   print('The number',n,'is Negative')
else :
   print('The number',n,'is Zero')

print('Good Bye')
```

Output:

I. **Enter the number: 0**
   The number 0 is Zero
   Good Bye
II. **Enter the number: 73**
   The number 73 is Positive
   Good Bye
III. **Enter the number: 0**

The number 0 is Zero

Good Bye

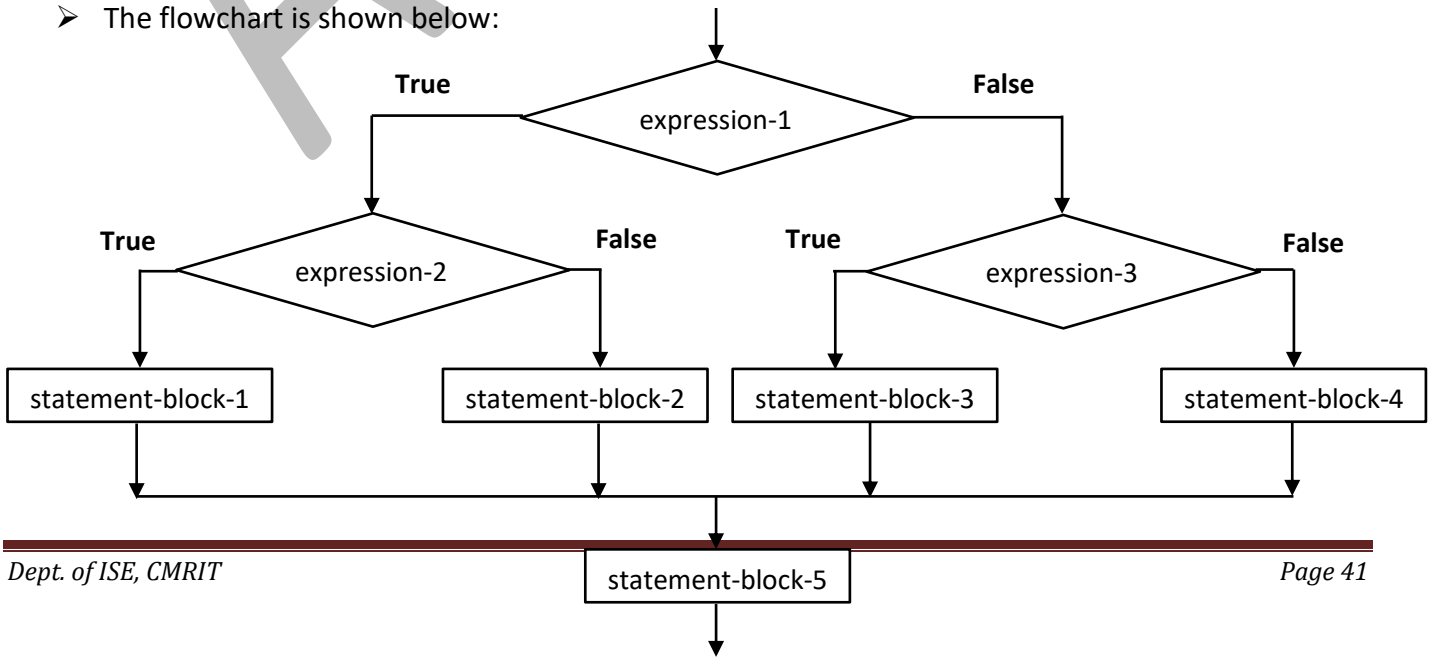
3.4 NESTED IF-ELSE STATEMENT (NESTED CONDITIONALS)

- It is used when an action has to be performed based on many decisions.
- **An if-else statement within another if-else statement is called nested if-else statement.**
- The syntax of nested if-else statement is:

```
if expression-1 :
      if expression-2 :
            statement-block-1
      else :
            statement-block-2
else
      if expression-3 :
            statement-block-3
      else :
            statement-block-4
statement-block-5
```

- The expression-1 is evaluated to true or false.
  - If expression-1 is evaluated to true, then expression-2 is evaluated to true or false. If expression-2 is evaluated to true, then statement-block-1 is executed. If expression-2 is evaluated to false, then statement-block-2 is executed.
  - If expression-1 is evaluated to false, then expression-3 is evaluated to true or false. If expression-3 is evaluated to true, then statement-block-3 is executed. If expression-3 is evaluated to false, then statement-block-4 is executed.

  In either of the cases, after execution of a particular statement-block the control comes outside the nested if-else statement and continues execution from statement-block-5.
- Although the indentation of the statements makes the structure apparent, *nested conditionals* become difficult to read very quickly. In general, it is a good idea to avoid them when you can.
- The flowchart is shown below:



- **# Python program to check if a number is lesser, greater or equal to another number.**

```
x = int(input('Enter X: '))
y = int(input('Enter Y: '))

if x == y :
   print('X and Y are Equal')
else :
   if x < y :
      print('X is lesser than Y')
   else :
      print('X is greater than Y')
```

Output:

I. Enter X: 6
   Enter Y: 9
   X is lesser than Y
II. Enter X: 84
   Enter Y: 62
   X is greater than Y
III. Enter X: 5
   Enter Y: 5
   X and Y are Equal

3.5 CATCHING EXCEPTIONS USING TRY AND EXCEPT

- **An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program.**
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- **For example:**

 # Python program to demonstrate an exception.

```
a = int(input('Enter a: '))
b = int(input('Enter b: '))

res = a/b

print(res)
```

Output:

I. Enter a: 10

Enter b: 5
2.0

II. Enter a: 4
Enter b: 0
Traceback (most recent call last):
File"C:/Users/akhil/AppData/Local/Programs/Python/Python36
32/tryexcept.py", line 6, in <module>
res = a/b
ZeroDivisionError: division by zero → This is an exception

- **The idea of try and except is that you know that some sequence of instruction(s) may have a problem and you want to add some statements to be executed if an error occurs. These extra statements (the except block) are ignored if there is no error.**
- You can think of the try and except feature in Python as an "insurance policy" on a sequence of statements.
- **Python starts by executing the sequence of statements in the try block. If all goes well, it skips the except block and proceeds. If an exception occurs in the try block, Python jumps out of the try block and executes the sequence of statements in the except block.**
- We can rewrite the above program to handle exceptions as follows:

```
# Python program to demonstrate catching an exception using try and catch.

a = int(input('Enter a: '))
b = int(input('Enter b: '))

try:
   res = a/b
   print(res)
except:
   print('Divide by Zero Exception!!!')
```

Output:

I. Enter a: 8
Enter b: 0
Divide by Zero Exception!!!

II. Enter a: 53
Enter b: 35
1.5142857142857142

- **Handling an exception with a try statement is called *catching* an exception.**

3.6 SHORT-CIRCUIT EVALUATION OF LOGICAL EXPRESSIONS

- **When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left to right**. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.
- **When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression**. **When the evaluation of a logical expression stops because the overall value is already known, it is called short-circuiting the evaluation.**
- While this may seem like a fine point, **the short-circuit behaviour leads to a clever technique called the guardian pattern.**
- Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y)>2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

  - The third calculation failed because Python was evaluating (x/y) and y was zero, which causes a runtime error.
  - But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the *short-circuit* rule and there was no error.

- **We can construct the logical expression to strategically place a *guard* evaluation just before the evaluation that might cause an error as follows:**

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
```

```
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

- In the first logical expression, x >= 2 is False so the evaluation stops at the and.
- In the second logical expression, x >= 2 is True but y != 0 is False so we never reach (x/y).
- In the third logical expression, the y != 0 is *after* the (x/y) calculation so the expression fails with an error.
- In the second expression, we say that y != 0 acts as a *guard* to insure that we only execute (x/y) if y is non-zero.

3.7 EXERCISES

1. Rewrite your pay computation to give the employee 1.5 times the hourly rate for hours worked above 40 hours.

   Enter Hours: 45
   Enter Rate: 10
   Pay: 475.0

Program:

```
hours = float(input('Enter the number of hours: '))

rate = float(input('Enter the rate: '))

if hours<=40 :
   pay = rate * hours
   print('The total pay is', pay)
else :
   rem = hours-40
   pay = 400 + (rem*rate*1.5)
   print('The total pay is', pay)
```

Output:

I. Enter the number of hours: 40
   Enter the rate: 10
   The total pay is 400.0

II. Enter the number of hours: 45
   Enter the rate: 10
   The total pay is 475.0

III. Enter the number of hours: 50.2

Enter the rate: 10

The total pay is 553.0

2. Rewrite your pay program using try and except so that your program handles non-numeric input gracefully by printing a message and exiting the program. The following shows two executions of the program:

Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input

Program:

```
try :
   hours = float(input('Enter the number of hours: '))

   rate = float(input('Enter the rate: '))

   if hours<=40 :
      pay = rate * hours
      print('The total pay is', pay)
   else :
      rem = hours-40
      pay = 400 + (rem*rate*1.5)
      print('The total pay is', pay)
except :
   print('Error! Please enter numeric input')
```

OR

```
try :
   hours = float(input('Enter the number of hours: '))

   rate = float(input('Enter the rate: '))
except :
   print('Error! Please enter numeric input')

else :
   if hours<=40 :
      pay = rate * hours
      print('The total pay is', pay)
   else :
      rem = hours-40
```

```
pay = 400 + (rem*rate*1.5)
print('The total pay is', pay)
```

Output:

I. **Enter the number of hours: nine**
Error! Please enter numeric input

II. **Enter the number of hours: 23**
Enter the rate: ten
Error! Please enter numeric input

III. **Enter the number of hours: 51**
Enter the rate: 13
The total pay is 614.5

3. Write a program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error message. If the score is between 0.0 and 1.0, print a grade using the following table:

Score Grade
>= 0.9 A
>= 0.8 B
>= 0.7 C
>= 0.6 D
< 0.6 F

Enter score: 0.95 A-

Enter score: perfect
Bad score

Enter score: 10.0
Bad score
Enter score: 0.75
C

Enter score: 0.5
F

Run the program repeatedly as shown above to test the various different values for input.

Program:

```
try :
   score = float(input('Enter the score: '))
except :
   print('Error, Score out of range.')
```

```
else :
    if score < 0 or score > 1.0 :
        print('Error, Score out of range')
    elif score >= 0.9 and score <= 1.0 :
        print('Grade A')
    elif score >= 0.8 and score < 0.9 :
        print('Grade B')
    elif score >= 0.7 and score < 0.8 :
        print('Grade C')
    elif score >= 0.6 and score < 0.7 :
        print('Grade D')
    elif score >= 0 and score <0.6 :
        print('Fail')
```

Output:

I. **Enter the score: 10**
   Error, Score out of range
II. **Enter the score: 0**
   Fail
III. **Enter the score: 0.62**
   Grade D
IV. **Enter the score: 0.99**
   Grade A
V. **Enter the score: 0.75**
   Grade C

4. Write a python program to check if a number is even or odd.
5. Write a python program to check if a number is prime or not.
6. Write a python program to find maximum two numbers.
7. Write a python program to find maximum of three numbers.
8. Write a python program to check whether the entered character is a vowel or consonant.
9. Write a python program to check whether the entered character is 'F' (Female) or 'M' (Male).
10. Write a python program to check if a year is leap year, century leap year or not a leap year.
11. Write a python program to input week number and print week day.
12. Write a python program to input month number and print the number of days in that month.
13. Write a python program to check whether the triangle, isosceles or scalene triangle.
14. Write a python program to compute roots of a quadratic equation.
15. Write a python program to calculate profit or loss.
16. Write a python program to input the basic salary of an employee and calculate its gross salary according to the following:

    Basic Salary <= 10000 : HRA = 20%, DA = 80%

    Basic Salary <= 20000 : HRA = 25%, DA = 90%

    Basic Salary > 20000 : HRA = 30%, DA = 95%

17. Write a python program to input electricity unit charges and calculate total electricity bill according to the given condition:
    For first 50 units Rs. 0.50/unit
    For next 100 units Rs. 0.75/unit
    For next 100 units Rs. 1.20/unit
    For unit above 250 Rs. 1.50/unit
    An additional surcharge of 20% is added to the bill
18. Write a python program to input marks of five subjects Physics, Chemistry, Biology, Mathematics and Computer. Calculate percentage and grade according to the following:
    Percentage >= 90% : Grade A
    Percentage >= 80% : Grade B
    Percentage >= 70% : Grade C
    Percentage >= 60% : Grade D
    Percentage >= 40% : Grade E
    Percentage < 40% : Grade F

3.8 GLOSSARY

body	The sequence of statements within a compound statement.
boolean expression	An expression whose value is either True or False.
branch	One of the alternative sequences of statements in a conditional statement.
chained conditional	A conditional statement with a series of alternative branches.
comparison operator	One of the operators that compares its operands: ==, !=, >, <, >=, and <=.
conditional statement	A statement that controls the flow of execution depending on some condition.
condition	The boolean expression in a conditional statement that determines which branch is executed.
compound statement	A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.
guardian pattern	Where we construct a logical expression with additional comparisons to take advantage of the short-circuit behaviour.
logical operator	One of the operators that combines boolean expressions: and, or, and not.
nested conditional	A conditional statement that appears in one of the branches of another conditional statement.
traceback	A list of the functions that are executing, printed when an exception occurs.
short circuit	When Python is part-way through evaluating a logical expression and stops the evaluation because Python knows the final value for the expression without needing to evaluate the rest of the expression.

4 FUNCTIONS

- **A function is a named sequence of statements that performs a computation.**

  OR

  A function is a set of statements that does some particular task.
- There are two types of functions:
  - i. **Built-in-functions:** Functions that are predefined.
  - ii. **User-defined functions:** Functions that are created by the users according to the requirement.

4.1 DEFINING A FUNCTION

- **A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.**
- **def** keyword is used to define the function (marks the start of function header).
- A function name to uniquely identify it. **Function naming follows the same rules of writing variable names (identifiers) in Python.**
  - Letters, numbers and some punctuation marks are legal, but the first character can't be a number.
  - You can't use a keyword as the name of a function.
  - Avoid having a variable and a function with the same name.
- **Parameters (arguments) through which we pass values to a function. They are optional**. The **empty parentheses** after the name indicate that this **function doesn't take any arguments**.
- A **colon (:) to mark the end of function header**.
- The first statement of a function can be an optional statement - the **documentation string (docstring) to describe what the function does.**
- **One or more valid python statements that make up the function body.** The body can contain any number of statements.
- **Statements must have same indentation level (usually 4 spaces).**
- An optional return statement to return a value from the function.
- The general syntax of defining a function is:

```
def function_name (parameters/arguments):
    """docstring"""  → Optional
    Statement 1
    Statement 2          Body of the function
    …
    Statement n
```

For Example:

```
def print_lyrics():
    print('Haan hum badalne lage,  Girne sambhalne lage')
    print('Jab se hai jaana tumhein, Teri ore chalne lage')
```

- If you type a function definition in interactive mode, **the interpreter prints ellipses (. . .) to let you know that the definition isn't complete**. To end the function, you have to enter an empty line (this is not necessary in a script).

```
>>> def print_lyrics():
...     print('Haan hum badalne lage, Girne sambhalne lage')
...     print('Jab se hain jaana tumhein, Teri ore chalne lage')
...
```

Note: **The below is an example that shows an error if you are not indenting the statements after the Function.**

header

```
>>> def print_lyrics():
... print('Jab se hai jaana tumhein, Teri ore chalne lage')
File "<stdin>", line 2
print('Jab se hai jaana tumhein, Teri ore chalne lage')
^
IndentationError: expected an indented block
```

- **Defining a function creates a variable with the same name.**

```
>>> print(print_lyrics)
<function print_lyrics at 0x05916228>
```

- **The value of print_lyrics is a function object, which has type "function".**

```
>>> print(type(print_lyrics))
<class 'function'>
```

4.2 FUNCTION CALL

- **A function call is a statement that executes a function. It consists of the function name followed by an argument list.**
- The general syntax of calling a function is:

```
function_name(parameters/arguments)
```

- **For Example:**

```
>>> print_lyrics()      → Function call
Haan hum badalne lage, Girne sambhalne lage
Jab se hain jaana tumhein, Teri ore chalne lage
```

- Once you have defined a function, you can use it inside another function.

```
>>> def repeat_lyrics():
...     print_lyrics()      → Function call
...     print_lyrics()      → Function call
...
>>> repeat_lyrics()      → Function call
Haan hum badalne lage, Girne sambhalne lage
Jab se hain jaana tumhein, Teri ore chalne lage
Haan hum badalne lage, Girne sambhalne lage
```

Jab se hain jaana tumhein, Teri ore chalne lage

4.3 ADVANTAGES OF FUNCTIONS

- Creating a new function gives you an opportunity to name a group of statements, which makes your program **easier to read, understand, 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.**

4.4 FRUITFUL FUNCTIONS AND VOID FUNCTIONS

- **A function that yields a result or returns a value is called is fruitful function.**
- When you call a fruitful function, you always want to do something with the result; for example, you might assign it to a variable or use it as part of an expression.
- **To return a result from a function, we use the return statement.**
- **When you call a function in interactive mode, Python displays the result. But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes.**

For example:

```
# A function that adds two numbers and returns a value.

def add_twonos(a,b):
  """Function to compute sum of two numbers."""
  sum = a + b
  return sum

v1 = int(input('Enter the first value: '))
v2 = int(input('Enter the second value: '))

res = add_twonos(v1,v2)

print('The sum of two numbers is: ', res)
```

Output:

I. **Enter the first value: 5**
   Enter the second value: 10
   The sum of two numbers is: 15
II. **Enter the first value: 347**
   Enter the second value: 213

The sum of two numbers is: 560

- **A function that performs an action but doesn't return a value is called void function.**

For example:

```
# A function to compute average of two numbers.

def avg_twonos(a,b):
  """Function to compute average of two numbers."""
   sum = a + b
   avg = sum / 2
   print('The average of two numbers is: ', avg)

v1 = float(input('Enter the first value: '))
v2 = float(input('Enter the second value: '))

avg_twonos(v1,v2)
```

Output:

I. Enter the first value: 45
   Enter the second value: 234
   The average of two numbers is: 139.5

II. Enter the first value: 654
    Enter the second value: 56
    The average of two numbers is: 355.0

4.5 BUILT-IN FUNCTIONS

- Python provides a number of important built-in functions that we can use without needing to provide the function definition.
- The **max** and **min** gives the **largest and smallest element** in the list respectively.

For Example:

```
>>> max('Hello World')
'r'
>>> min('Hello World')
' '
>>> max(10,45,67,6,-6,348.6,58)
348.6
>>> min(10,45,67,6,-6,348.6,58)
-6
```

The max function tells us the "largest character" in the string (which turns out to be the letter "r") and the min function shows us the smallest character (which turns out to be a space).

- **len function** which tells us **how many items are in its argument.**

For Example:

If the **argument to len is a string, it returns the number of characters in the string**.

```
>>> len('Hello World')
11
>>> list = [10, 325, 52, 25, 55.2, 436]
>>> print(len(list))
6
>>> len(10,45,67,6,-6,348.6,58)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (7 given)
```

Note: These functions are not limited to strings. They can operate on any set of values.

4.6 TYPE CONVERSION FUNCTIONS

- **Python also provides built-in functions that convert values from one type to another.**
- The **int function takes any value and converts it to an integer, if it can, or complains otherwise**:

  For Example:

  - **>>> int('32')** → **When 32 is passed as string it converts it into integer.**
    32
  - **>>> int('hello')** → **Gives an error.**
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    ValueError: invalid literal for int() with base 10: 'hello'
- **int can convert floating-point values to integers**, but **it doesn't round off; it chops off the fraction part**.

  For Example:

  - **>>> int(4.1926345)**
    4
  - **>>> int(-2.6035)**
    -2
  - **>>> int('25')**
    25
  - **>>> int('6.62536')** → **When 6.62536 (floating-point) is passed as string it gives an error.**
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    ValueError: invalid literal for int() with base 10: '6.62536'
- **float converts integers and strings to floating-point numbers.**

  For Example:

  - **>>> float(32)**
    32.0
  - **>>> float('3.1412')**

3.1412

- float('hello')
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: could not convert string to float: 'hello'

➢ **str converts its argument to a string.**
For Example:

- >>> str(32)
  '32'
- >>> str('3.1412')
  '3.1412'

4.7 MATH FUNCTIONS

➢ Python has a **math module** that provides most of the **mathematical functions**.
➢ **Before we can use the module, we have to import it**:
>>> import math → **This statement creates a module object named math**
➢ If you print the module object, we get some information about it:
>>> print(math)
<module 'math' (built-in)>
➢ The **module object contains the functions and variables defined in the module**. **To access one of the functions, you have to specify the name of the module and the name of the function, separated by a dot** (also known as a period). This format is called **dot notation**.

SOME OF THE FUNCTIONS IN MATH MODULE

Number-theoretic and representation functions

math.ceil(*x*)	Return the ceiling of *x*, the smallest integer greater than or equal to *x*.
math.fabs(*x*)	Return the absolute value of *x*.
math.factorial(*x*)	Return *x* factorial. Raises ValueError if *x* is not integral or is negative. (Note: ValueError is an exception raised when a built-in operation or function receives an argument that has the right type but an inappropriate value)
math.floor(*x*)	Return the floor of *x*, the largest integer less than or equal to *x.*
math.fsum(*iterable*)	Return an accurate floating point sum of values in the iterable.
math.gcd(*a*, *b*)	Return the greatest common divisor of the integers *a* and *b*. If either *a* or *b* is nonzero, then the value of gcd(a, b) is the largest positive integer that divides both *a* and *b*. gcd(0, 0) returns 0.
math.isfinite(*x*)	Return True if *x* is neither an infinity nor a NaN, and False otherwise. (Note that 0.0 *is* considered finite.)
math.isinf(*x*)	Return True if *x* is a positive or negative infinity, and False otherwise.
math.isnan(*x*)	Return True if *x* is a NaN (not a number), and False otherwise
math.modf(*x*)	Return the fractional and integer parts of *x*. Both results carry the sign of *x* and are floats.
math.trunc(*x*)	Return the Real value *x* truncated to an Integral (usually an integer).

Power and logarithmic functions

math.exp(*x*)	Return e**x.
math.log(*x*[, *base*])	With one argument, return the natural logarithm of *x* (to base *e*). With two arguments, return the logarithm of *x* to the given *base*, calculated as log(x)/log(base).
math.log1p(*x*)	Return the natural logarithm of *1+x* (base *e*). The result is calculated in a way which is accurate for *x* near zero.
math.log2(*x*)	Return the base-2 logarithm of *x*. This is usually more accurate than log(x, 2).
math.log10(*x*)	Return the base-10 logarithm of *x*. This is usually more accurate than log(x, 10).
math.pow(*x*, *y*)	Return x raised to the power y. Exceptional cases follow Annex 'F' of the C99 standard as far as possible. In particular, pow(1.0, x) and pow(x, 0.0) always return 1.0, even when x is a zero or a NaN. If both x and y are finite, x is negative, and y is not an integer then pow(x, y) is undefined, and raises ValueError. Unlike the built-in ** operator, math.pow() converts both its arguments to type float. Use ** or the built-in pow() function for computing exact integer powers.
math.sqrt(*x*)	Return the square root of *x*.

Trigonometric functions

math.sin(*x*)	Return the sine of *x* radians.
math.cos(*x*)	Return the cosine of *x* radians.
math.tan(*x*)	Return the tangent of *x* radians.
math.hypot(*x*, *y*)	Return the Euclidean norm, sqrt(x*x + y*y). This is the length of the vector from the origin to point (x, y).
math.acos(*x*)	Return the arc cosine of *x*, in radians.
math.asin(*x*)	Return the arc sine of *x*, in radians.
math.atan(*x*)	Return the arc tangent of *x*, in radians.
math.atan2(*y*, *x*)	Return atan(y / x), in radians.

Angular conversion

math.degrees(*x*)	Convert angle *x* from radians to degrees.
math.radians(*x*)	Convert angle *x* from degrees to radians.

Hyperbolic functions

math.acosh(*x*)	Return the inverse hyperbolic cosine of *x*.
math.asinh(*x*)	Return the inverse hyperbolic sine of *x*.
math.atanh(*x*)	Return the inverse hyperbolic tangent of *x*.
math.cosh(*x*)	Return the hyperbolic cosine of *x*.
math.sinh(*x*)	Return the hyperbolic sine of *x*.
math.tanh(*x*)	Return the hyperbolic tangent of *x*.

Special functions

math.gamma(*x*)	Return the Gamma function at *x*.
math.lgamma(*x*)	Return the natural logarithm of the absolute value of the Gamma function at *x*.
math.erf(*x*)	Return the error function at *x*.

	The erf() function can be used to compute traditional statistical functions such as the cumulative standard normal distribution.
math.erfc(*x*)	Return the complementary error function at *x*.

Constants

math.pi	The mathematical constant π = 3.141592…, to available precision.
math.e	The mathematical constant e = 2.718281…, to available precision.
math.tau	The mathematical constant τ = 6.283185…, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius.
math.inf	A floating-point positive infinity. (For negative infinity, use -math.inf.) Equivalent to the output of float('inf').
math.nan	A floating-point "not a number" (NaN) value. Equivalent to the output of float('nan').

Note: Python floats typically carry no more than 53 bits of precision.

Program:

```
#Program to demonstrate built-in math function.

import math

#math.ceil(x) function.
print('Ceiling function:', math.ceil(23.56))
print('Ceiling function:', math.ceil(-23.56))
print('Ceiling function:', math.ceil(12.32))
print('Ceiling function:', math.ceil(-12.32))

print('\n')

#math.floor(x) function.
print('Floor function:', math.floor(23.56))
print('Floor function:', math.floor(-23.56))
print('Floor function:', math.floor(12.32))
print('Floor function:', math.floor(-12.32))

print('\n')

#math.factorial(x) function.
print('Factorial Function:', math.factorial(15))
#print('Factorial Function:', math.factorial(5.2))    --> Gives a valueError exception
#print('Factorial Function:', math.factorial(-75.34)) --> Gives a valueError exception
```

```
print('\n')

#math.fabs(x) function.
print('Fabs Function:', math.fabs(-876.54))
print('Fabs Function:', math.fabs(-7235))

print('\n')

#math.gcd(x,y) function.
print('Greatest Common Divisor:', math.gcd(10,24))
#print('Greatest Common Divisor:', math.gcd(12.30,24))   --> Gives an error
print('Greatest Common Divisor:', math.gcd(-10,24))

print('\n')

#math.exp(x) function i.e., e**x.
print('Exponent function:', math.exp(34))
print('Exponent function:', math.exp(-67.45))

print('\n')

#math.pow(x,y) function i.e., x**y.
print('Power function:', math.pow(2,31))
print('Power function:', math.pow(2,-8))
print('Power function:', math.pow(12.5,53.5))

print('\n')

#math.sqrt(x) fubction.
print('Square root function:', math.sqrt(625))
#print('Square root function:', math.sqrt(-3984))    --> Gives ValueError exception
print('Square root function:', math.sqrt(2467.38))

print('\n')

print('Sin function:', math.sin(64.47))
print('Cos function:', math.cos(64.47))
print('Tan function:', math.tan(64.47))

print('\n')

#math.log2(x) function.
```

```
print('Log2 function:', math.log2(25))
#print('Log2 function:', math.log2(-25.8))  --> Gives ValueError exception
print('Log2 function:', math.log2(43.769))

print('\n')

#math.log10(x) function.
print('Log10 function:', math.log10(25))
#print('Log10 function:', math.log10(-25.8))  --> Gives ValueError exception
print('Log10 function:', math.log10(43.769))

print('\n')

print('Degrees to radians:', math.radians(90))
print('Radians to degrees:', math.degrees(1.5707))

print('\n')

#Constants
print('The value of pi:', math.pi)
print('The value of e:', math.e)
print('The value of tau:', math.tau)
```

Output:

```
Ceiling function: 24
Ceiling function: -23
Ceiling function: 13
Ceiling function: -12

Floor function: 23
Floor function: -24
Floor function: 12
Floor function: -13

Factorial Function: 1307674368000

Fabs Function: 876.54
Fabs Function: 7235.0

Greatest Common Divisor: 2
Greatest Common Divisor: 2
```

```
Exponent function: 583461742527454.9
Exponent function: 5.0913997350542235e-30

Power function: 2147483648.0
Power function: 0.00390625
Power function: 4.8382209306170583e+58

Square root function: 25.0
Square root function: 49.67272893651002

Sin function: 0.9977328054584116
Cos function: -0.06729969473992774
Tan function: -14.825220371563946

Log2 function: 4.643856189774724
Log2 function: 5.451837517668948

Log10 function: 1.3979400086720377
Log10 function: 1.6411666243046132

Degrees to radians: 1.5707963267948966
Radians to degrees: 89.9944808811984

The value of pi: 3.141592653589793
The value of e: 2.718281828459045
The value of tau: 6.283185307179586
```

4.8 RANDOM NUMBERS

- Python offers random module that can generate random numbers.

1. For Integers:

Syntax :

random.randrange (start(opt), stop, step(opt))

Parameters :

start(opt) : Number consideration for generation starts from this, default value is 0. This parameter is optional.

stop : Numbers less than this are generated. This parameter is mandatory.

step(opt) : Step point of range, this won't be included. This is optional. Default value is 1.

Return Value :

This function generated the numbers in the sequence start-stop skipping step.

Exceptions :

Raises ValueError if **stop <= start** and number is **non- integral**.

✓ **random.randrange(stop)**

```
import random

print(random.randrange(30))
```

Output:

I. 5
II. 2
III. 21
IV. 27

✓ **random.randrange(*start*, *stop*[, *step*])**

Return a randomly selected element from range(start, stop, step).

Note: [] means optional.

Program

```
import random

# Using randrange() to generate numbers from 50-100
print ("\nRandom number from 50-100 is : ")
print (random.randrange(50,100))

# Using randrange() to generate numbers from 50-100 skipping 5
print ("\nRandom number from 50-100 skip 5 is : ")
print (random.randrange(50,100,5))

#Throws exception
#print (random.randrange(50,10))
#print (random.randrange(50.3,100))
```

Output:

I. **Random number from 50-100 is :**
95

Random number from 50-100 skip 5 is :

75

II. Random number from 50-100 is :
60

Random number from 50-100 skip 5 is :
55

✓ **random.randint(a,b)**

Return a random integer *N* such that a <= N <= b

```
import random

print(random.randint(1,100))
```

→ Inclusive of 1 and 100.

Output:

I. 96
II. 76
III. 15

2. For Real/Floating:

✓ **random.random()**

Return the next random floating point number in the range (0.0, 1.0) i.e., excluding 1.0.

```
import random

print(random.random())
```

Output:

I. 0.48644020748452865
II. 0.9207854756869541
III. 0.8515254897126728

✓ **random.uniform(a,b)**

Return a random floating point number *N* such that a <= N <= b for a <= b and b <= N <= a for b < a.

The end-point value b may or may not be included in the range depending on floating-point rounding in the equation a + (b-a) * random().

```
import random
```

```
print(random.uniform(1,100))
```

Output:

I. 51.433547569607704
II. 90.0503177294139
III. 14.294244323244204

✓ **random.triangular(*low*, *high*, *mode*)**

Return a random floating point number *N* such that low <= N <= high and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero (0.0) and one (1.0). The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

```
import random

print(random.triangular(12.43, 78.87))
print(random.triangular())
print(random.triangular(-12.43, -78.87, 2.4))
```

Output:

I. 28.26005842476202
0.7088800121986798
-57.94680323674265
II. 33.60197377415188
0.5080741473203966
-7.139455370648093
III. 36.7400365668324
0.6532371234185053
-34.338098776806014

3. For Sequences

✓ **random.choice(seq)**

Return a random element from the non-empty sequence *seq (*list, tuple or string). If *seq* is empty, raises Index Error.

Program:

```
import random

color_list = ['Red', 'Blue', 'Green', 'White', 'Black']
print(random.choice(color_list))
```

```
print(random.choice([1, 2, 3, 4, 5,6]))

print(random.choice('Python Application Programming'))

my_list = [2, 109, False, 10, "Lorem", 482, "Ipsum"]
print(random.choice(my_list))
```

Output:

I. Black
   1
   y
   Lorem
II. Green
   5
   m
   10
III. White
   1
   y
   482

✓ **random.shuffle(*x*[, *random*])**

Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function random().

Program:

```
import random

color_list = ['Red', 'Blue', 'Green', 'White', 'Black']
random.shuffle(color_list)
print(color_list)

list = [1, 2, 3, 4, 5,6]
random.shuffle(list)
print(list)

my_list = [2, 109, False, 10, "Lorem", 482, "Ipsum"]
random.shuffle(my_list)
print(my_list)
```

Output:

I. ['Red', 'White', 'Black', 'Blue', 'Green']
[4, 6, 1, 5, 3, 2]
[2, 'Ipsum', 482, 'Lorem', False, 109, 10]

II. ['Red', 'White', 'Blue', 'Black', 'Green']
[3, 6, 5, 2, 1, 4]
[109, 482, 'Lorem', 2, 'Ipsum', False, 10]

III. ['Red', 'Black', 'Blue', 'Green', 'White']
[3, 2, 6, 5, 4, 1]
['Ipsum', 109, False, 'Lorem', 2, 10, 482]

4.9 EXERCISES

Rewrite all the programs mentioned in Module 1 Exercise 3.7 and Module 2 Exercise 1.7 using functions.

4.10 GLOSSARY

Algorithm	A general process for solving a category of problems.
Argument	A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
Body	The sequence of statements inside a function definition.
Composition	Using an expression as part of a larger expression, or a statement as part of a larger statement.
Deterministic	Pertaining to a program that does the same thing each time it runs, given the same inputs.
dot notation	The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.
flow of execution	The order in which statements are executed during a program run.
fruitful function	A function that returns a value.
Function	A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.
function call	A statement that executes a function. It consists of the function name followed by an argument list.
function definition	A statement that creates a new function, specifying its name, parameters, and the statements it executes.
function object	A value created by a function definition. The name of the function is a variable that refers to a function object.
Header	The first line of a function definition.
import statement	A statement that reads a module file and creates a module object.
module object	A value created by an import statement that provides access to the data and code defined in a module.

Parameter	A name used inside a function to refer to the value passed as an argument.
Pseudorandom	Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.
return value	The result of a function. If a function call is used as an expression, the return value is the value of the expression.
void function	A function that does not return a value.

QUESTIONS

1. Explain computer hardware architecture with a neat diagram.
2. Difference between compiler and interpreter.
3. What is a program? Explain the building blocks of a program.
4. Explain different types of errors.
5. What is Python? List the features and applications of Python.
6. What are values? Explain the different types of values with example.
7. What is a variable? Explain the different rules in naming a variable. Also give examples of valid and invalid variable names.
8. What is a keyword? List the keywords in Python.
9. Define
    a. Statement
    b. Expression
    c. Boolean Expression
10. What are operators and operands? Explain with an example.
11. Explain different types of operators with an example for each.
12. Explain rules of precedence (Order of operations) in Python.
13. What is modulus operator used for?
14. Explain with an example how '+' operator works on strings.
15. Explain the print() and input() function in Python with different examples (one example for each type)
16. Explain
    a. Conditional execution (if statement)
    b. Alternate execution (if-else statement)
    c. Chained conditionals (else-if ladder/cascaded if-else statement)
    d. Nested conditionals (nested if-else statements)
17. Explain with an example how exception are handled using try and except.
18. Explain short-circuit evaluation of logical expression with an example.
19. What is a function? What are the different types of functions? What are the advantages of using functions?
20. Explain how to define and call a user-defined function with example.
21. Explain fruitful and void functions with an example.
22. Explain any 10 function present in math module with an example.
23. Explain type conversion functions with examples.

24. Explain how random numbers can be generated for integers, floating(real) and sequences with examples.

Extra

To accept any number of arguments where each of them needs to be a `int` or `float` instead of the first being a sequence. This would only require to add one `*` to the function declaration.

```
def f(*x):
    return sum(c)
```