UPCISS

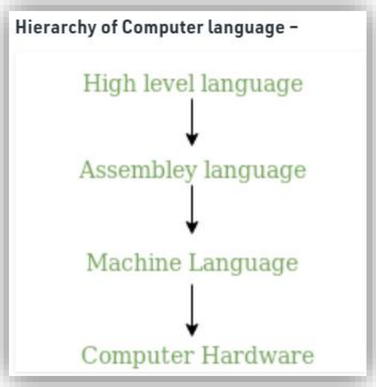
O-Level (M3-R5)

Programming and Problem Solving Through Python Language

Video Tutorial
O-Level M3-R5 Full Video Playlist Available on YouTube Channel **UPCISS**

Introduction to Programming

Before we understand what programming is, you must know what a computer is. A computer is a device that can accept human instruction, processes it, and responds to it or a computer is a computational device that is used to process the data under the control of a computer program. Program is a sequence of instruction along with data.



The basic components of a computer are:

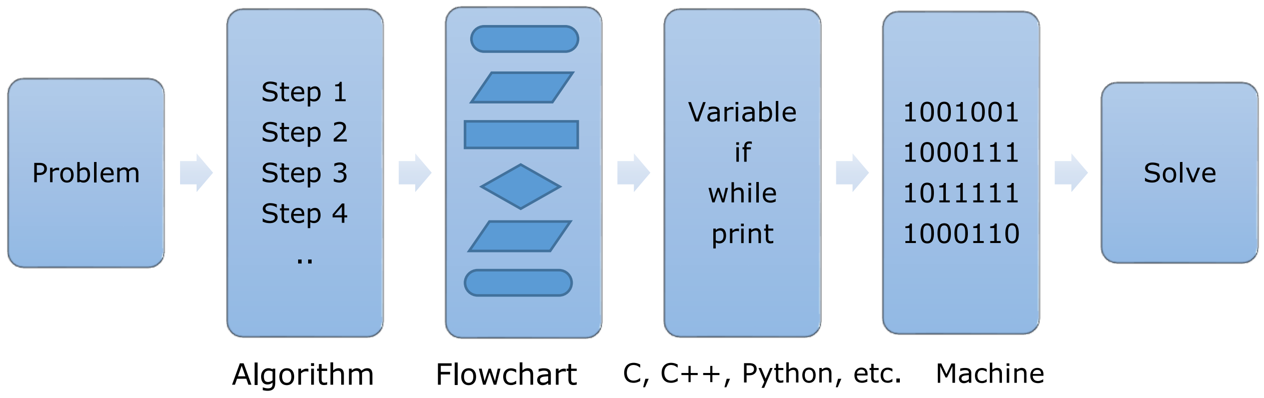
- Input unit
- Central Processing Unit(CPU)
- Output unit

The CPU is further divided into three parts-

- Memory unit
- Control unit
- Arithmetic Logic unit

Most of us have heard that CPU is called the brain of our computer because it accepts data, provides temporary memory space to it until it is stored (saved) on the hard disk, performs logical operations on it and hence processes (here also means converts) data into information. We all know that a computer consists of hardware and software. Software is a set of programs that performs multiple tasks together. An operating system is also software (system software) that helps humans to interact with the computer system.

A program is a set of instructions given to a computer to perform a specific operation. Or computer is a computational device that is used to process the data under the control of a computer program. While executing the program, raw data is processed into the desired output format. These computer programs are written in a programming language which are high-level languages. High level languages are nearly human languages that are more complex than the computer understandable language which are called machine language, or low level language. So after knowing the basics, we are ready to create a very simple and basic program. Like we have different languages to communicate with each other, likewise, we have different languages like C, C++, C#, Java, python, etc. to communicate with the computers. The computer only understands binary language (the language of 0's and 1's) also called machine-understandable language or low-level language but the programs we are going to write are in a high-level language which is almost similar to human language.



Basic Model of Computation

Problem definition:

A problem definition involves the clear identification of the problem in terms of available input parameters and desired solution.

Approach towards solving the problem:

After a problem is identified, the user needs to implement a step-by-step solution in terms of algorithms.

Graphical representation of problem solving sequence:

This step involves representing the steps of algorithm pictorially by using a flowchart. Each component of the flowchart presents a definite process to solve the problem.

Converting the sequence in a programming language:

Converting the graphical sequence of processes into a language that the user and the computer can understand and use for problem solving is called programming. After the program is compiled the user can obtain the desired solution for the problem by executing the machine language version of the program.

Algorithm

A sequential solution of any program that written in human language, called algorithm. Algorithm is first step of the solution process, after the analysis of problem, programmer writes the algorithm of that problem.

It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output. The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Characteristics of Algorithm:

- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon with the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and looping statements are difficult to show in Algorithms.

Example of Algorithm?

Algorithm for add two numbers entered by the user.

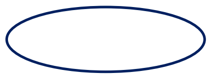
- Step 1: Start
- Step 2: Declare 3 variables num1, num2 and sum.
- Step 3: Read values num1 and num2.
- Step 4: Add num1 and num2 and assign the result to sum.
  sum ← num1+num2
- Step 5: Print sum
- Step 6: Stop

Flowchart

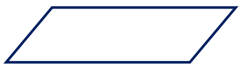
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols used in Flowchart Designs

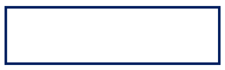
- **Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



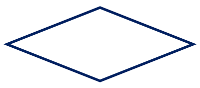
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



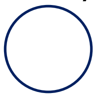
- **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.



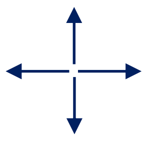
- **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



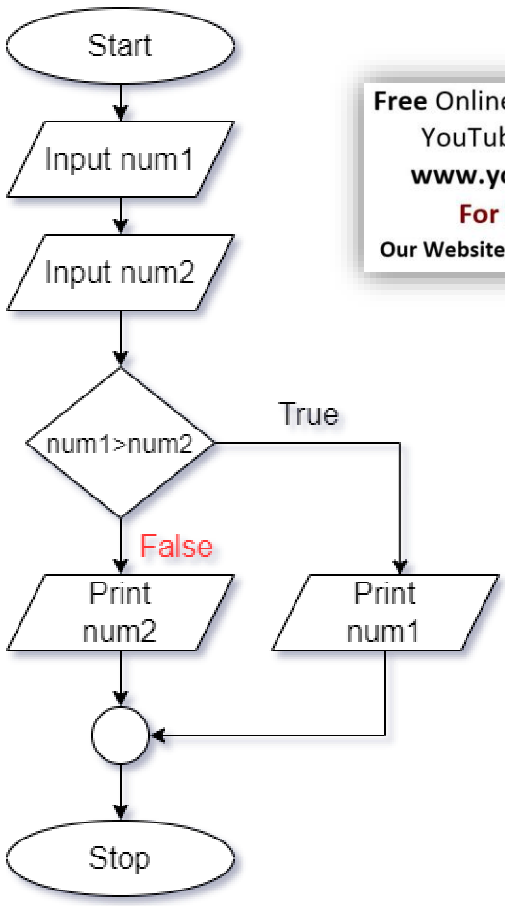
- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



Advantages of Flowchart:
- Flowcharts are better way of communicating the logic of system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts helps in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- It provides better documentation.
- Flowcharts serve as a good proper documentation.

Disadvantages of Flowchart:
- It is difficult to draw flowchart for large and complex programs.
- In this there is no standard to determine the amount of detail.
- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.

Example: Draw a flowchart to input two numbers from user and display the larger of two numbers



Programming Language

As we know, to communicate with a person, we need a specific language, similarly to communicate with computers, programmers also need a language is called Programming language.
Before learning the programming language, let's understand what is language?

What is Language?
Language is a mode of communication that is used to **share ideas, opinions with each other**. For example, if we want to teach someone, we need a language that is understandable by both communicators.

What is a Programming Language?
A programming language is a **computer language** that is used by **programmers (developers) to communicate with computers**. It is a set of instructions written in any specific language (C, C++, Java, Python, etc.) to perform a specific task.
A programming language is mainly used to **develop desktop applications, websites, and mobile applications.**

Characteristics of a programming Language

- A programming language must be simple, easy to learn and use, have good readability, and be human recognizable.
- Abstraction is a must-have Characteristics for a programming language in which the ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development.
- Necessary tools for the development, debugging, testing, and maintenance of a program must be provided by a programming language.
- A programming language should provide a single environment known as Integrated Development Environment (IDE).
- A programming language must be consistent in terms of syntax and semantics.

Most Popular Programming Languages

- C
- Python
- C++
- Java
- SCALA
- C#
- R
- Ruby
- Go
- Swift
- JavaScript

Compilation

The compilation is a process of converting the source code into object code. It is done with the help of the compiler or interpreter. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

Introduction of Compiler Design

The compiler is software that converts a program written in a high-level language (Source Language) to low-level language (Object/Target/Machine Language).

- Cross Compiler that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one on which the compiler is running.
- Source-to-source Compiler or transcompiler or transpiler is a compiler that translates source code written in one programming language into the source code of another programming language.

Language processing systems (using Compiler) –

We know a computer is a logical assembly of Software and Hardware. The hardware knows a language that is hard for us to grasp, consequently, we tend to write programs in a high-level language that is much less complicated for us to comprehend and maintain in thoughts. Now, these programs go through a series of transformations so that they can readily be used by machines. This is where language procedure systems come in handy.

- **High-Level Language –** If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called preprocessor directives. They direct the pre-processor about what to do.
- **Pre-Processor –** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc.
- **Assembly Language –** It's neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler –** For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. Its translate assembly language to machine code.
- **Interpreter –** An interpreter converts high-level language into low-level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing, and executes the source code whereas the interpreter does the same line by line. Compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower with respect to compiled ones.
- **Relocatable Machine Code –** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
- **Loader/Linker –** It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

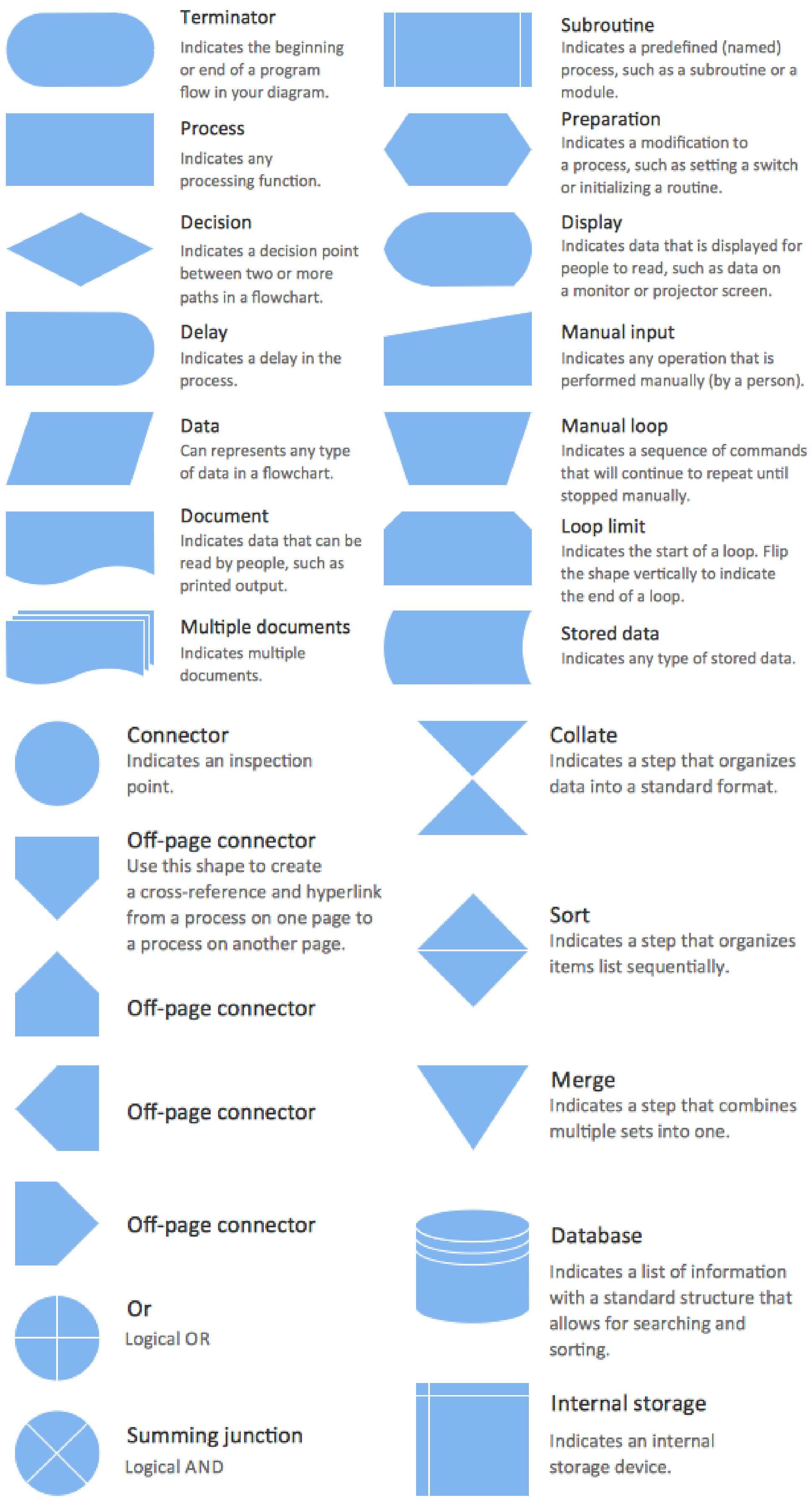
Differences between Testing and Debugging

Testing:
Testing is the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by its design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.
Debugging:
Debugging is the process of fixing a bug in the software. It can defined as the identifying, analyzing and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

Testing	Debugging
Testing is the process to find bugs and errors.	Debugging is the process to correct the bugs found during testing.
It is the process to identify the failure of implemented code.	It is the process to give the absolution to code failure.
Testing is the display of errors.	Debugging is a deductive process.
Testing is done by the tester.	Debugging is done by either programmer or developer.
There is no need of design knowledge in the testing process.	Debugging can't be done without proper design knowledge.
Testing can be done by insider as well as outsider.	Debugging is done only by insider. Outsider can't do debugging.
Testing can be manual or automated.	Debugging is always manual. Debugging can't be automated.
It is based on different testing levels i.e. unit testing, integration testing, system testing etc.	Debugging is based on different types of bugs.
Testing is a stage of software development life cycle (SDLC).	Debugging is not an aspect of software development life cycle, it occurs as a consequence of testing.
Testing is composed of validation and verification of software.	While debugging process seeks to match symptom with cause, by that it leads to the error correction.
Testing is initiated after the code is written.	Debugging commences with the execution of a test case.

Algorithms and Flowcharts

Flow Chart Symbols



Terminator
Indicates the beginning or end of a program flow in your diagram.

Process
Indicates any processing function.

Decision
Indicates a decision point between two or more paths in a flowchart.

Delay
Indicates a delay in the process.

Data
Can represents any type of data in a flowchart.

Document
Indicates data that can be read by people, such as printed output.

Multiple documents
Indicates multiple documents.

Subroutine
Indicates a predefined (named) process, such as a subroutine or a module.

Preparation
Indicates a modification to a process, such as setting a switch or initializing a routine.

Display
Indicates data that is displayed for people to read, such as data on a monitor or projector screen.

Manual input
Indicates any operation that is performed manually (by a person).

Manual loop
Indicates a sequence of commands that will continue to repeat until stopped manually.

Loop limit
Indicates the start of a loop. Flip the shape vertically to indicate the end of a loop.

Stored data
Indicates any type of stored data.

Connector
Indicates an inspection point.

Off-page connector
Use this shape to create a cross-reference and hyperlink from a process on one page to a process on another page.

Off-page connector

Off-page connector

Off-page connector

Or
Logical OR

Summing junction
Logical AND

Collate
Indicates a step that organizes data into a standard format.

Sort
Indicates a step that organizes items list sequentially.

Merge
Indicates a step that combines multiple sets into one.

Database
Indicates a list of information with a standard structure that allows for searching and sorting.

Internal storage
Indicates an internal storage device.

Let’s see the difference between algorithm and flow chart:

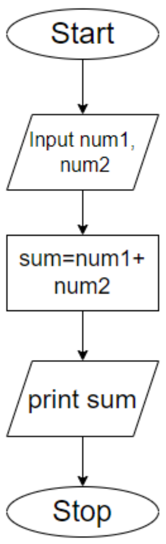
Algorithm	Flowchart
Algorithm is step by step procedure to solve the problem.	Flowchart is a diagram created by different shapes to show the flow of data.
Algorithm is complex to understand.	Flowchart is easy to understand.
In algorithm plain text are used.	In flowchart, symbols/shapes are used.
Algorithm is easy to debug.	Flowchart it is hard to debug.
Algorithm is difficult to construct.	Flowchart is simple to construct.
Algorithm does not follow any rules.	Flowchart follows rules to be constructed.
Algorithm is the pseudo code for the program.	Flowchart is just graphical representation of that logic.

Sequential processing algorithms and flowchart.

Algorithm for add two numbers entered by the user.

- Step 1: Start
- Step 2: Declare 3 variables num1, num2 and sum.
- Step 3: Read values num1 and num2.
- Step 4: Add num1 and num2 and assign the result to sum.
  sum ← num1+num2
- Step 5: Print sum
- Step 6: Stop

Flowchart for add two numbers entered by the user.

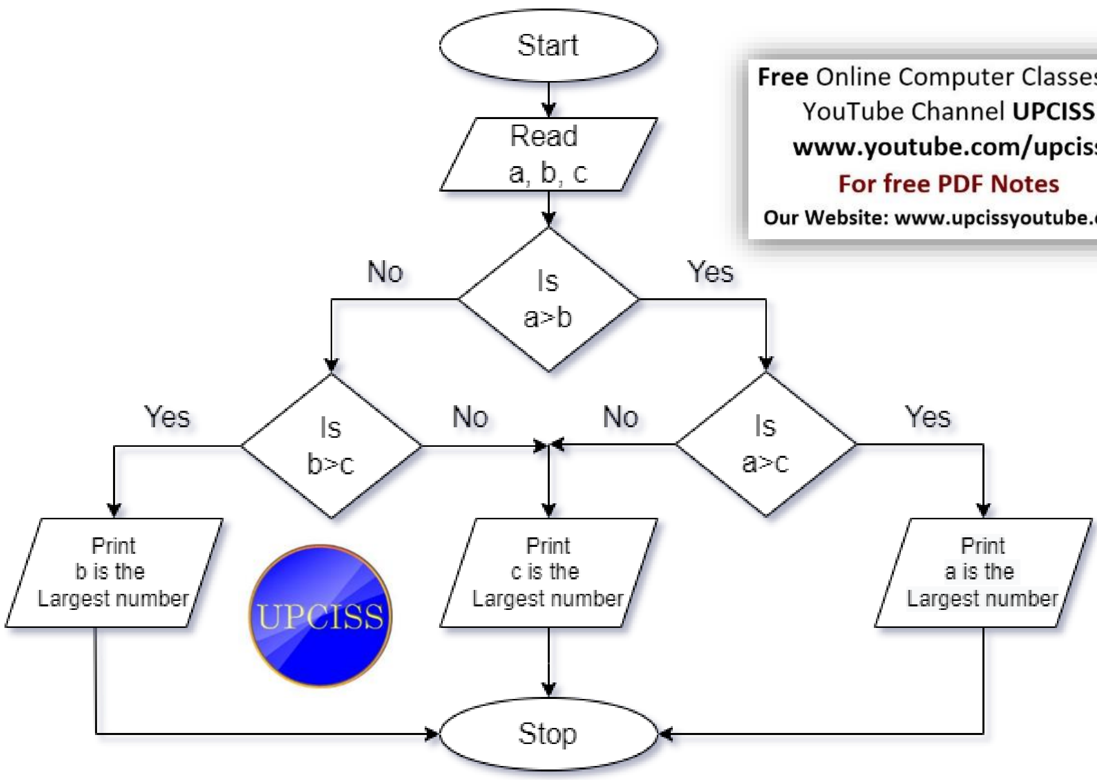


Decision based processing algorithms and flowchart.

Algorithm for find the largest among three different numbers entered by the user.

```
• Step 1: Start
• Step 2: Declare variables a, b and c.
• Step 3: Read variables a, b and c.
• Step 4: If a>b
                  If a>c
                         Print a is the largest number.
                  Else
                         Print c is the largest number.
             Else
                  If b> c
                         Print b is the largest number.
                  Else
                         Print c is the largest number.
• Step 5: Stop
```

Flowchart for find the largest among three different numbers entered by the user.

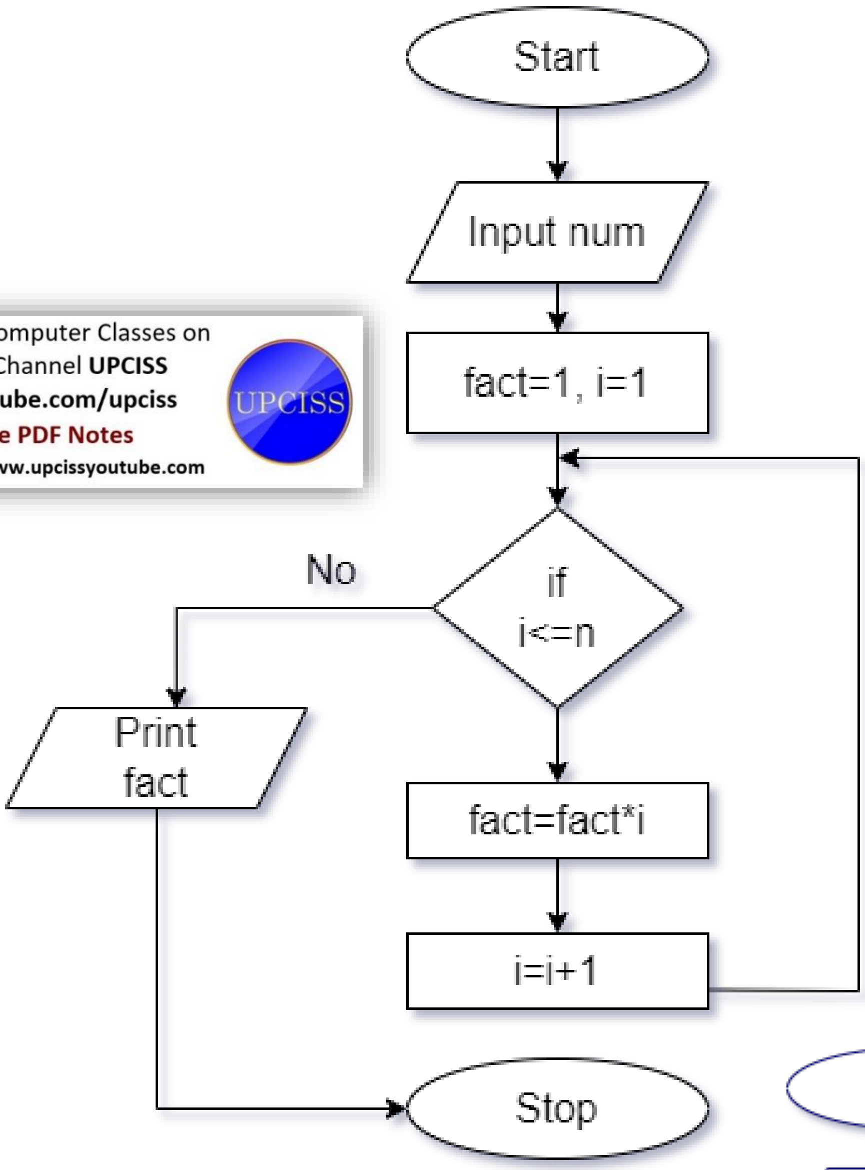

Start
Read a, b, c
Is a>b
No
Yes
Yes
Is b>c
No
No
Is a>c
Yes
Print b is the Largest number
Print c is the Largest number
Print a is the Largest number
Stop

Iterative processing algorithms and flowchart.

Algorithm for find the factorial of a number entered by the user.

```
• Step 1: Start
• Step 2: Declare variables num, fact and i.
• Step 3: Read value of num
• Step 4: initialize variables fact ←1, i ←1
• Step 5: Repeat the steps until i<=num
       fact ← fact*i
       i ← i+1
• Step 6: Print fact
• Step 7: Stop
```

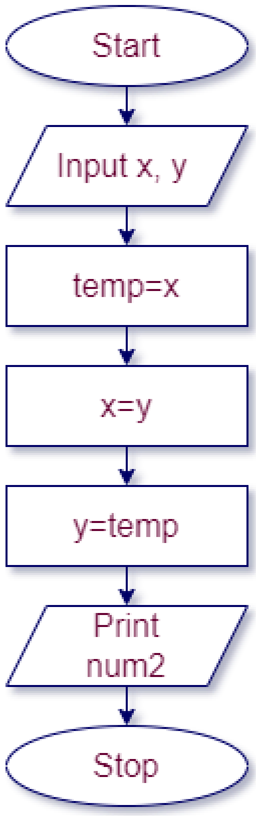
Flowchart for find the factorial of a number entered by the user.



Some Examples of algorithm and flowchart

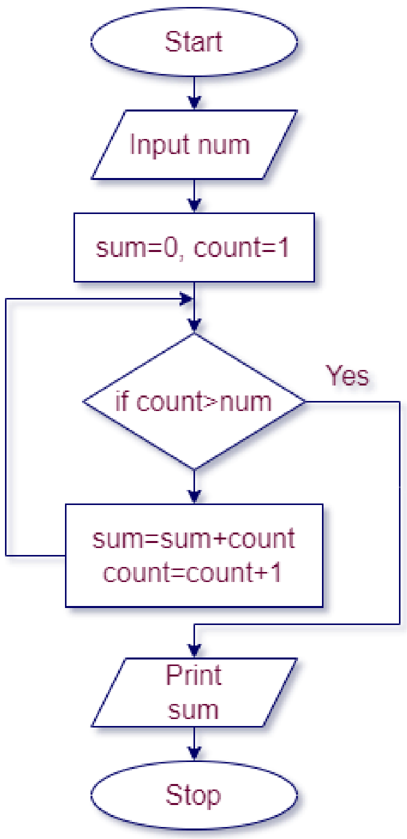
Algorithm and flowchart for exchanging values of two variables.

- Step 1: Start
- Step 2: Declare variables x, y and temp.
- Step 3: Read values x and y.
- Step 4: temp ← x
- Step 5: x ← y
- Step 6: y ← temp
- Step 7: Display x & y
- Step 8: Stop



Algorithm and flowchart for summation of a set of numbers.

- Step 1: Start
- Step 2: Declare variables count, num and sum.
- Step 3: Read the value of num.
- Step 4: Initialize sum as 0 and count as 1.
- Step 5: If count>num (go to step 8)
- Step 6: sum = sum + count
- Step 7: count = count + 1 (go to step 5)
- Step 8: Print the value of sum
- Step 9: Stop



Algorithm for Decimal Base to Binary Base conversion.

1. Store the remainder when the number is divided by 2 in an array.
2. Divide the number by 2
3. Repeat the above two steps until the number is greater than zero.
4. Print the array in reverse order now.

For Example:

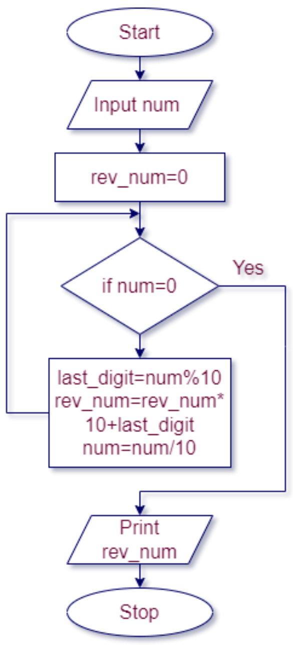
If the decimal number is 10.

- Step 1: Remainder when 10 is divided by 2 is zero. (Therefore, arr[0] = 0.)
- Step 2: Divide 10 by 2. New number is 10/2 = 5.
- Step 3: Remainder when 5 is divided by 2 is 1. (Therefore, arr[1] = 1.)
- Step 4: Divide 5 by 2. New number is 5/2 = 2.
- Step 5: Remainder when 2 is divided by 2 is zero. (Therefore, arr[2] = 0.)
- Step 6: Divide 2 by 2. New number is 2/2 = 1.
- Step 7: Remainder when 1 is divided by 2 is 1. (Therefore, arr[3] = 1.)
- Step 8: Divide 1 by 2. New number is 1/2 = 0.
- Step 9: Since number becomes = 0. Print the array in reverse order.

Therefore the equivalent binary number is 1010.

Algorithm and flowchart for reversing digits of an integer.

- Step 1: Start
- Step 2: Read the value of num.
- Step 3: rev_num = 0
- Step 4: if num=0 (go to step 8)
- Step 5: last_digit = num%10
- Step 6: rev_num=rev_num * 10 + last_digit
- Step 7: num = num / 10 (go to step 4)
- Step 8: Print rev_num
- Step 9: Stop



Algorithm and flowchart for GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers.

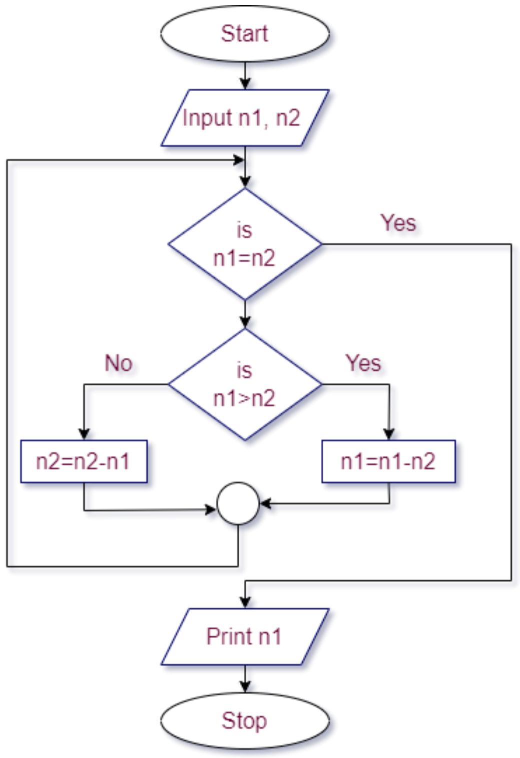
- Step 1: Start
- Step 2: Read n1, n2
- Step 3: If n1=n2

  Then go to step 5
- Step 4: If n1>n2

  n1 ← n1- n2 go to step 3

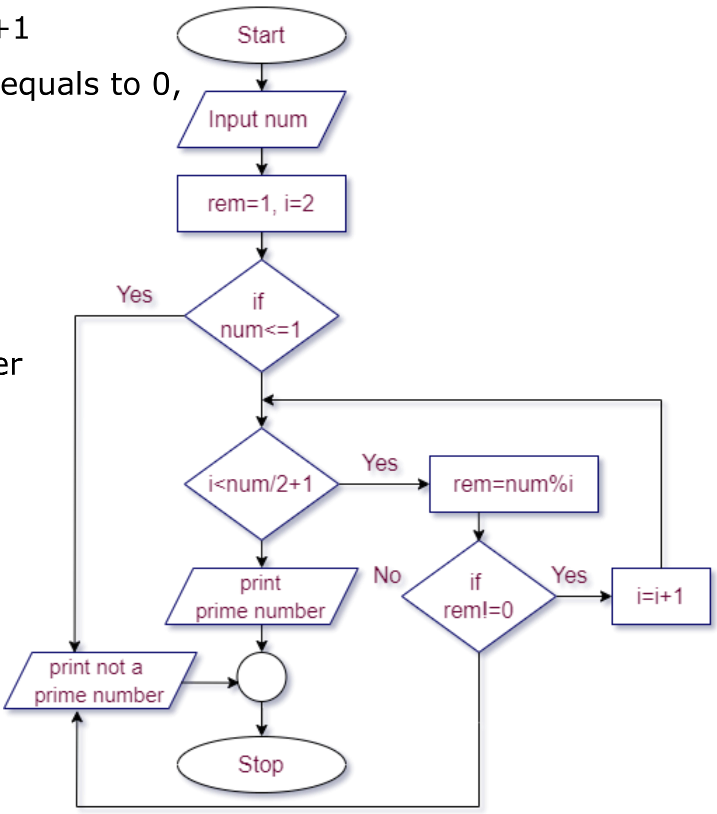
  else

  n2 ← n2 – n1 go to step 3
- Step 5: Print n1
- Step 6: Stop



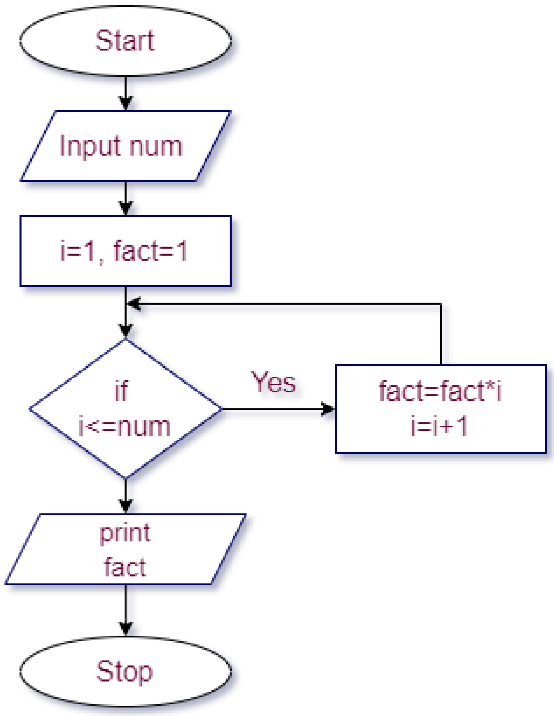
Algorithm and flowchart for Test whether a number is prime or not.

```
Step 1: Start
Step 2: Read num from user
Step 3: Initialize variables rem←1, i←2
Step 4: If num<=1        // any number less than 1 is not a prime number
            Display num is not a prime number
            Go to step 7
Step 5: Repeat the steps until i<num/2+1
            If remainder of number divide i equals to 0,
                Set rem=0
                Go to step 6
            i ← i+1
Step 6: If rem=0
            Display num is not prime number
        Else
            Display num is prime number
Step 7: Stop
```



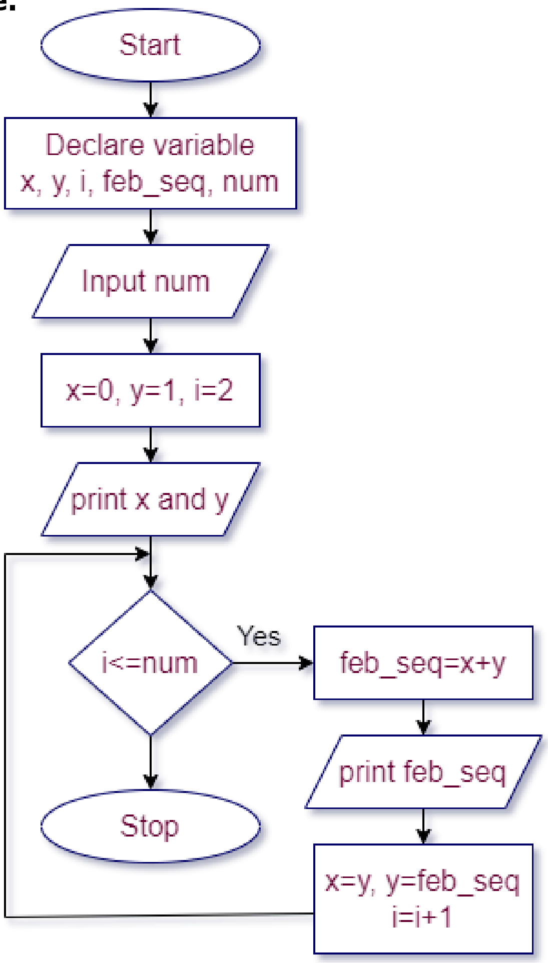
Algorithm and flowchart for factorial computation.

```
Step 1: Start
Step 2: Read num from user
Step 3: Initialize variables i←1, fact←1
Step 4: Repeat this step until i<=num
            fact ← fact*i
            i ← i+1
Step 5: Display fact
Step 6: Stop
```



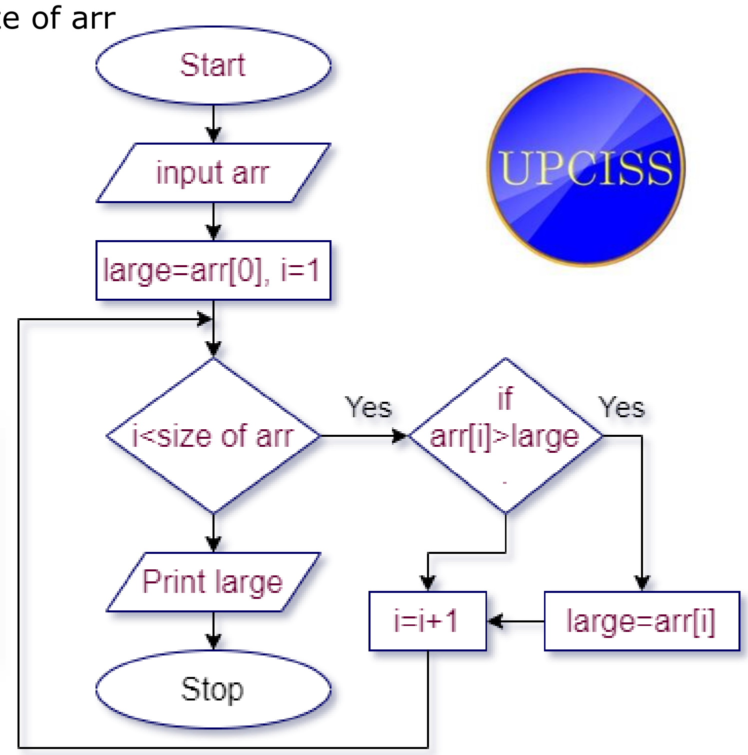
Algorithm and flowchart for Fibonacci sequence.

```
Step 1: Start
Step 2: Declare variable x, y, i, feb_seq, num
Step 3: Read num from user
Step 4: Initialize variable x←0, y←1 and i←2
Step 5: Print x and y
Step 6: Repeat until i<=num:
        Step 6.1: feb_seq =x + y
        Step 6.2: print feb_seq
        Step 6.3: x = y, y = feb_seq
        Step 6.4: i = i + 1
Step 7: Stop
```



Algorithm and flowchart for Find largest number in an array.

```
Step 1: Start
Step 2: input the array element as arr.
Step 3: Initialize large ← arr[0] and i ← 1
Step 4: Repeat this step until i < size of arr
        If arr[i] > large
                large ← arr[i]
        i ← i+1
Step 5: print large
Step 6: Stop
```



Algorithm and flowchart for Reverse order of elements of an array.

Step 1: Start

Step 2: Read arr

Step 3: Place the two pointers (let start and end) at the start and end of the array.

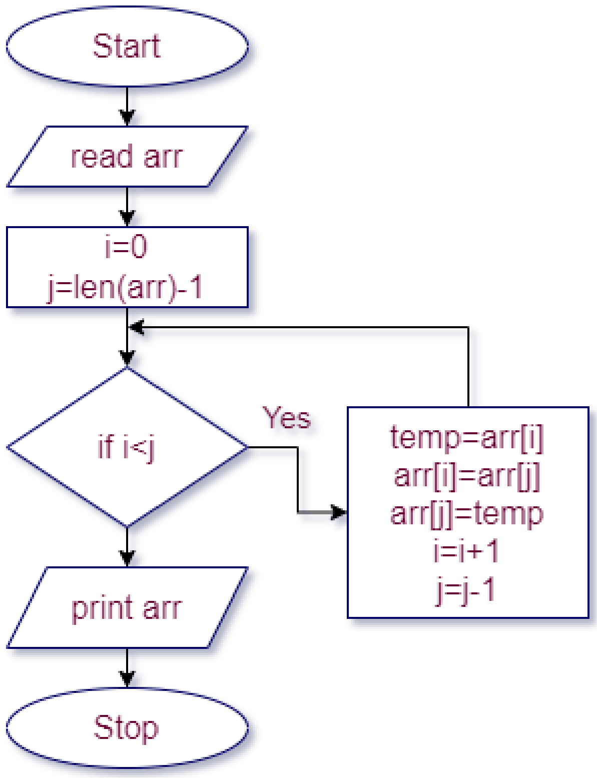
Step 4: Swap arr[start] and arr[end] using temp variable.

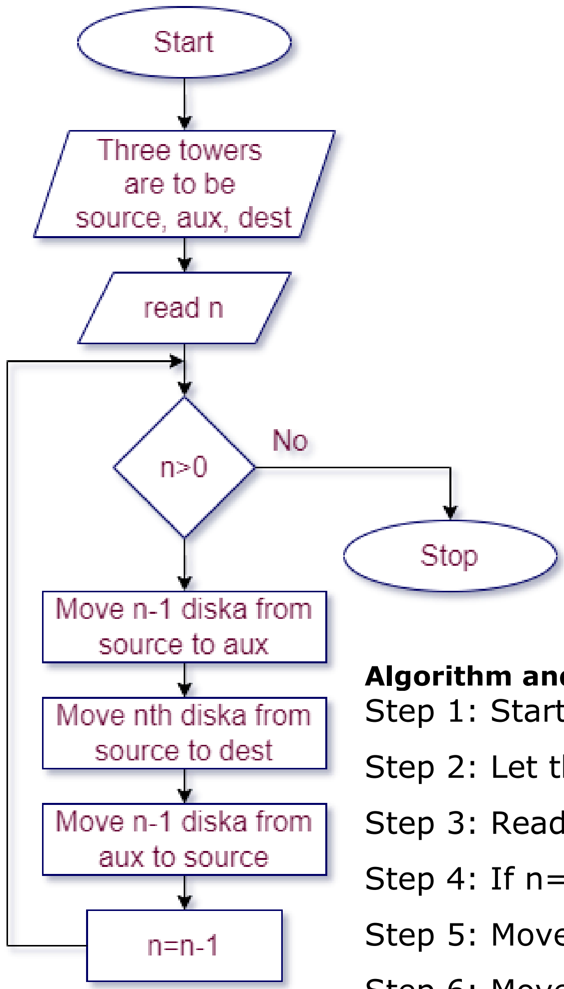
Step 5: Increment start and decrement end with 1

Step 6: If start reached to the start >= end, then terminate otherwise repeat from step 4 and step 5.

Step 7: Display arr

Step 8: Stop




Start
Three towers are to be source, aux, dest
read n
n>0
No
Stop
Move n-1 diska from source to aux
Move nth diska from source to dest
Move n-1 diska from aux to source
n=n-1

Algorithm and flowchart for Tower of Hanoi.

Step 1: Start

Step 2: Let the three towers be the source, dest, aux.

Step 3: Read the number of disks, n from the user.

Step 4: If n=0 go to step 9

Step 5: Move n-1 disks from source to aux.

Step 6: Move nth disk from source to dest.

Step 7: Move n-1 disks from aux to dest.

Step 8: Repeat Steps 5 to 7, by decrementing n by 1.

Step 9: Stop

Introduction to Python

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.
It is used for:

- web development (server-side),
- software development,
- mathematics,
- System scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Good to know

- The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.
- In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

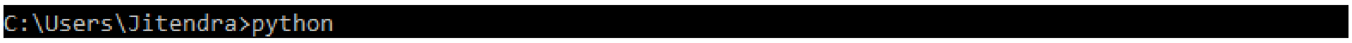
Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

The Python Command Line

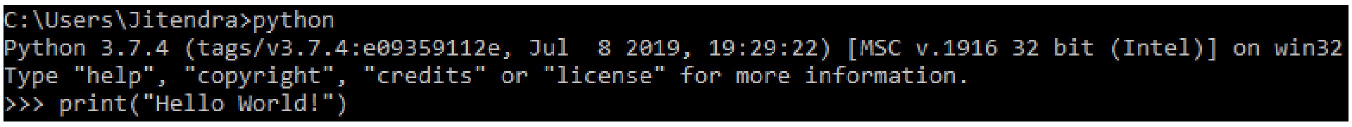
To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

Type the following on the Windows, Mac or Linux command line:



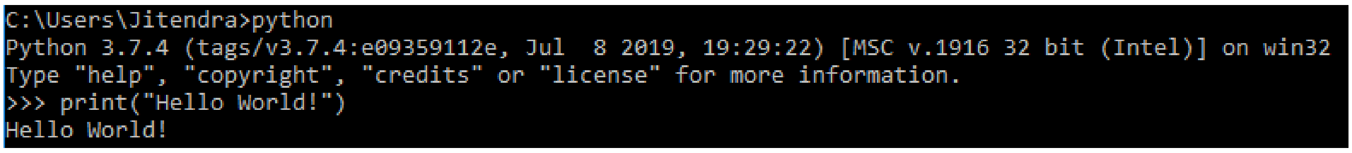
```
C:\Users\Jitendra>python
```

From there you can write any python, including our hello world example from earlier in the tutorial:



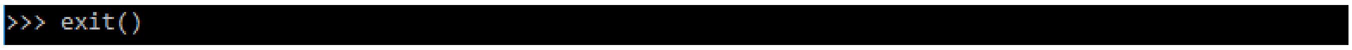
```
C:\Users\Jitendra>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
```

Which will write "Hello, World!" in the command line:



```
C:\Users\Jitendra>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
```

Whenever you are done in the python command line, you can simply type the following to quit the python command line interface:

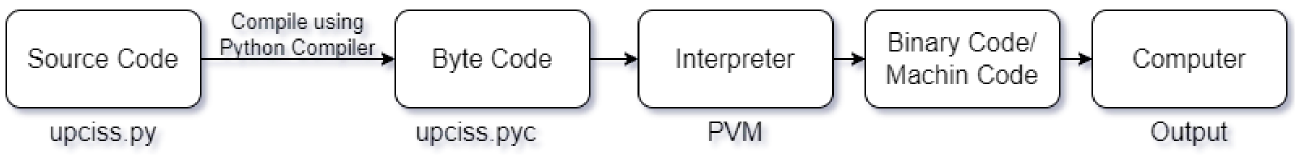


```
>>> exit()
```

Python Virtual Machine (PVM)

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instructions into machine code so the computer can execute those machine code instructions and display the output.

Interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution.



Python Variables

Creating Variables

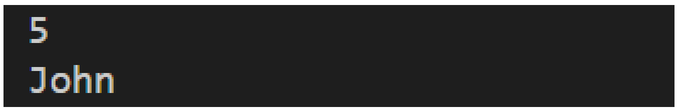
Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Example

```
x = 5
y = "John"
print(x)
print(y)
```



```
5
John
```

Variables do not need to be declared with any particular type and can even change type after they have been set.

Example

```
x = 4 # x is of type int
x = "Sally" # x is now of type str
print(x)
```

String variables can be declared either by using single or double quotes:

Example

```
x = "John"
# is the same as
x = 'John'
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Remember that variable names are case-sensitive

Assign Value to Multiple Variables

Python allows you to assign values to multiple variables in one line:

Example

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
print(z)
```

```
Orange
Banana
Cherry
```

And you can assign the *same* value to multiple variables in one line:

Example

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

Output Variables

The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the `+` character:

Example

```
x = "awesome"
print("Python is " + x)
```

```
Python is awesome
```

You can also use the `+` character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z =  x + y
print(z)
```

```
Python is awesome
```

For numbers, the `+` character works as a mathematical operator:

Example

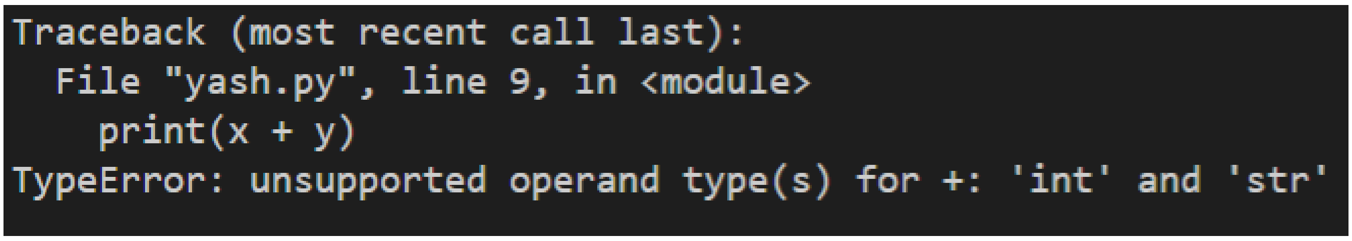
```
x = 5
y = 10
print(x + y)
```

```
15
```

If you try to combine a string and a number, Python will give you an error:

Example

```
x = 5
y = "John"
print(x + y)
```



```
Traceback (most recent call last):
  File "yash.py", line 9, in <module>
    print(x + y)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python Comments

Comments can be used to explain Python code.
Comments can be used to make the code more readable.
Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code: Comment shortcut key = ctrl+/

Example

```
#print("Hello, World!")
print("Cheers, Mate!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a # for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Or, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place you comment inside it:

Example

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Numbers

Python Numbers

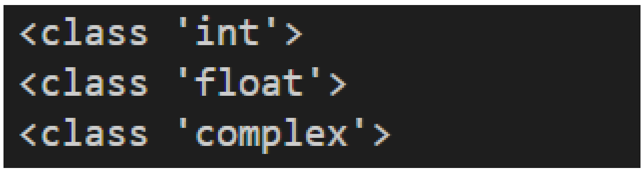
There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
print(type(x))
print(type(y))
print(type(z))
```



```
<class 'int'>
<class 'float'>
<class 'complex'>
```

To verify the type of any object in Python, use the `type()` function:

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example Integers:

```
x = 1
y = 35656222554887711
z = -3255522
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example- Floats:

```
x = 1.10
y = 1.0
z = -35.59
i = 35e3
j = 12E4
k = -87.7e100
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Complex
Complex numbers are written with a "j" as the imaginary part:

Example Complex:

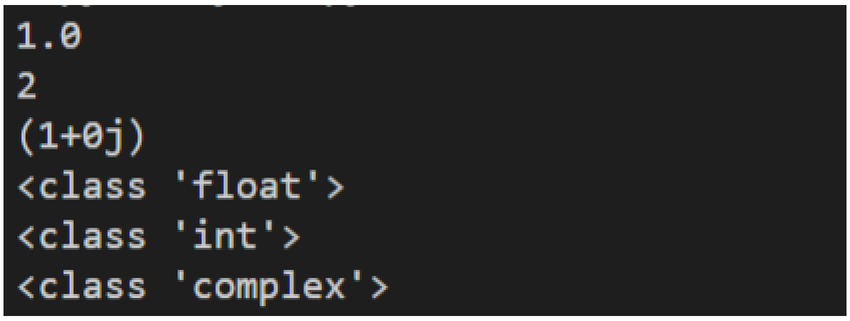
```
x = 3+5j
y = 5j
z = -5j
```

Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

Example Convert from one type to another:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```



```
1.0
2
(1+0j)
<class 'float'>
<class 'int'>
<class 'complex'>
```

Note: You cannot convert complex numbers into another number type.

Python Strings

String Literals

String literals in python are surrounded by either single quotation marks, or double quotation marks.

`'hello'` is the same as `"hello"`.

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Python constants

Sometimes, you may want to store values in variables. But you don't want to change these values throughout the execution of the program.

To do it in other programming languages, you can use constants. The constants like variables but their values don't change during the program executes.

The bad news is that Python doesn't support constants.

To work around this, you use all capital letters to name a variable to indicate that the variable should be treated as a constant. For example:

```
MY_PHONE_NUMBER = 9100000000
```

When encountering variables like these, you should not change their values. These variables are constant by convention, not by rules.

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
```

```
ut labore et dolore magna aliqua.'''
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing Unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1]) # returns "e"
```

Example

Substring. Get the characters from position 2 to position 5 (5 position not included):

```
b = "Hello, World!"
print(b[2:5]) # returns "llo"
print(b[2:5:2]) # returns "lo"
print(b[::-1]) # returns "!dlroW ,olleH"
```

Example The strip() method removes any whitespace from the beginning or the end:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
print(a.lstrip()) # returns "Hello, World! " Removes any whitespace from the beginning.
print(a.rstrip()) # returns " Hello, World!" Removes any whitespace from the end.
```

Example The len() method returns the length of a string:

```
a = "Hello, World!"
print(len(a)) # returns 13
```

Example The lower() method returns the string in lower case:

```
a = "Hello, World!"
print(a.lower())# returns "hello, world!"
```

Example The upper() method returns the string in upper case:

```
a = "Hello, World!"
print(a.upper())# returns "HELLO, WORLD!"
```

Example The title() Converts the first character of each word to upper case:

```
a = "hello, world!"
print(a.title())# returns "Hello, World!"
```

Example The count() Returns the number of times a specified value occurs in a string:

```
a = "hello, world!"
print(a.count("l")) # returns 3
```

Example The find() Searches the string for a specified value and returns the position of where it was found:

```
a = "Hello, World!"
print(a.find("l")) # returns 2
print(a.find("l",4)) # returns 10
```

Example The replace() method replaces a string with another string:

```
a = "Hello, World!"
print(a.replace("H", "J")) # returns "Jello, World!"
```

Example The split() method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

String Format

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

Example

```
age = 36
txt = "My name is John, I am " + age
print(txt) # returns error
```

But we can combine strings and numbers by using the `format()` method!

The `format()` method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:

Example Use the format() method to insert numbers into strings:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age)) # returns "My name is John, and I am 36"
```

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers `{0}` to be sure the arguments are placed in the correct placeholders:

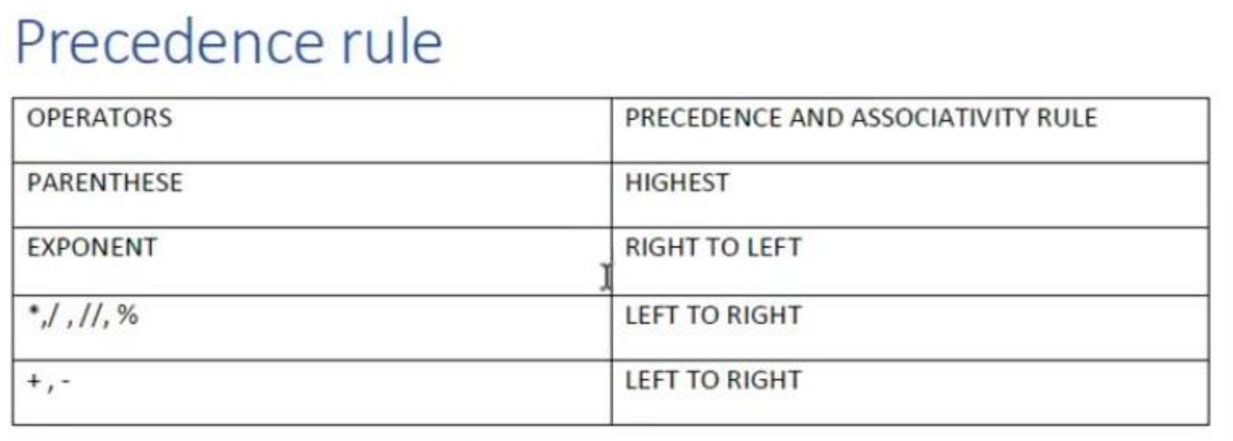
Example

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
# returns "I want to pay 49.95 dollars for 3 pieces of item 567."
```

Example

```
quantity = 3
itemno = 567
price = 49.95
print(f"I want to pay {price} dollars for {quantity} pieces of item {itemno}.")
# returns "I want to pay 49.95 dollars for 3 pieces of item 567."
```

Note: All string methods returns new values. They do not change the original string, because string are immutable in python.



Precedence rule

OPERATORS	PRECEDENCE AND ASSOCIATIVITY RULE
PARENTHESE	HIGHEST
EXPONENT	RIGHT TO LEFT
*,/ ,//, %	LEFT TO RIGHT
+ , -	LEFT TO RIGHT

Python Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example	Try it
+	Addition	x + y	
-	Subtraction	x - y	
*	Multiplication	x * y	
/	Division	x / y	
%	Modulus	x % y	
**	Exponentiation	x ** y	
//	Floor division	x // y	

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As	Try it
=	x = 5	x = 5	
+=	x += 3	x = x + 3	
-=	x -= 3	x = x - 3	
*=	x *= 3	x = x * 3	
/=	x /= 3	x = x / 3	
%=	x %= 3	x = x % 3	
//=	x //= 3	x = x // 3	
**=	x **= 3	x = x ** 3	
&=	x &= 3	x = x & 3	
|=	x |= 3	x = x | 3	
^=	x ^= 3	x = x ^ 3	
>>=	x >>= 3	x = x >> 3	
<<=	x <<= 3	x = x << 3	

Python Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example	Try it
==	Equal	x == y	
!=	Not equal	x != y	
>	Greater than	x > y	
<	Less than	x < y	
>=	Greater than or equal to	x >= y	
<=	Less than or equal to	x <= y	

Python Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example	Try it
and	Returns True if both statements are true	x < 5 and x < 10	
or	Returns True if one of the statements is true	x < 5 or x < 4	
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)	

Python Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example	Try it
is	Returns true if both variables are the same object	x is y	
is not	Returns true if both variables are not the same object	x is not y	

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example	Try it
in	Returns True if a sequence with the specified value is present in the object	x in y	
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	

Python Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Sequence Data Types

Python Collections (Arrays)
There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Python List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Example Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist) # returns ['apple', 'banana', 'cherry']
```

Access Items

You access the list items by referring to the index number:

Example Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]
print(thislist[1])  # returns banana
```

Change Item Value

To change the value of a specific item, refer to the index number:

Example Change the second item:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "grapes"
print(thislist) # returns ['apple', 'grapes', 'cherry']
```

Loop through a List

You can loop through the list items by using a `for` loop:

Example Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Check if Item Exists

To determine if a specified item is present in a list use the `in` keyword:

Example Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

List Length

To determine how many items a list has, use the `len()` method:

Example Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))  # returns 3
```

Add Items

To add an item to the end of the list, use the `append()` method:

Example Using the append() method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)  # returns ['apple', 'banana', 'cherry', 'orange']
```

To add an item at the specified index, use the `insert()` method:

Example Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)  # returns ['apple', 'orange', 'banana', 'cherry']
```

Remove Item

There are several methods to remove items from a list:

Example The remove() method removes the specified item:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist) # returns ['apple', 'cherry']
```

Example The pop() method removes the specified index, (or the last item if index is not specified):

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)  # returns ['apple', 'banana']
```

Example The del keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)  # returns ['banana', 'cherry']
```

Example The del keyword can also delete the list completely:

```
thislist = ["apple", "banana", "cherry"]
del thislist
```

Example The clear() method empties the list:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)  # returns []
```

Copy a List

You cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

There are ways to make a copy, one way is to use the built-in List method `copy()`.

Example Make a copy of a list with the copy() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)  # returns ['apple', 'banana', 'cherry']
```

Another way to make a copy is to use the built-in method `list()`.

Example Make a copy of a list with the list() method:

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)  # returns ['apple', 'banana', 'cherry']
```

The list() Constructor

It is also possible to use the `list()` constructor to make a new list.

Example Using the list() constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(thislist)  # returns ['apple', 'banana', 'cherry']
```

List Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position

remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Python Tuple

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

Example Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)  # returns ('apple', 'banana', 'cherry')
```

Access Tuple Items

You can access tuple items by referring to the index number, inside square brackets:

Example Return the item in position 1:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])  # returns banana
```

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**.

Loop through a Tuple

You can loop through the tuple items by using a `for` loop.

Example Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Tuple Length

To determine how many items a tuple has, use the `len()` method:

Example Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))  # returns 3
```

Add Items

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

Example You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

Example The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
```

The tuple() Constructor

It is also possible to use the `tuple()` constructor to make a tuple.

Example Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-
brackets
print(thistuple)
```

Python Sets

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

Example Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)  # returns {'cherry', 'banana', 'apple'}
```

Note: Sets are unordered, so you cannot be sure in which order the items will appear.

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Example Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Add Items

To add one item to a set use the `add()` method.
To add more than one item to a set use the `update()` method.

Example Add an item to a set, using the add() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)  # returns {'apple', 'cherry', 'orange', 'banana'}
```

Example Add multiple items to a set, using the update() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)  # returns {'apple', 'grapes', 'orange', 'banana', 'mango', 'cherry'}
```

Get the Length of a Set

To determine how many items a set has, use the `len()` method.

Example Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))  # returns 3
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example Remove "banana" by using the remove() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")
```

```
print(thisset)   # returns {'cherry', 'apple'}
```

Example Remove "banana" by using the discard() method:

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset) # returns {'cherry', 'apple'}
```

You can also use the `pop()`, method to remove an item, but this method will remove the *last* item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the `pop()` method is the removed item.

Example Remove the last item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)        # returns apple

print(thisset)  # returns {'cherry', 'banana'}
```

Note: Sets are *unordered*, so when using the `pop()` method, you will not know which item that gets removed.

Example The clear() method empties the set:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)   # returns set()
```

Example The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset
```

The set() Constructor

It is also possible to use the `set()` constructor to make a set.

Example Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

Python Dictionaries

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Example Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)   # returns {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example Get the value of the "model" key:

```
x = thisdict["model"]
```

There is also a method called `get()` that will give you the same result:

Example Get the value of the "model" key:

```
x = thisdict.get("model")
```

Change Values

You can change the value of a specific item by referring to its key name:

Example Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"]=2019 # returns {'brand': 'Ford', 'model': 'Mustang', 'year':
2019}
```

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.
When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

Example Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

Example You can also use the values() function to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

Example Loop through both *keys* and *values*, by using the items() function:

```
for x, y in thisdict.items():
  print(x, y)
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Dictionary Length

To determine how many items (key-value pairs) a dictionary has, use the `len()` method.

Example Print the number of items in the dictionary:

```
print(len(thisdict))
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Removing Items

There are several methods to remove items from a dictionary:

Example The pop() method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

Example The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example The del keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

Example The del keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
```

Example The clear() keyword empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example Make a copy of a dictionary with the copy() method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

Another way to make a copy is to use the built-in method `dict()`.

Example Make a copy of a dictionary with the dict() method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

The dict() Constructor

It is also possible to use the `dict()` constructor to make a new dictionary:

Example

```
thisdict = dict(brand="Ford", model="Mustang", year=1964)
# note that keywords are not string literals
# note the use of equals rather than colon for the assignment
print(thisdict)
```

Python If ... Else

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: `a == b`
- Not Equals: `a != b`
- Less than: `a < b`
- Less than or equal to: `a <= b`
- Greater than: `a > b`
- Greater than or equal to: `a >= b`

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the `if` keyword.

Example If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, `a` and `b`, which are used as part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation, using whitespace, to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The `elif` keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example `a` is equal to `b`, so the first condition is not true, but the `elif` condition is true, so we print to screen that "a and b are equal".

Else

The `else` keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also have an `else` without the `elif`:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
```

```
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example One line if else statement:

```
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Example One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example Test if a is greater than b, AND if c is greater than a:

```
if a > b and c > a:
  print("Both conditions are True")
```

Or

The or keyword is a logical operator, and is used to combine conditional statements:

Example Test if a is greater than b, OR if a is greater than c:

```
if a > b or a > c:
  print("At least one of the conditions is True")
```

Python While Loops

Python has two primitive loop commands:

- while loops
- for loops

The while Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

Example Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

The break Statement

With the `break` statement we can stop the loop even if the while condition is true:

Example Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement

With the `continue` statement we can stop the current iteration, and continue with the next:

Example Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Python for Loops

A `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

The for loop does not require an indexing variable to set beforehand.

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Example Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

Example Exit the loop when x is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    break
  print(x)
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the `range()` function,

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example Using the range() function:

```
for x in range(6):
  print(x)
```

Note that `range(6)` is not the values of 0 to 6, but the values 0 to 5.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6):

Example Using the start parameter:

```
for x in range(2, 6):
  print(x)
```

The `range()` function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: `range(2, 30, 3)`:

Example Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
  print(x)
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Example Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

A nested loop is a loop inside a loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

Example Print each adjective for every fruit:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:
  for y in fruits:
    print(x, y)
```

Python Keywords

Python has a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers:

Method	Description
and	A logical operator
as	To create an alias
assert	For debugging
break	To break out of a loop
class	To define a class
continue	To continue to the next iteration of a loop
def	To define a function
del	To delete an object
elif	Used in conditional statements, same as else if
else	Used in conditional statements
except	Used with exceptions, what to do when an exception occurs
False	Boolean value, result of comparison operations
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
for	To create a for loop
from	To import specific parts of a module
global	To declare a global variable
if	To make a conditional statement
import	To import a module
in	To check if a value is present in a list, tuple, etc.
is	To test if two variables are equal
lambda	To create an anonymous function
None	Represents a null value
nonlocal	To declare a non-local variable
not	A logical operator
or	A logical operator
pass	A null statement, a statement that will do nothing
raise	To raise an exception
return	To exit a function and return a value
True	Boolean value, result of comparison operations
try	To make a try...except statement
while	To create a while loop
with	Used to simplify exception handling
yield	To end a function, returns a generator

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the `def` keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")

my_function()
```

Parameters

Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Passing a List as a Parameter

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

Example

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

Return Values

To let a function return a value, use the `return` statement:

Example

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Recursion

Python also accepts function recursion, which means a defined function can call itself.
Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.
The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

lambda *arguments* : *expression*

The expression is executed and the result is returned:

Example A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Python Classes and Objects

Python is an object oriented programming language.

Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class`:

Example Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

Create Object

Now we can use the class named myClass to create objects:

Example Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example Create a class named Person, use the __init__() function to assign values for name and age:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Object Methods

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

Let us create a method in the Person class:

Example

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

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

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

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

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The `self` parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named `self` , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example Use the words *mysillyobject* and *abc* instead of *self*:

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

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

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

Modify Object Properties

You can modify properties on objects like this:

Example Set the age of p1 to 40:

```
p1.age = 40
```

Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Example Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example Delete the p1 object:

```
del p1
```

Python Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Example Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname
method:

x = Person("John", "Doe")
x.printname()
```

Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Example Create a class named Student, which will inherit the properties and methods from the Person class:

```
class Student(Person):
  pass
```

Note: Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Example Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Add the __init__() Function

So far we have created a child class that inherits the properties and methods from it's parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example Add the __init__() function to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Python also have a `super()` function that will make the child class inherit all the methods and properties from it's parent:

Example

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from it's parent.

Add Properties

Example Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year `2019` should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the __init__() function:

Example Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Example Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Python Iterators

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `__iter__()` and `__next__()`.

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

Example Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

Even strings are iterable objects, and can return an iterator:

Example Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a `for` loop to iterate through an iterable object:

Example Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

Example Iterate the characters of a string:

```
mystr = "banana"

for x in mystr:
  print(x)
```

The `for` loop actually creates an iterator object and executes the next() method for each loop.

Create an Iterator

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

Example Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Stop Iteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the StopIteration statement.

In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example Stop after 20 iterations:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

Python Scope

A variable is only available from inside the region it is created. This is called **scope**.

Local Scope

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

Example A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

Function Inside Function

As explained in the example above, the variable `x` is not available outside the function, but it is available for any function inside the function:

Example The local variable can be accessed from a function within the function:

```
def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
```

Global Scope

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example A variable created outside of a function is global and can be used by anyone:

```
x = 300

def myfunc():
  print(x)

myfunc()

print(x)
```

Naming Variables

If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function):

Example The function will print the local x, and then the code will print the global x:

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.

The `global` keyword makes the variable global.

Example If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Also, use the `global` keyword if you want to make a change to a global variable inside a function.

Example To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

Python Modules

What is a Module?

Consider a module to be the same as a code library.

A file containing a set of functions you want to include in your application.

Create a Module

To create a module just save the code you want in a file with the file extension `.py`:

Example Save this code in a file named mymodule.py

```
def greeting(name):
  print("Hello, " + name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Example Import the module named mymodule, and call the greeting function:

```
import mymodule

mymodule.greeting("Jonathan")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Naming a Module

You can name the module file whatever you like, but it must have the file extension `.py`

Re-naming a Module

You can create an alias when you import a module, by using the `as` keyword:

Example Create an alias for mymodule called mx:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example Import and use the platform module:

```
import platform

x = platform.system()
print(x)
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

Example List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Note: The dir() function can be used on *all* modules, also the ones you create yourself.

Import from Module

You can choose to import only parts from a module, by using the `from` keyword.

Example The module named mymodule has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

Example Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Reloading modules

reload() reloads a previously imported module. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object.

Note: The argument should be a module which has been successfully imported.

Usage:

For Python2.x
    reload(module)

For above 2.x and <=Python3.3
    import imp
    imp.reload(module)

For >=Python3.4
    import importlib
    importlib.reload(module)

Scope of Object and Names

- The scope refers to the region in the program code where variables and names are accessible.
- It also determines the visibility and lifetime of a variable in the program code.
- Assigning a value, defining a function or class (using def or class) or importing module operation creates a name and its place in the code defines its scope.
- The names or objects which are accessible are called in-scope.
- The names or objects which are not accessible are called out-of-scope.
- The Python scope concept follows the LEGB (Local, Enclosing, Global and built-in) rule.
- The scope concept helps to avoid the name collision and use of global names across the programs.

Names and Objects in Python

- Python is a dynamically types language, so when we assign the value to the variable for the first time, it creates the variable. Others names or objects come in existence as given:

Operation	Statement
Assignment	X=value
Import Operation	import module
Function definitions	def fun( )
Arguments in the function	def fun(x1, x2 , x3)
Class definition	class abc :

- Python uses the location of the name assignment or definition to associate it with a scope.
- If a variable assigned value in the function, it has a local scope.
- If a variable assigned value at the top level and outside all the functions, it has a global scope.

Python Scope Vs Namespace

- The scope of a variables and objects are related to the concept of the namespace.
- The scope determines the visibility of the names and objects in the program.
- A namespace is a collection of names.
- In python, scopes are implemented as dictionaries that maps names to objects. These dictionaries are called namespaces.
- The dictionaries have the names as key and the objects as value.
- A strings, lists, functions, etc everything in Python is an object.

Searching a name in the Namespace

- Whenever we use a name, such as a variable or a function name, Python searches through different scope levels (or namespaces) to determine whether the name exists or not.
- If the name exists, then we always get the first occurrence of it.
- Otherwise, we get an error.

Types of namespace

- **Built-in Namespace** It includes functions and exception names that are built-in in the python.
- **Global Namespace** It includes the names from the modules that are imported in the code. It lasts till the end of program.
- **Local Namespace** It includes the names defined in the function. It is created when the function is called and ends when the value returned.

LEGB Rule for Python Scope

- The LEGB stands for Local, Enclosing, Global and Built-in.
- Python resolves names using the LEGB rules.
- The LEGB rule is a kind of name lookup procedure, which determines the order in which Python looks up names.
- For example, if we access a name, then Python will look that name up sequentially in the local, enclosing, global, and built-in scope.

Local (or function) scope

- It is the code block or body of any Python function or lambda expression.
- This Python scope contains the names that you define inside the function.
- These names will only be visible from the code of the function.
- It’s created at function call, not at function definition, so we have as many different local scopes as function calls.
- It is applicable if we call the same function multiple times, or recursively.
- Each call will result in a new local scope being created.

Enclosing (or nonlocal) scope

- It is a special scope that only exists for nested functions.
- If the local scope is an inner or nested function, then the enclosing scope is the scope of the outer or enclosing function.
- This scope contains the names that you define in the enclosing function.

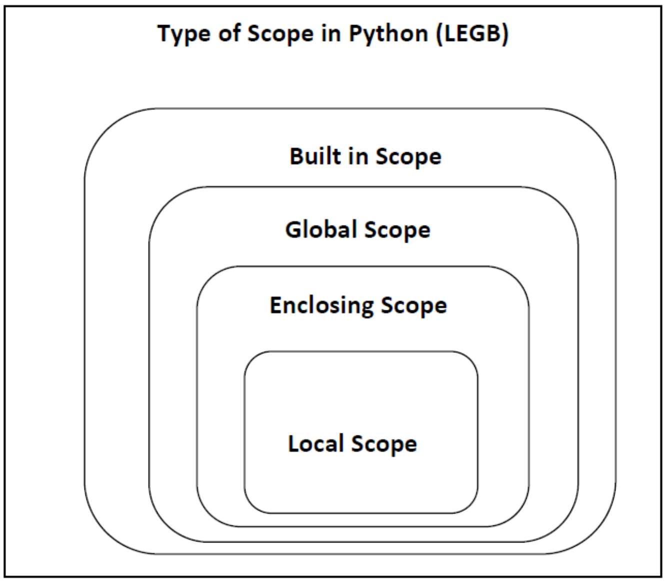
- The names in the enclosing scope are visible from the code of the inner and enclosing functions.

Global (or module) scope

- It is the top-most scope in a Python program, script, or module.
- This scope contains all of the names that are defined at the top level of a program or a module.
- Names in this Python scope are visible from everywhere in your code.

Built-in scope

- It is a special scope which is created or loaded at script run or opens an interactive session.
- This scope contains names such as keywords, functions, exceptions, and other attributes that are built into Python.
- Names in this scope are also available from everywhere in your code.



Example

```
var1 = 5 #var1 is in the global namespace
    def ABC( ):
        var2 = 6 # var2 is in the local namespace
        def XYZ( ):
             var3 = 7 # var3 is in the nested local namespace
```

- In this example, the **var1** is declared in the global namespace because it is not enclosed inside any function. So it is accessible everywhere in the script.
- **var2** is inside the **ABC().** So, it can be accessed only inside the **ABC()** and outside the function, it no longer exists.
- **var3** also has a local scope, the function is nested and we can use **var3** only inside **XYZ().**

Python Date time

Python Dates

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

Example Import the datetime module and display the current date:

```
import datetime

x = datetime.datetime.now()
print(x)
```

Date Output

When we execute the code from the example above the result will be:

```
2019-09-24 15:35:35.403417
```

The date contains year, month, day, hour, minute, second, and microsecond.

The `datetime` module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example Return the year and name of weekday:

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

Example Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

The `datetime()` class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of `0`, (`None` for timezone).

The strftime() Method

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

Example Display the name of the month:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec
%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST

%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18
%X	Local version of time	17:41:00
%%	A % character	%

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `else` block lets you execute code when there is no error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example This statement will raise an error, because x is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

Example In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
```

```
except:
  print("Something went wrong when opening the file")
```

The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the `raise` keyword.

Example Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

File Processing (Handling)

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with Reading and Writing files.

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist "a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

"r+" To read and write data into the file. The previous data in the file will not be deleted

"w+" To write and read data. It will override existing data

"a+" To append and read data from the file. It won't override existing data

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Python File Open

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the `readline()` method:

Example Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

By calling `readline()` two times, you can read the two first lines:

Example Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

Example Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Example

Return all lines in the file, as a list where each line is an item in the list object:

```
f = open("demofile.txt", "r")
print(f.readlines())
```

Close Files

It is a good practice to always close the file when you are done with it.

Example Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Python File Write

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

`"a"` - Append - will append to the end of the file

`"w"` - Write - will overwrite any existing content

Example Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

`"x"` - Create - will create a file, returns an error if the file exist

`"a"` - Append - will create a file if the specified file does not exist

`"w"` - Write - will create a file if the specified file does not exist

Example Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Python Delete File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

With statement

with statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Unlike the above implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources.

Syntax: with open filename as file:

```
# Program to show various ways to
# read data from a file.

L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

# Creating a file
with open("myfile.txt", "w") as file1:
        # Writing data to a file
        file1.write("Hello \n")
        file1.writelines(L)
        file1.close() # to change file access modes
```

```
with open("myfile.txt", "r+") as file1:
        # Reading form a file
        print(file1.read())
```

Output:

```
Hello
This is Delhi
This is Paris
This is London
```

Python File tell() Method

Example Find the current file position:

```
f = open("demofile.txt", "r")
print(f.tell())
```

Definition and Usage

The `tell()` method returns the current file position in a file stream.

Tip: You can change the current file position with the seek method.

Python File seek() Method

Example Change the current file position to 4, and return the rest of the line:

```
f = open("demofile.txt", "r")
f.seek(4)
print(f.readline())
```

Definition and Usage

The `seek()` method sets the current file position in a file stream.

The `seek()` method also returns the new postion.

Command Line Arguments

The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:

- Using sys.argv
- Using getopt module
- Using argparse module

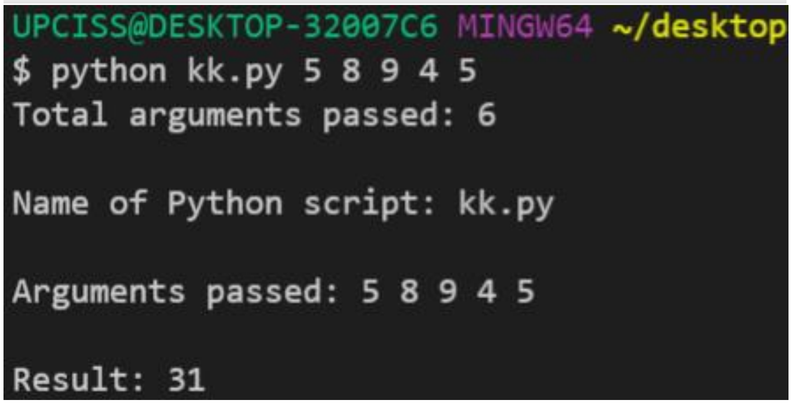
Using sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
One such variable is sys.argv which is a simple list structure. It's main purpose are:

- It is a list of command line arguments.
- len(sys.argv) provides the number of command line arguments.
- sys.argv[0] is the name of the current Python script.

Example Let's suppose there is a Python script for adding two numbers and the numbers are passed as command-line arguments.

```python
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("\nName of Python script:", sys.argv[0])

print("\nArguments passed:", end = " ")
for i in range(1, n):
    print(sys.argv[i], end = " ")

# Addition of numbers
Sum = 0
# Using argparse module
for i in range(1, n):
    Sum += int(sys.argv[i])

print("\n\nResult:", Sum)
```



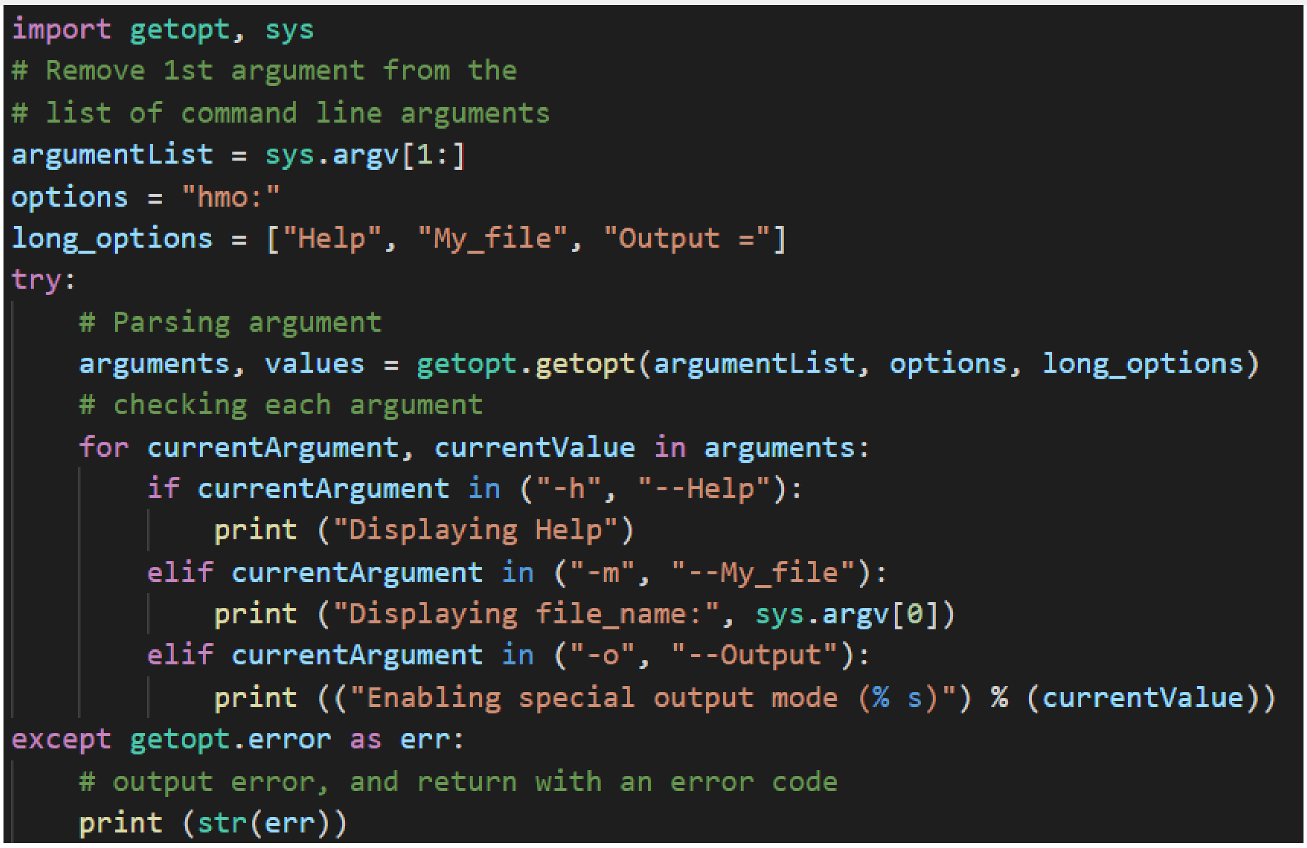
```
UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py 5 8 9 4 5
Total arguments passed: 6

Name of Python script: kk.py

Arguments passed: 5 8 9 4 5

Result: 31
```

Using getopt module

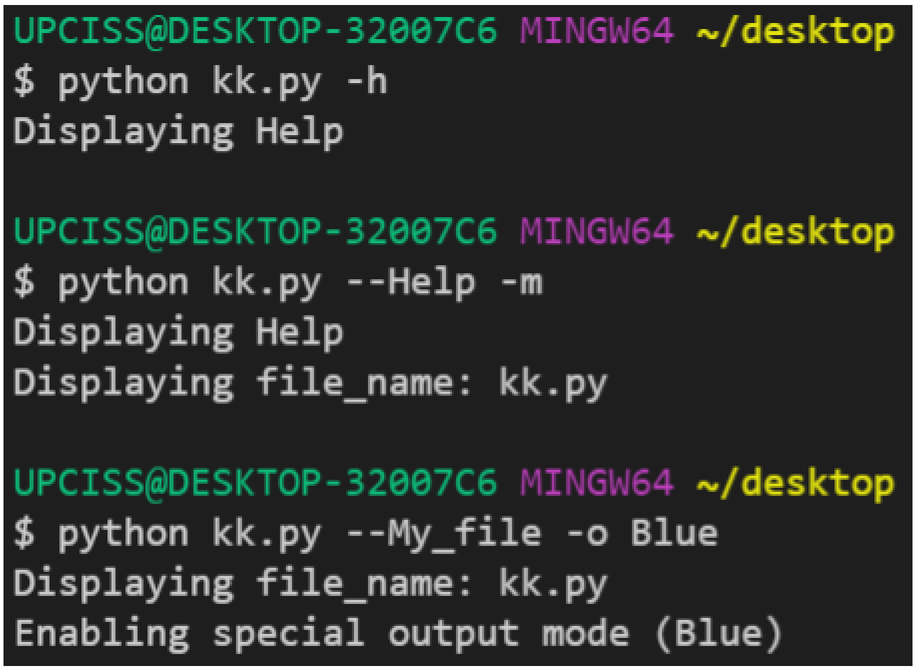
Python **getopt module** is similar to the getopt() function of C. Unlike sys module getopt module extends the separation of the input string by parameter validation. It allows both short, and long options including a value assignment. However, this module requires the use of the sys module to process input data properly. To use getopt module, it is required to remove the first element from the list of command-line arguments.

Syntax: getopt.getopt(args, options, [long_options])
Parameters:
args: List of arguments to be passed.
options: String of option letters that the script want to recognize. Options that require an argument should be followed by a colon (:).
long_options: List of string with the name of long options. Options that require arguments should be followed by an equal sign (=).
Return Type: Returns value consisting of two elements: the first is a list of (option, value) pairs. The second is the list of program arguments left after the option list was stripped.

Example



```
import getopt, sys
# Remove 1st argument from the
# list of command line arguments
argumentList = sys.argv[1:]
options = "hmo:"
long_options = ["Help", "My_file", "Output ="]
try:
    # Parsing argument
    arguments, values = getopt.getopt(argumentList, options, long_options)
    # checking each argument
    for currentArgument, currentValue in arguments:
        if currentArgument in ("-h", "--Help"):
            print ("Displaying Help")
        elif currentArgument in ("-m", "--My_file"):
            print ("Displaying file_name:", sys.argv[0])
        elif currentArgument in ("-o", "--Output"):
            print (("Enabling special output mode (% s)") % (currentValue))
except getopt.error as err:
    # output error, and return with an error code
    print (str(err))
```

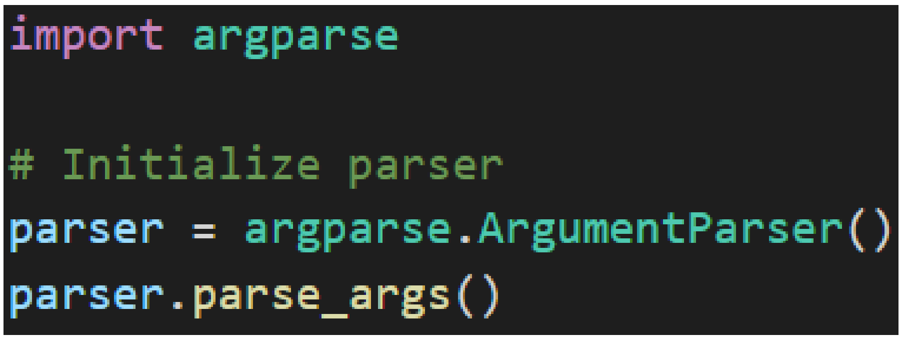


```
UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py -h
Displaying Help

UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py --Help -m
Displaying Help
Displaying file_name: kk.py

UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py --My_file -o Blue
Displaying file_name: kk.py
Enabling special output mode (Blue)
```

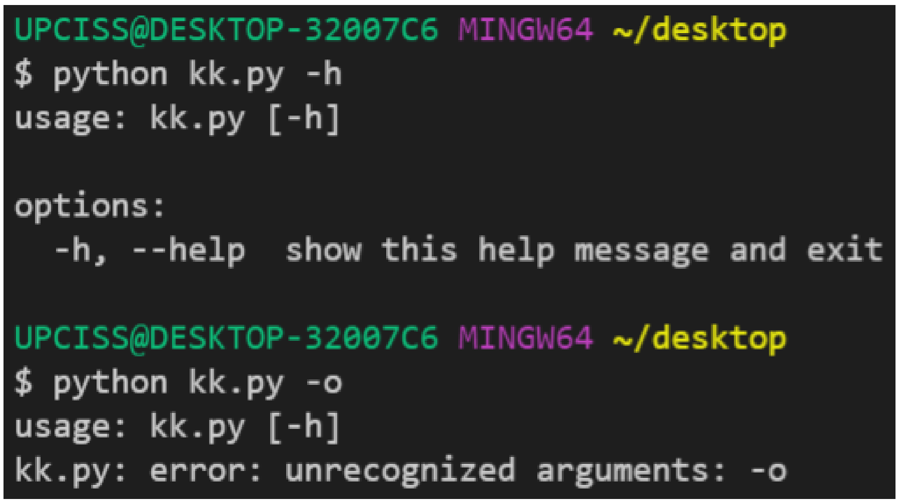
Using argparse module
Using argparse module is a better option than the above two options as it provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc.

Note: As a default optional argument, it includes -h, along with its long version –help.

Example 1: Basic use of argparse module.

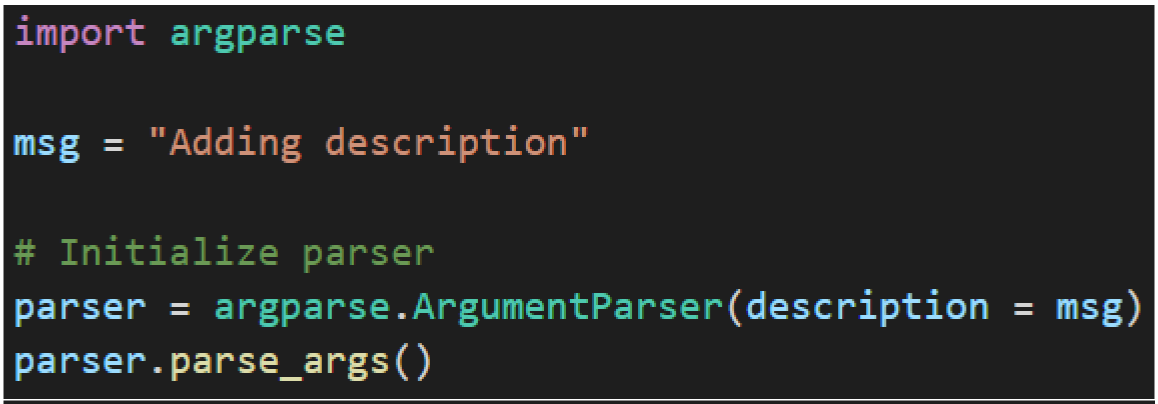


```python
import argparse

# Initialize parser
parser = argparse.ArgumentParser()
parser.parse_args()
```

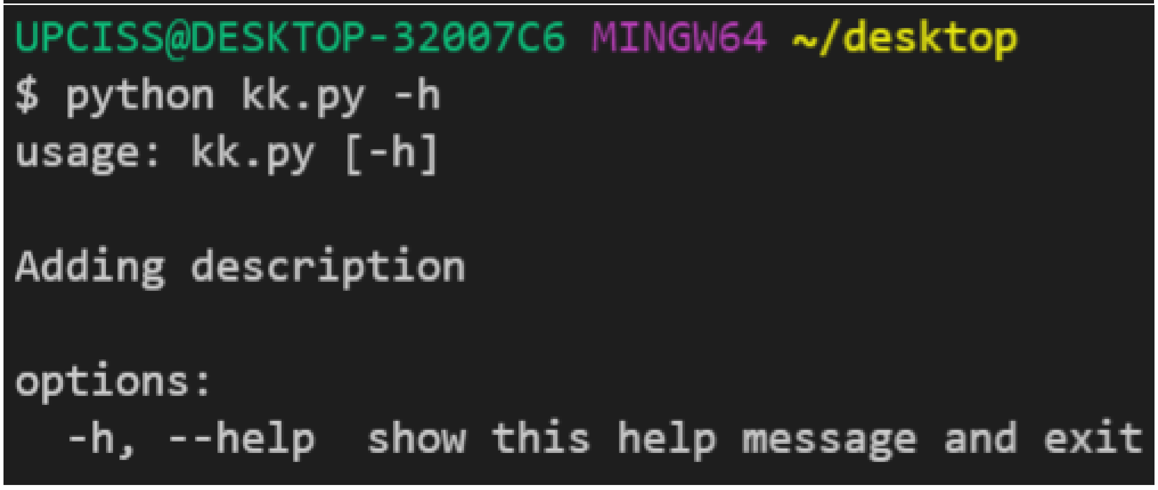


```
UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py -h
usage: kk.py [-h]

options:
  -h, --help  show this help message and exit

UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py -o
usage: kk.py [-h]
kk.py: error: unrecognized arguments: -o
```

Example 2: Adding description to the help message.



```
import argparse

msg = "Adding description"

# Initialize parser
parser = argparse.ArgumentParser(description = msg)
parser.parse_args()
```



```
UPCISS@DESKTOP-32007C6 MINGW64 ~/desktop
$ python kk.py -h
usage: kk.py [-h]

Adding description

options:
  -h, --help  show this help message and exit
```

NumPy Basics

What is NumPy?
NumPy is a Python library used for working with arrays.
It also has functions for working in domain of linear algebra, fourier transform, and matrices.
NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
NumPy stands for Numerical Python.

Why Use NumPy?
In Python we have lists that serve the purpose of arrays, but they are slow to process.
NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.
Arrays are very frequently used in data science, where speed and resources are very important.
Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
This behavior is called locality of reference in computer science.
This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

Import NumPy

Once NumPy is installed, import it in your applications by adding the `import` keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Example

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

NumPy as np

NumPy is usually imported under the `np` alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the `as` keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as `np` instead of `numpy`.

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Checking NumPy Version

The version string is stored under `__version__` attribute.

Example

```
import numpy as np
print(np.__version__)
```

NumPy Creating Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`.
We can create a NumPy `ndarray` object by using the `array()` function.

Example

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that `arr` is `numpy.ndarray` type.

To create an `ndarray`, we can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an `ndarray`:

Example Use a tuple to create a NumPy array:

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

Nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example Create a 0-D array with value 42

```
import numpy as np
arr = np.array(42)
print(arr)
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
These are the most common and basic arrays.

Example Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.

NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

Example Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

3-D arrays
An array that has 2-D arrays (matrices) as its elements is called 3-D array.
These are often used to represent a 3rd order tensor.

Example Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Check Number of Dimensions?
NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

Example Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Higher Dimensional Arrays
An array can have any number of dimensions.
When the array is created, you can define the number of dimensions by using the `ndmin` argument.

Example Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

NumPy Array Indexing

Access Array Elements
Array indexing is the same as accessing an array element.
You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example Get the first element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Example Get the second element from the following array.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
```

Example Get third and fourth elements from the following array and add them.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

Example Access the element on the first row, second column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

Example Access the element on the 2nd row, 5th column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd row: ', arr[1, 4])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Example Explained

`arr[0, 1, 2]` prints the value `6`.

And this is why:

The first number represents the first dimension, which contains two arrays:

[[1, 2, 3], [4, 5, 6]]

and:

[[7, 8, 9], [10, 11, 12]]

Since we selected `0`, we are left with the first array:

[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays:
[1, 2, 3]
and:
[4, 5, 6]
Since we selected 1, we are left with the second array:
[4, 5, 6]
The third number represents the third dimension, which contains three values:
4
5
6
Since we selected 2, we end up with the third value:
6

Negative Indexing
Use negative indexing to access an array from the end.

Example Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Array Slicing

Slicing arrays
Slicing in python means taking elements from one given index to another given index.
We pass slice instead of index like this: `[start:end]`.
We can also define the step, like this: `[start:end:step]`.
If we don't pass start its considered 0
If we don't pass end its considered length of array in that dimension
If we don't pass step its considered 1

Example Slice elements from index 1 to index 5 from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Note: The result *includes* the start index, but *excludes* the end index.

Example Slice elements from index 4 to the end of the array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Example Slice elements from the beginning to index 4 (not included):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
```

Negative Slicing

Use the minus operator to refer to an index from the end:

Example Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

STEP

Use the `step` value to determine the step of the slicing:

Example Return every other element from index 1 to index 5:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

Example Return every other element from the entire array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[::2])
```

Slicing 2-D Arrays

Example From the second element, slice elements from index 1 to index 4 (not included):

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

Note: Remember that *second element* has index 1.

Example From both elements, return index 2:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 2])
```

Example From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

NumPy Data Types

Data Types in Python

By default Python have these data types:

- `strings` - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- `integer` - used to represent integer numbers. e.g. -1, -2, -3
- `float` - used to represent real numbers. e.g. 1.2, 42.42
- `boolean` - used to represent True or False.
- `complex` - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like `i` for integers, `u` for unsigned integers etc.
Below is a list of all data types in NumPy and the characters used to represent them.

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called `dtype` that returns the data type of the array:

Example Get the data type of an array object:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

Example Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

Creating Arrays with a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

Example Create an array with data type string:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

For `i`, `u`, `f`, `S` and `U` we can define size as well.

Example Create an array with data type 4 bytes integer:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='i4')
print(arr)
print(arr.dtype)
```

What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

Example A non-integer string like 'a' cannot be converted to integer (will raise an error):

```
import numpy as np
arr = np.array(['a', '2', '3'], dtype='i')
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.
The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.
The data type can be specified using a string, like `'f'` for float, `'i'` for integer etc. or you can use the data type directly like `float` for float and `int` for integer.

Example Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

Example Change data type from float to integer by using int as parameter value:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

Example Change data type from integer to boolean:

```
import numpy as np
arr = np.array([1, 0, 3])
newarr = arr.astype(bool)
print(newarr)
print(newarr.dtype)
```

NumPy Array Copy vs View

The Difference between Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.
The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

COPY:

Example Make a copy, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

The copy SHOULD NOT be affected by the changes made to the original array.

VIEW:

Example Make a view, change the original array, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

The view SHOULD be affected by the changes made to the original array.

Make Changes in the VIEW:

Example Make a view, change the view, and display both arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31
print(arr)
print(x)
```

The original array SHOULD be affected by the changes made to the view.

Check if Array Owns its Data

As mentioned above, copies *owns* the data, and views *does not own* the data, but how can we check this?

Every NumPy array has the attribute `base` that returns `None` if the array owns the data.

Otherwise, the `base` attribute refers to the original object.

Example Print the value of the base attribute to check if an array owns it's data or not:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```

The copy returns `none`.

The view returns the original array.

NumPy Array Shape

Shape of an Array

The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.

Example Print the shape of a 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

The example above returns `(2, 4)`, which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Example Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('shape of array :', arr.shape)
```

What does the shape tuple represent?

Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

NumPy Array Reshaping

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Example Convert the following 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Reshape From 1-D to 3-D

Example Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.
We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Example Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

Returns Copy or View?

Example Check if the returned array is a copy or a view:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(arr.reshape(2, 4).base)
```

The example above returns the original array, so it is a view.

Unknown Dimension

You are allowed to have one "unknown" dimension.
Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.
Pass `-1` as the value, and NumPy will calculate this number for you.

Example Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

Note: We can not pass `-1` to more than one dimension.

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.
We can use `reshape(-1)` to do this.

Example Convert the array into a 1D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)
print(newarr)
```

Note: There are a lot of functions for changing the shapes of arrays in numpy `flatten`, `ravel` and also for rearranging the elements `rot90`, `flip`, `fliplr`, `flipud` etc. These fall under Intermediate to Advanced section of numpy.

NumPy Array Iterating

Iterating means going through elements one by one.
As we deal with multi-dimensional arrays in numpy, we can do this using basic `for` loop of python.
If we iterate on a 1-D array it will go through each element one by one.

Example Iterate on the elements of the following 1-D array:

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

Iterating 2-D Arrays

In a 2-D array it will go through all the rows.

Example Iterate on the elements of the following 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  print(x)
```

If we iterate on a *n*-D array it will go through n-1th dimension one by one.
To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Example Iterate on each scalar element of the 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  for y in x:
    print(y)
```

Iterating 3-D Arrays

In a 3-D array it will go through all the 2-D arrays.

Example Iterate on the elements of the following 3-D array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  print(x)
```

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Example Iterate down to the scalars:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  for y in x:
    for z in y:
      print(z)
```

Iterating Arrays Using nditer()

The function nditer() is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic for loops, iterating through each scalar of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality.

Example Iterate through the following 3-D array:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
  print(x)
```

Iterating Array With Different Data Types

We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating.

NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer() we pass flags=['buffered'].

Example Iterate through the array as a string:

```
import numpy as np
arr = np.array([1, 2, 3])
for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
  print(x)
```

Iterating With Different Step Size

We can use filtering and followed by iteration.

Example Iterate through every scalar element of the 2D array skipping 1 element:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in np.nditer(arr[:, ::2]):
  print(x)
```

Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those usecases.

Example Enumerate on following 1D arrays elements:

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Example Enumerate on following 2D array's elements:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

NumPy Joining Array

Joining means putting contents of two or more arrays in a single array.
In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.
We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example Join two arrays

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

Example Join two 2-D arrays along rows (axis=1):

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.
We pass a sequence of arrays that we want to join to the `stack()` method along with the axis. If axis is not explicitly passed it is taken as 0.

Example

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

Stacking Along Rows

NumPy provides a helper function: `hstack()` to stack along rows.

Example

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))
print(arr)
```

Stacking Along Columns

NumPy provides a helper function: `vstack()` to stack along columns.

Example

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2))
print(arr)
```

Stacking along Height (depth)

NumPy provides a helper function: `dstack()` to stack along height, which is the same as depth.

Example

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.dstack((arr1, arr2))
print(arr)
```

NumPy Splitting Array

Splitting is reverse operation of Joining.
Joining merges multiple arrays into one and Splitting breaks one array into multiple.
We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

Example Split the array in 3 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

Note: The return value is an array containing three arrays.
If the array has less elements than required, it will adjust from the end accordingly.

Example Split the array in 4 parts:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 4)
print(newarr)
```

Note: We also have the method `split()` available but it will not adjust the elements when elements are less in source array for splitting like in example above, `array_split()` worked properly but `split()` would fail.

Split into Arrays

The return value of the `array_split()` method is an array containing each of the split as an array.
If you split an array into 3 arrays, you can access them from the result just like any array element:

Example Access the splitted arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays.

Use the `array_split()` method, pass in the array you want to split and the number of splits you want to do.

Example Split the 2-D array into three 2-D arrays.

```
import numpy as np
arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])
newarr = np.array_split(arr, 3)
print(newarr)
```

The example above returns three 2-D arrays.

Let's look at another example, this time each element in the 2-D arrays contains 3 elements.

Example Split the 2-D array into three 2-D arrays.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],
[16, 17, 18]])
newarr = np.array_split(arr, 3)
print(newarr)
```

The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the row (axis=1).

Example Split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],
[16, 17, 18]])
newarr = np.array_split(arr, 3, axis=1)
print(newarr)
```

An alternate solution is using `hsplit()` opposite of `hstack()`

Example Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15],
[16, 17, 18]])
newarr = np.hsplit(arr, 3)
print(newarr)
```

Note: Similar alternates to `vstack()` and `dstack()` are available as `vsplit()` and `dsplit()`.

NumPy Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the `where()` method.

Example Find the indexes where the value is 4:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

The example above will return a tuple: `(array([3, 5, 6],)`

Which means that the value 4 is present at index 3, 5, and 6.

Example Find the indexes where the values are even:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

Example Find the indexes where the values are odd:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 1)
print(x)
```

Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The `searchsorted()` method is assumed to be used on sorted arrays.

Example Find the indexes where the value 7 should be inserted:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

Example explained: The number 7 should be inserted on index 1 to remain the sort order.

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

Search from the Right Side

By default the left most index is returned, but we can give `side='right'` to return the right most index instead.

Example Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

Example explained: The number 7 should be inserted on index 2 to remain the sort order.
The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

Multiple Values

To search for more than one value, use an array with the specified values.

Example Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np
arr = np.array([1, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 6])
print(x)
```

NumPy Sorting Arrays

Sorting means putting elements in an *ordered sequence*.
Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

Example Sort the array:

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

Note: This method returns a copy of the array, leaving the original array unchanged.

You can also sort arrays of strings, or any other data type:

Example Sort the array alphabetically:

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

Example Sort a boolean array:

```
import numpy as np
arr = np.array([True, False, True])
print(np.sort(arr))
```

Sorting a 2-D Array

If you use the sort() method on a 2-D array, both arrays will be sorted:

Example Sort a 2-D array:

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

NumPy Filter Array

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.
In NumPy, you filter an array using a *boolean index list*.

A *boolean index list* is a list of booleans corresponding to indexes in the array.
If the value at an index is `True` that element is contained in the filtered array, if the value at that index is `False` that element is excluded from the filtered array.

Example Create an array from the elements on index 0 and 2:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

The example above will return `[41, 43]`, why?
Because the new filter contains only the values where the filter array had the value `True`, in this case, index 0 and 2.

Creating the Filter Array

In the example above we hard-coded the `True` and `False` values, but the common use is to create a filter array based on conditions.

Example Create a filter array that will return only values higher than 42:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is higher than 42, set the value to True, otherwise False:
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

Example Create a filter array that will return only even elements from the original array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is completely divisble by 2, set the value to True,
otherwise False
  if element % 2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
```

```
newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

Creating Filter Directly From Array
The above example is quite a common task in NumPy and NumPy provides a nice way to tackle it.
We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

Example Create a filter array that will return only values higher than 42:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

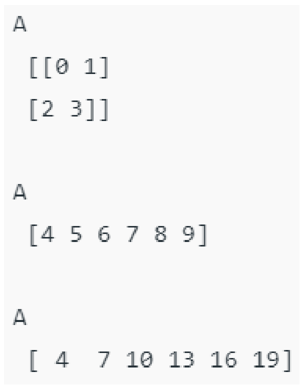
Example Create a filter array that will return only even elements from the original array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
filter_arr = arr % 2 == 0
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Array from Numerical Ranges

numpy.arange() in Python
The **arange([start,] stop[, step,][, dtype]) :** Returns an array with evenly spaced elements as per the interval. The interval mentioned is half-opened i.e. [Start, Stop)
Parameters :
  start : [optional] start of interval range. By default start = 0
  stop : end of interval range
  step : [optional] step size of interval. By default step size = 1,
  For any output out, this is the distance between two adjacent values, out[i+1] - out[i].
  dtype : type of output array
Example:

```
import numpy as np
print("A\n", np.arange(4).reshape(2, 2), "\n")
print("A\n", np.arange(4, 10), "\n")
print("A\n", np.arange(4, 20, 3), "\n")
```



```
A
 [[0 1]
 [2 3]]

A
 [4 5 6 7 8 9]

A
 [ 4  7 10 13 16 19]
```

numpy.eye() in Python

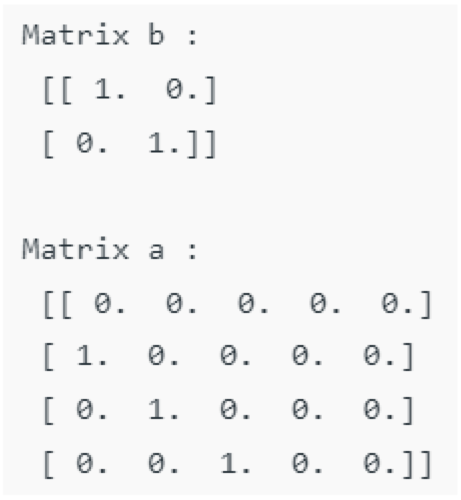
numpy.eye(R, C = None, k = 0, dtype = type <'float'>) : –The eye tool returns a 2-D array with **1's** as the diagonal and **0's** elsewhere. The diagonal can be main, upper, or lower depending on the optional parameter **k**. A positive **k** is for the upper diagonal, a negative **k** is for the lower, and a **0 k** (default) is for the main diagonal.

Parameters :

R : Number of rows
C : [optional] Number of columns; By default M = N
k : [int, optional, 0 by default]
Diagonal we require; k>0 means diagonal above main diagonal or vice versa.
dtype : [optional, float(by Default)] Data type of returned array.

Example:

```
import numpy as np
# 2x2 matrix with 1's on main diagonal
b = np.eye(2, dtype = float)
print("Matrix b : \n", b)

# matrix with R=4 C=5 and 1 on diagonal
# below main diagonal
a = np.eye(4, 5, k = -1)
print("\nMatrix a : \n", a)
```



```
Matrix b :
 [[ 1.  0.]
 [ 0.  1.]]

Matrix a :
 [[ 0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]]
```

Related to Python

eval() Function

Definition and Usage

The `eval()` function evaluates the specified expression, if the expression is a legal Python statement, it will be executed.

```
x='input("Entar a number: ")'
y=eval(eval(x))
print(y)
print(type(y))
```

```
Entar a number: 6
6
<class 'int'>
```

swapcase() Method

Definition and Usage

The `swapcase()` method returns a string where all the upper case letters are lower case and vice versa.

```
x='upciss'
p=x.swapcase()
print(p)
```

```
UPCISS
```

String join() Method

Definition and Usage

The `join()` method takes all items in an iterable and joins them into one string.

A string must be specified as the separator.

```
l=['welcome','to','upciss']
msg=' '.join(l)
print(msg)
```

```
welcome to upciss
```

String partition() Method

Definition and Usage

The `partition()` method searches for a specified string, and splits the string into a tuple containing three elements.

The first element contains the part before the specified string.

The second element contains the specified string.

The third element contains the part after the string.

```
l='welcome to upciss youtube channel'
msg=l.partition('upciss')
print(msg)
```
```
('welcome to ', 'upciss', ' youtube channel')
```

```
islower(), isupper(), istitle(), isspace(), isalpha(), isdigit(), isalnum()

All Function return the Boolean value True/False.
```

```
l='Welcome To Upciss Youtube Channel'
msg=l.islower()
print(msg)
```
```
False
```

```
startswith(), endswith()
l='welcome to upciss'
msg=l.startswith('welcome')
print(msg)
```
```
True
```

```
                                ceil, floor, trunc function
import math
k=45.1
print(math.ceil(k))  #return 46 integer not greater than k

k=45.59
print(math.floor(k))  #return 45 integer greater than or equal to k

k=45.59
print(math.trunc(k))  #return 45 simply remove the decimals
```

```
                                      DocString
def add(a,b):
    """add 2 number and print with f String method"""
    print(f'Total : {a+b}')

add(4,5) # return 9
print(add.__doc__)  # return add 2 number and print with f String method
```

It takes a lot of hard work to make notes, so if you can pay some fee 50, 100, 200 rupees which you think is reasonable, if you are able to Thank you...

नोट्स बनाने में बहुत मेहनत लगी है , इसलिए यदि आप कुछ शुल्क 50,100, 200 रूपए जो आपको उचित लगता है pay कर सकते है, अगर आप सक्षम है तो, धन्यवाद ।

G Pay

Jitendra Kumar

Account Number

17000100008177

IFSC Code

BARB0KAFARA

UPI ID

Jitendraupciss@okicici

Scan QR