PYTHON DOCUMENTATION

Contents:

1.INTRODUCTION

1. INTRODUCTION:

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.

Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0, released 2000, introduced features like list comprehensions and a garbage collection system capable of collecting reference cycles. Python 3.0, released 2008, was a major revision of the language that is not completely backward-compatible, and much Python 2 code does not run unmodified on Python 3. Due to concern about the amount of code written for Python 2, support for Python 2.7 (the last release in the 2.x series) was extended to 2020. Language developer Guido van Rossum shouldered sole responsibility for the project until July 2018 but now shares his leadership as a member of a five-person steering council.

Python interpreters are available for many operating systems. A global community of programmers develops and maintains CPython, an open source reference implementation. A non-profit organization, the Python Software Foundation, manages and directs resources for Python and CPython development.

1.1HISTROY OF PYTHON:

FIG 1.He the man behind the python.

Python was conceived in the late 1980s[33] by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL), capable of exception handling and interfacing with the Amoeba operating system. Its implementation began in December 1989.[35] Van Rossum continued as Python's lead developer until July 12, 2018, when he announced his "permanent vacation" from his responsibilities as Python's *Benevolent Dictator For Life*, a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. In January, 2019, active Python core developers elected Brett Cannon, Nick Coghlan, Barry Warsaw, Carol Willing and Van Rossum to a five-member "Steering Council" to lead the project.

Python 2.0 was released on 16 October 2000 with many major new features, including a cycle-detecting garbage collector and support for Unicode.

Python 3.0 was released on 3 December 2008. It was a major revision of the language that is not completely backward-compatible.[39] Many of its major features were backported to Python 2.6.x and 2.7.x version series. Releases of Python 3 include the 2to3 utility, which automates (at least partially) the translation of Python 2 code to Python 3.

Python 2.7's end-of-life date was initially set at 2015 then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3. In January 2017, Google announced work on a Python 2.7 to Go transcompiler to improve performance under concurrent workloads.

1.2Types Of Python:

There are so many versions in python from last few years the people majorly using both python 2.7.

In latest the python 3.7 version is here to use.There are so many differences between two versions.

Now let’s talk about Python 2.7

Python 2.7 is the last major release in the 2.x series, as the Python maintainers have shifted the focus of their new feature development efforts to the Python 3.x series. This means that while Python 2 continues to receive bug fixes, and to be updated to build correctly on new hardware and versions of supported operated systems, there will be no new full feature releases for the language or standard library.

However, while there is a large common subset between Python 2.7 and Python 3, and many of the changes involved in migrating to that common subset, or directly to Python 3, can be safely automated, some other changes (notably those associated with Unicode handling) may require careful consideration, and preferably robust automated regression test suites, to migrate effectively.

This means that Python 2.7 will remain in place for a long time, providing a stable and supported base platform for production systems that have not yet been ported to Python 3. The full expected lifecycle of the Python 2.7 series is detailed in PEP 373.

This is all about the python 2.7

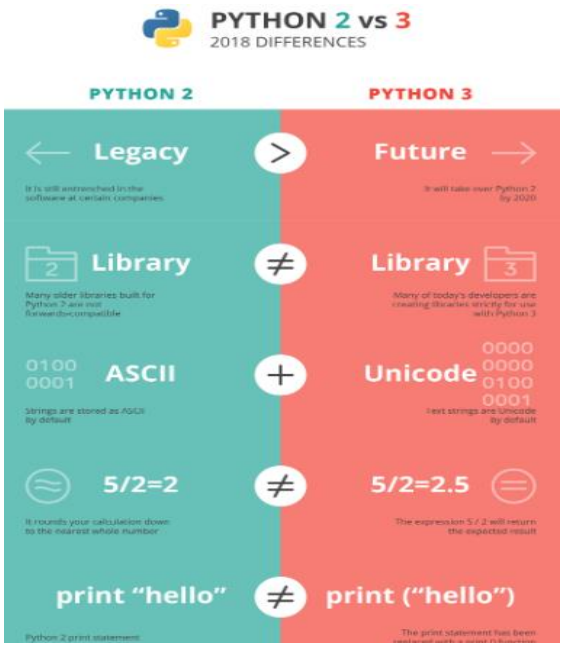
Now let’s talk about the Python 3.7

Now, in 2018, it’s more of a no-brainer: Python 3 is the clear winner for new learners or those wanting to update their skills. Here, we'll cover why Python 3 is better, and why companies have been moving from Python 2 to 3 .

That said, there are still some situations where picking up Python 2 might be advantageous. Or you may just want to learn a little of the history and the differences between Python 2 and 3 for curiosity's sake.

So, without further ado, let’s take a look at some of the major differences between Python 2 vs Python 3–plus where to learn Python 3 programming as a beginner.

1.3 Difference between the two Version 2x & 3x:



2.DATA TYPES

2.1 INTRODUCTION:

Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

Python enables us to check the type of the variable used in the program. Python provides us the type() function which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

2.2 TYPES OF DATA TYPES:

Standard data types:

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. **Numbers**
2. **String**
3. **List**
4. **Tuple**
5. **Dictionary**
6. **Set**

In this section of the tutorial, we will give a brief introduction of the above data types. We will discuss each one of them in detail later in this tutorial.

2.2(b) Strings & Methods:

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

The operator * is known as repetition operator as the operation "Python " *2 returns "Python Python ".

The following example illustrates the string handling in python

```
1.      str1 = 'hello anjan ' #string str1
2.      str2 = ' how are you' #string str2
3.      print (str1[0:2]) #printing first two character using slice operator
4.      print (str1[4]) #printing 4th character of the string
5.      print (str1*2) #printing the string twice
6.      print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello anjan hello anjan
hello anjan how are you
```

Methods of String in Python:

There are total 45 methods in string that we use in python they are:
Python has a set of built-in methods that you can use on strings.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string

index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations

partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case

zfill()	Fills the string with a specified number of 0 values at the beginning

2.2(a) Numbers & Types:

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example

```
1.      a = 3 , b = 5  #a and b are number objects
```

Python supports 4 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
3. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
4. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts respectively).

2.2(c). Lists &Methods:

Lists are similar to arrays in C. However; the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

```
l  = [1, "hi", "python", 2]
print (l[3:]);
print (l[0:2]);
print (l);
print (l + l);
print (l * 3);
```

Output:

```
[2]
[1, 'hi']
[1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]
```

Methods of List in Python:

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

2.2(d) Tuple & Methods:

Tuple:

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
1.  t  = ("hi", "python", 2)
2.  print (t[1:]);
3.  print (t[0:1]);
4.  print (t);
5.  print (t + t);
6.  print (t * 3);
7.  print (type(t))
8.  t[2] = "hi";
```

Output:

```
('python', 2)
('hi',)
```

```
('hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2)
('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)
<type 'tuple'>
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Methods of Tuple in Python:

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

2.2(e) Dictionaries & Methods :

Dictionary is an ordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

```
1.  d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};
2.  print("1st name is "+d[1]);
```

```
3.          print("2nd name is "+ d[4]);
4.          print (d);
5.          print (d.keys());
6.          print (d.values());
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
[1, 2, 3, 4]
['Jimmy', 'Alex', 'john', 'mike']
```

Methods of Dictionary in Python:

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key.

	If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

2.2(f) Set & Methods:

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)
```

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).

However, the set itself is mutable. We can add or remove items from it.

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

How to create a set?

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set().

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

Python has a set of built-in methods that you can use on sets.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element

symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

3.Operators

Python Operators:

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a particular programming language. Python provides a variety of operators described as follows.

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

3.1 Arithmetic operators:

Arithmetic operators are used to perform arithmetic operations between two operands. It includes +(addition), - (subtraction),

*(multiplication), /(divide), %(reminder), //(floor division), and exponent (**).

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a ? b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

3.2 Comparison operators:

Comparison operators are used to comparing the value of the two operands and returns boolean true or false accordingly. The comparison operators are described in the following table.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
<>	If the value of two operands is not equal, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

3.3 Assignment operators:

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Description
=	It assigns the the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.

//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

3.4 Bitwise operator:

The bitwise operators perform bit by bit operation on the values of the two operands.

For example,

```
1.  if a = 7;
2.    b = 6;
3.  then, binary (a) = 0111
4.     binary (b) = 0011
5.
6.  hence, a & b = 0011
7.       a | b = 0111
8.            a ^ b = 0100
9.        ~ a = 1000
```

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting

	bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

3.5 Logical operators:

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression **a** is true then not (a) will be false and vice versa.

3.6 Membership operators:

Python membership operators are used to check the membership of value inside a data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

3.7 Identity operators:

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.

Operator Precedence:

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

Operator	Description
**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus and minus.
* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus and minus
>> <<	Left shift and right shift
&	Binary and.
^ |	Binary xor and or
<= < > >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

4. ITERABLE LOOPS

4.1 For loop & Examples and Structure:

Python for loop:

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

```
for iterating_var in sequence:
    statement(s)
```

Example:

```
i=1
n=int(input("Enter the number up to which you want to print the natural numb ers?"))
for i in range(0,10):
    print(i,end = ' ')
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

Python for loop example : printing the table of the given number

```
i=1;
```

```
2.          num = int(input("Enter a number:"));
3.          for i in range(1,11):
4.              print("%d X %d = %d"%(num,i,num*i));
```

Output:

```
Enter a number:10
10 X 1 = 10
10 X 2 = 20
10 X 3 = 30
10 X 4 = 40
10 X 5 = 50
10 X 6 = 60
10 X 7 = 70
10 X 8 = 80
10 X 9 = 90
10 X 10 = 100
```

Nested for loop in python:

Python allows us to nest any number of for loops inside a for loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax of the nested for loop in python is given below.

```
1.          for iterating_var1 in sequence:
2.              for iterating_var2 in sequence:
3.                  #block of statements
4.          #Other statements
```

Example 1:

```
1.          n = int(input("Enter the number of rows you want to print?"))
2.          i,j=0,0
3.          for i in range(0,n):
4.              print()
5.              for j in range(0,i+1):
6.                  print("*",end="")
```

Output:

```
Enter the number of rows you want to print?5
*
**
***
****
*****
```

Using else statement with for loop

Unlike other languages like C, C++, or Java, python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

Example 1:

```
1.          for i in range(0,5):
2.              print(i)
3.          else:print("for loop completely exhausted, since there is no break.");
```

In the above example, for loop is executed completely since there is no break statement in the loop. The control comes out of the loop and hence the else block is executed.

Output:

```
0
1
2
```

3

4

for loop completely exhausted, since there is no break.

Example 2:

```
1.      for i in range(0,5):
2.          print(i)
3.          break;
4.      else:print("for loop is exhausted");
5.      print("The loop is broken due to break statement...came out of loop")
```

In the above example, the loop is broken due to break statement therefore the else statement will not be executed. The statement present immediate next to else block will be executed.

Output:

0

The loop is broken due to break statement...came out of loop

4.2 While loop &examples and Structure:

Python while loop:

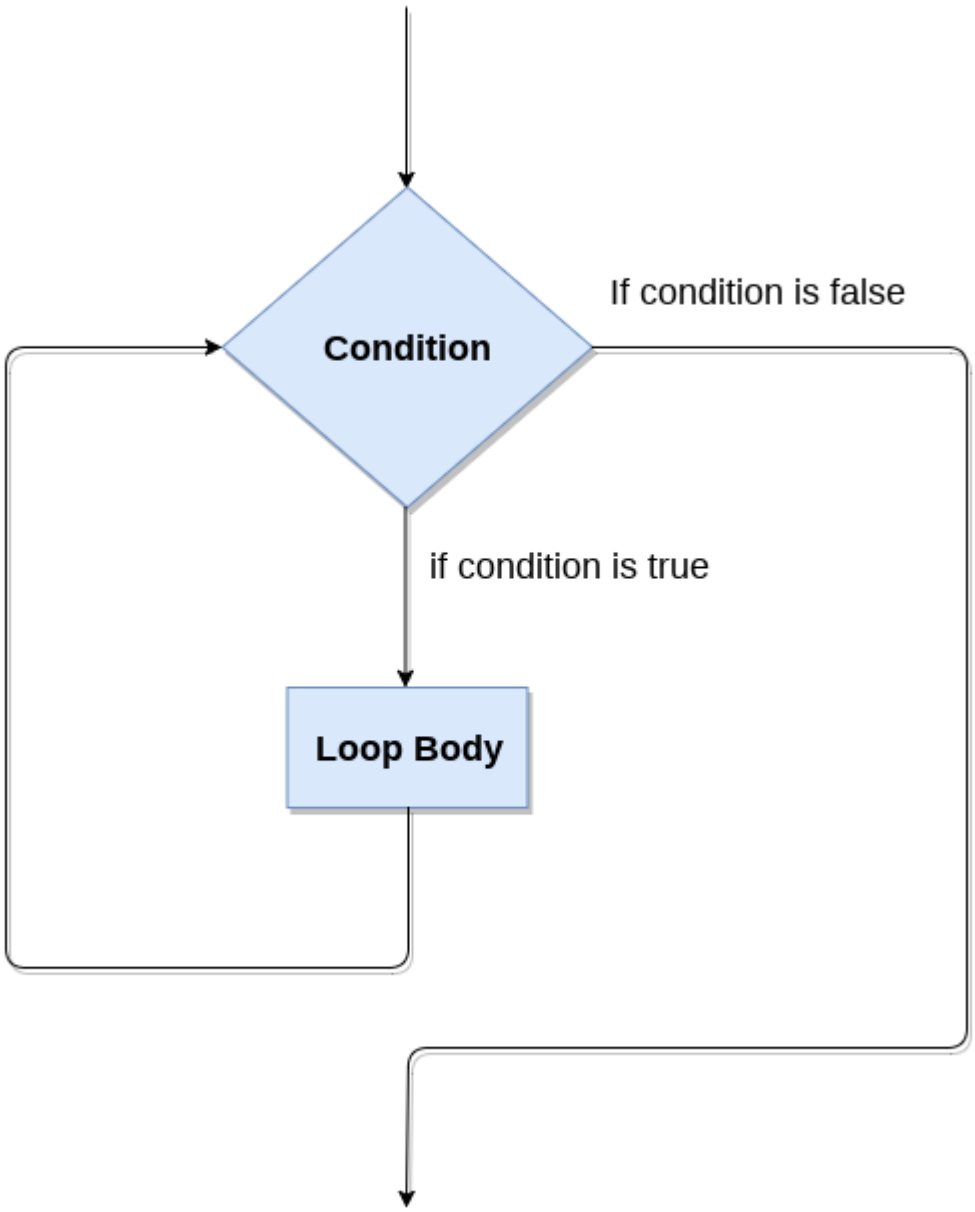
The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.

It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax is given below.

```
1.      while expression:
2.          statements
```

Here, the statements can be a single statement or the group of statements. The expression should be any valid python expression resulting into true or false. The true is any non-zero value.



Example 1:

```
i=1;
while i<=10:
    print(i);
    i=i+1;
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Example 2:

```
i=1
number=0
b=9
number = int(input("Enter the number?"))
while i<=10:
    print("%d X %d = %d \n"%(number,i,number*i));
    i = i+1;
```

Output:

```
Enter the number?10
```

```
10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100
```

Infinite while loop:

If the condition given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition whereas 0 indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

Example 1:

```
1.  while (1):
```

```
2.          print("Hi! we are inside the infinite while loop");
```

Output:

```
Hi! we are inside the infinite while loop
(infinite times)
```

Example 2:

```
1.      var = 1
2.      while var != 2:
3.          i = int(input("Enter the number?"))
4.          print ("Entered value is %d"%(i))
```

Output:

```
Enter the number?102
Entered value is 102
Enter the number?102
Entered value is 102
Enter the number?103
Entered value is 103
Enter the number?103
(infinite loop)
```

Using else with Python while loop:

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Consider the following example.

```
1.      i=1;
2.      while i<=5:
3.          print(i)
4.          i=i+1;
5.      else:print("The while loop exhausted");
```

Output:

```
1
2
3
4
5
The while loop exhausted
```

Example 2:

```
1.      i=1;
2.      while i<=5:
3.          print(i)
4.          i=i+1;
5.          if(i==3):
6.              break;
7.      else:print("The while loop exhausted");
```

Output:

```
1
2
```

Python break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

```
#loop statements
break;
```

Example 1:

```
list =[1,2,3,4]
count = 1;
for i in list:
    if i == 4:
        print("item matched")
        count = count + 1;
        break
```

```
8.          print("found at",count,"location");
```

Output:

```
item matched
found at 2 location
```

Example 2:

```
1.          str = "python"
2.          for i in str:
3.              if i == 'o':
4.                  break
5.              print(i);
```

Output:

```
p
y
t
h
```

Example 3: break statement with while loop:

```
1.          i = 0;
2.          while 1:
3.              print(i," ",end=""),
4.              i=i+1;
5.              if i == 10:
6.                  break;
7.          print("came out of while loop");
```

Output:

```
0  1  2  3  4  5  6  7  8  9  came out of while loop
```

Example 3:

```
1.          n=2
2.          while 1:
3.             i=1;
4.             while i<=10:
5.                print("%d X %d = %d\n"%(n,i,n*i));
6.                i = i+1;
7.             choice = int(input("Do you want to continue printing the table, press 0 for n
               o?"))
8.             if choice == 0:
9.                break;
10.            n=n+1
```

Output:

```
2 X 1 = 2


2 X 2 = 4


2 X 3 = 6


2 X 4 = 8


2 X 5 = 10


2 X 6 = 12


2 X 7 = 14


2 X 8 = 16
```

```
2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0
```

Python continue Statement

The continue statement in python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

The syntax of Python continue statement is given below.

```
1.          #loop statements
2.          continue;
3.          #the code to be skipped
```

Example 1:

```
1.          i = 0;
2.          while i!=10:
3.              print("%d"%i);
4.              continue;
5.              i=i+1;
```

Output:

```
infinite loop
```

Example 2:

```
1.          i=1; #initializing a local variable
2.          #starting a loop from 1 to 10
3.          for i in range(1,11):
4.              if i==5:
5.                  continue;
6.              print("%d"%i);
```

Output:

```
1
2
3
4
6
7
8
9
10
```

Pass Statement

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

Pass is also used where the code will be written somewhere but not yet written in the program file.

The syntax of the pass statement is given below.

Example:

```
1.          list = [1,2,3,4,5]
2.          flag = 0
3.          for i in list:
4.              print("Current element:",i,end=" ");
5.              if i==3:
6.                  pass;
7.                  print("\nWe are inside pass block\n");
8.                  flag = 1;
9.              if flag==1:
10.                 print("\nCame out of pass\n");
```

11. flag=0;

Output:

```
Current element: 1 Current element: 2 Current element: 3
We are inside pass block


Came out of pass

Current element: 4 Current element: 5
```

5. DECISION MAKING LOOPS

5.1 If & Structure:

Python If-else statements:

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that,

	it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

Indentation in Python:

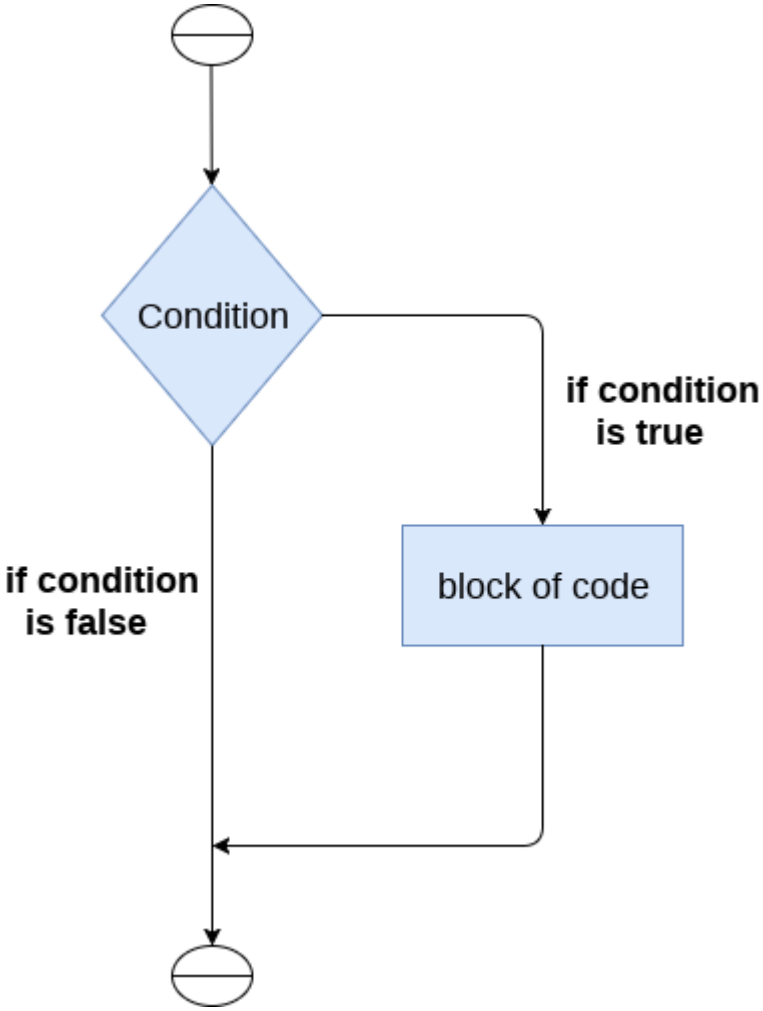
For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

The if statement:

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
1.  if expression:
2.      statement
```

Example 1:

```
1.  num = int(input("enter the number?"))
2.  if num%2 == 0:
3.      print("Number is even")
```

Output:

```
enter the number?10
Number is even
```

Example 2 : Program to print the largest of the three numbers.

```
a = int(input("Enter a? "));
b = int(input("Enter b? "));
c = int(input("Enter c? "));
if a>b and a>c:
    print("a is largest");
if b>a and b>c:
    print("b is largest");
if c>a and c>b:
    print("c is largest");
```

Output:

```
Enter a? 100
Enter b? 120
Enter c? 130
c is largest
```

5.2 Elif & Structure:

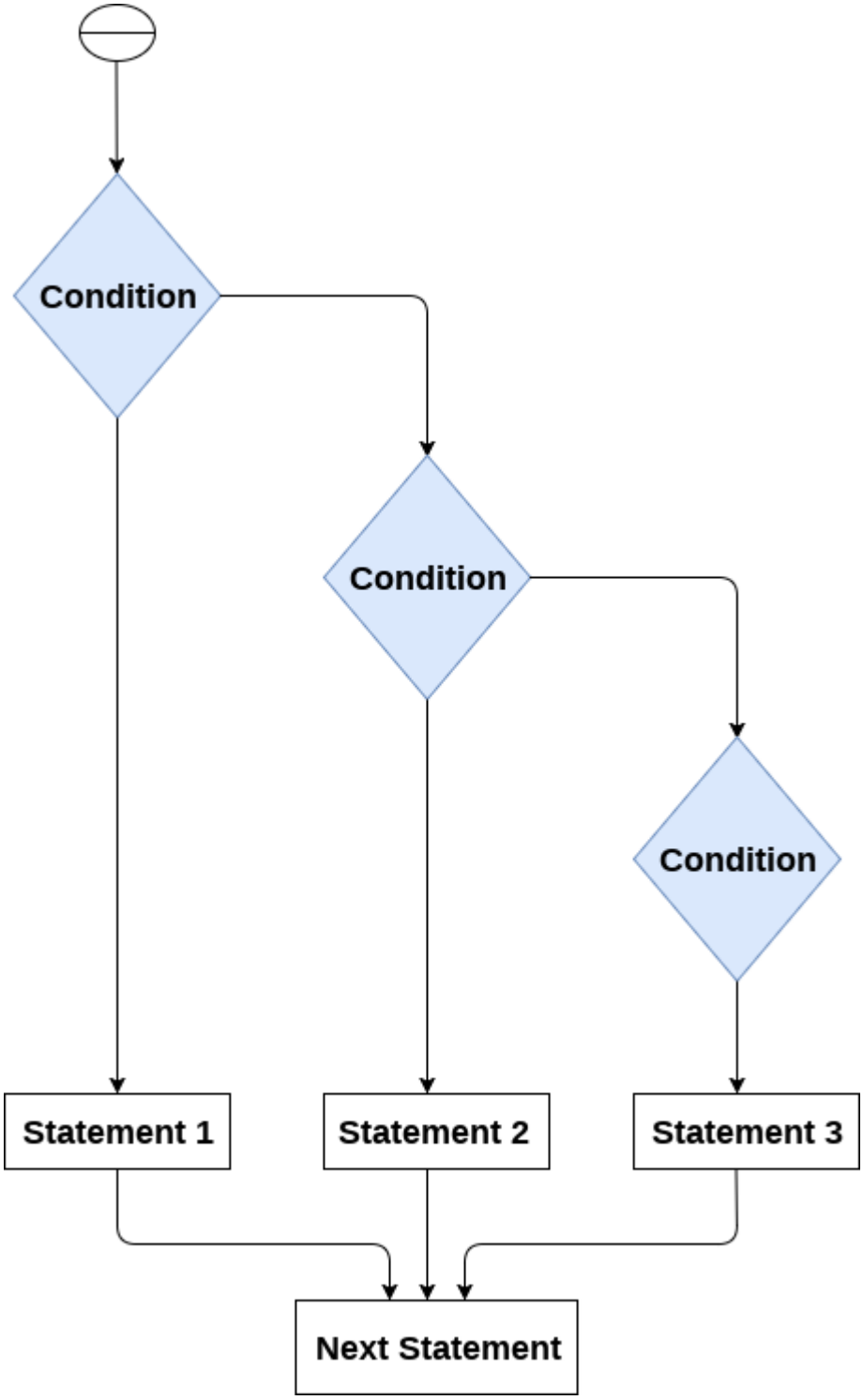
The elif statement:

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.

```
1.      if expression 1:
2.          # block of statements
3.
4.      elif expression 2:
5.          # block of statements
6.
7.      elif expression 3:
8.          # block of statements
9.
10.     else:
11.         # block of statements
```


Condition
Condition
Condition
Statement 1
Statement 2
Statement 3
Next Statement

Example 1:

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

Example 2:

```
marks = int(input("Enter the marks? "))
f marks > 85 and marks <= 100:
  print("Congrats ! you scored grade A ...")
lif marks > 60 and marks <= 85:
  print("You scored grade B + ...")
lif marks > 40 and marks <= 60:
  print("You scored grade B ...")
lif (marks > 30 and marks <= 40):
  print("You scored grade C ...")
lse:
  print("Sorry you are fail ?")
```

6. FUNCTIONS

6.1 Types of Fuctions:

In this article, you'll learn about functions; what is a function, the syntax, components and types of a function. Also, you'll learn to create a function in Python.

` A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Function Arguments:

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments
- Keyword variable length arguments

Defining a Function:

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

1. Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).

2. Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

3. The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.

4. The code block within every function starts with a colon (:) and is indented.

5. The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):

   "This prints a passed string into this function"

   print str

   return
```

Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
#!/usr/bin/python



# Function definition is here

def printme( str ):

   "This prints a passed string into this function"
```

```
   print str

   return;


# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")
```

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

6.2 Requried argument function:

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
#!/usr/bin/python

# Function definition is here def printme( str ):

   "This prints a passed string into this function"

   print str

   return;

# Now you can call printme function

printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):
   File "test.py", line 11, in <module>
      printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

6.3 Default argument function:

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
#!/usr/bin/python

# Function definition is heredef printinfo( name, age = 35 ):

   "This prints a passed info into this function"

   print "Name: ", name

   print "Age ", age

   return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

6.4 Variable length function:

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length*arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
```

```
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

```
#!/usr/bin/python

# Function definition is heredef printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

6.5 Keyword argument function:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
#!/usr/bin/python

# Function definition is heredef printme( str ):

   "This prints a passed string into this function"

   print str

   return;

# Now you can call printme function

printme( str = "My string")
```

When the above code is executed, it produces the following result −

```
My string
```

The following example gives more clear picture. Note that the order of parameters does not matter.

```
#!/usr/bin/python

# Function definition is heredef printinfo( name, age ):

   "This prints a passed info into this function"

   print "Name: ", name

   print "Age ", age

   return;

# Now you can call printinfo function
```

```
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result –

```
Name:  miki
Age  50
```

6.6 Keyword variable length argument function:

The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the kwargs as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the kwargs there doesn't seem to be any order in which they were printed out.

```
# Python program to illustrate
# *kargs for variable number of keyword arguments

def myFun(**kwargs):
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

# Driver code
myFun(first ='Geeks', mid ='for', last='Geeks')
```

Output:

```
last == Geeks
mid == for
first == Geeks
```

6.7 Anonymous function(Lambda):

The *Anonymous* Functions :

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

1. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

2. An anonymous function cannot be a direct call to print because lambda requires an expression

3. Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

4. Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax:

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Following is the example to show how *lambda* form of function works −

```
#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a functionprint "Value of total : ", sum( 10, 20 )print
"Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result –

```
Value of total :  30
Value of total :  40
```

If you want clear explanation for anonymous function.The link is provided down below.Please cheak out.

https://www.youtube.com/watch?v=hYzwCsKGRrg

7. FILE HANDLING

File Handling in Python:

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 alike 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.

Working of open() function

We use **open ()** function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use **open()** function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: **open(filename, mode)**. There are three kinds of mode, that Python provides and how files can be opened:

- " **r** ", for reading.
- " **w** ", for writing.
- " **a** ", for appending.
- " **r+** ", for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " **r** " by default. Let's look at this program and try to analyze how the read mode works:

```
# a file named "geek", will be opened with the reading mode.
file = open('Anjan.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)
```

The open command will open the file in the read mode and the for loop will print each line present in the file.

7.1 Read-only File:

Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

```
# Python code to illustrate read() mode
file = open("file.text", "r")
print file.read()
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print file.read(5)
```

7.2 Write-only File:

Creating a file using write() mode

Let's see how to create a file and how write mode works:
To manipulate the file, write the following in your Python environment

```
# Python code to create a file
```

```
file = open('Anjan.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

7.3 Append File:

Let’s see how the append mode works:

```
# Python code to illustrate append() mode
file = open('Anjan.txt','a')
file.write("This will add this line")
file.close()
```

There are also various other commands in file handling that is used to handle various tasks like:

8. EXCEPTION HANDLING

8.1 Exception:

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

8.2 Exception Handling:

Exception Handling − This would be covered in this tutorial. Here is a list standard Exceptions available in Python − Standard Exceptions.

We have explored basic python till now

If you are a beginner, then you might be getting a lot of Tracebacks till now. The tracebacks are generated due to runtime errors.

Like other languages, python also provides the runtime errors via exception handling method with the help of try-except. Some of the standard exceptions which are most frequent include IndexError, ImportError, IOError, ZeroDivisionError, TypeError.

Exception is the base class for all the exceptions in python. You can check the exception hierarchy here.

Let us try to access the array element whose index is out of bound and handle the corresponding exception.

```
# Python program to handle simple runtime error

a = [1, 2, 3]
try:
    print "Second element = %d" %(a[1])

    # Throws error since there are only 3 elements in array
    print "Fourth element = %d" %(a[3])

except IndexError:
    print "An error occurred"
```

Output:

```
Second element = 2
An error occurred
```

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed.

```
# Program to handle multiple errors with one except statement
```

```
try :
    a = 3
    if a < 4 :

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print "Value of b = ", b

# note that braces () are necessary here for multiple exceptions
except(ZeroDivisionError, NameError):
    print "\nError Occurred and Handled"
```

Output:

```
Error Occurred and Handled
```

If you change the value of ‘a’ to greater than or equal to 4, the the output will be

```
Value of b =
Error Occurred and Handled
```

The output above is so because as soon as python tries to access the value of b, NameError occurs.

8.3 Custom exception:

User-defined Exceptions in Python with Examples

Prerequisite- This article is an extension to Exception Handling.

Python throws errors and exceptions, when there is a code gone wrong, which may cause program to stop abruptly. Python also provides exception handling method with the help of try-except. Some of the standard exceptions which are most frequent include IndexError, ImportError, IOError, ZeroDivisionError, TypeError and FileNotFoundError. A user can create his own error using exception class.

Creating User-defined Exception

Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in **"Error"**similar to naming of the standard exceptions in python. For example:

```
# A python program to create user-defined exception


# class MyError is derived from super class Exception

class MyError(Exception):
```

```
        # Constructor or Initializer
        def __init__(self, value):
            self.value = value

        # __str__ is to print() the value
        def __str__(self):
            return(repr(self.value))

    try:
        raise(MyError(3*2))

    # Value of Exception is stored in error
    except MyError as error:
        print('A New Exception occured: ',error.value)
```

Output:

```
('A New Exception occured: ', 6)
```

9. REGULAR EXPRESSION(Regx)

In this tutorial, you will learn about regular expressions (RegEx), and use Python's re module to work with RegEx (with the help of examples).

Python RegEx:

Python has a module named re to work with regular expressions. To use it, we need to import the module.

```
1.      import re
```

The module defines several functions and constants to work with RegEx.

9.1 Match:

Here, match contains a match object.

Match object

You can get methods and attributes of a match object using dir() function.

Some of the commonly used methods and attributes of match objects are:

---

match.group()

The group() method returns the part of the string where there is a match.

Example 6: Match object

```
import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
  print(match.group())
else:
  print("pattern not found")

# Output: 801 35
```

Here, match variable contains a match object.

Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the part of the string of these parenthesized subgroups. Here's how:

```
>>> match.group(1)
'801'

>>> match.group(2)
'35'
>>> match.group(1, 2)
('801', '35')

>>> match.groups()
('801', '35')
```

match.start(), match.end() and match.span()

The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.

```
>>> match.start()
2
>>> match.end()
8
```

The span() function returns a tuple containing start and end index of the matched part.

```
>>> match.span()
(2, 8)
```

---

match.re and match.string

The re attribute of a matched object returns a regular expression object. Similarly, stringattribute returns the passed string.

```
>>> match.re
re.compile('(\\d{3}) (\\d{2})')

>>> match.string
'39801 356, 2102 1111'
```

We have covered all commonly used methods defined in the re module. If you want to learn more, visit Python 3 re module.

9.2 Search:

re.search()

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

```
match = re.search(pattern, str)
```

Example 5: re.search()

```
import re

string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

# Output: pattern found inside the string
```

9.3 Findall():

re.findall()

The re.findall() method returns a list of strings containing all matches.

Example 1: re.findall()

```
# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'
```

```
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

If the pattern is no found, re.findall() returns an empty list.

9.4 Split():

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example 2: re.split()

```
import re

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

If the pattern is no found, re.split() returns a list containing an empty string.

You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.

```
import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'

# maxsplit = 1
```

```
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

By the way, the default value of maxsplit is 0; meaning all possible splits.

9.5 Compile:

Function compile():

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

Module Regular Expression is imported using __import__().

import re

compile() creates regular expression character class [a-e],

which is equivalent to [abcde].

class [abcde] will match with string with 'a', 'b', 'c', 'd', 'e'.

p = re.compile('[a-e]')

findall() searches for the Regular Expression and return a list upon finding

print(p.findall("Aye, said Mr. Gibenson Stark"))

Output:

```
['e', 'a', 'd', 'b', 'e', 'a']
```

Understanding the Output:
First occurrence is ‘e’ in “Aye” and not ‘A’, as it being Case Sensitive.
Next Occurrence is ‘a’ in “said”, then ‘d’ in “said”, followed by ‘b’ and ‘e’ in “Gibenson”, the Last ‘a’ matches with “Stark”.

Metacharacter blackslash ‘\’ has a very important role as it signals various sequences. If the blackslash is to be used without its special meaning as metacharcter, use’\\’

```
\d   Matches any decimal digit, this is equivalent
     to the set class [0-9].
\D   Matches any non-digit character.
\s   Matches any whitespace character.
\S   Matches any non-whitespace character
\w   Matches any alphanumeric character, this is
     equivalent to the class [a-zA-Z0-9_].
\W   Matches any non-alphanumeric character.
```

Set class [\s,.] will match any whitespace character, ‘,’, or,’.’ .

```
import re

# \d is equivalent to [0-9].
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

# \d+ will match a group on [0-9], group of one or greater size
p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

Output:

```
['1', '1', '4', '1', '8', '8', '6']
['11', '4', '1886']
```

```
import re
```

```
# \w is equivalent to [a-zA-Z0-9_].
p = re.compile('\w')
print(p.findall("He said * in some_lang."))

# \w+ matches to group of alphanumeric charcter.
p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he said *** in some_language."))

# \W matches to non alphanumeric characters.
p = re.compile('\W')
print(p.findall("he said *** in some_language."))
```

Output:

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a', 'n', 'g']
['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_language']
[' ', ' ', '*', '*', '*', ' ', ' ', '.']
```

```
import re

# '*' replaces the no. of occurrence of a character.
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

Output:

```
['ab', 'abb', 'a', 'abbb']
```

Understanding the Output:
Our RE is ab*, which ‘a’ accompanied by any no. of ‘b’s, starting from 0.
Output ‘ab’, is valid because of singe ‘a’ accompanied by single ‘b’.
Output ‘abb’, is valid because of singe ‘a’ accompanied by 2 ‘b’.
Output ‘a’, is valid because of singe ‘a’ accompanied by 0 ‘b’.
Output ‘abbb’, is valid because of singe ‘a’ accompanied by 3 ‘b’.

9.6 Sub():

re.sub()

The syntax of re.sub() is:

```
re.sub(pattern, replace, string)
```

The method returns a string where matched occurrences are replaced with the content of replace variable.

Example 3: re.sub()

```
# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
```

```
print(new_string)

# Output: abc12de23f456
```

If the pattern is no found, re.sub() returns the original string.

You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.

```

import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)

# Output:
# abc12de 23
# f45 6
```

---

re.subn():

The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

Example 4: re.subn()

```
# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.subn(pattern, replace, string)
print(new_string)

# Output: ('abc12de23f456', 4)
```

10. FILTER USING LAMBDA FUNCTION

10.1 Filter:

Python lambda (Anonymous Functions) | filter, map, reduce:

In Python, anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions. It has the following syntax:

```
lambda arguments: expression
```

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Let's look at this example and try to understand the **difference between a normal def defined function and lambda function**. This is a program that returns the cube of a given value:

```
# Python code to illustrate cube of a number
# showing difference between def() and lambda().
def cube(y):
    return y*y*y;

g = lambda x: x*x*x
print(g(7))
```

```
print(cube(5))
```

Output:

```
343
125
```

- **Without using Lambda :** Here, both of them returns the cube of a given number. But, while using def, we needed to define a function with a name cube and needed to pass a value to it. After execution, we also needed to return the result from where the function was called using the return keyword.
- **Using Lambda :** Lambda definition does not include a "return" statement, it always contains an expression which is returned. We can also put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all. This is the simplicity of lambda functions.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Use of lambda() with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True. Here is a small program that returns the odd numbers from an input list:

```
# Python code to illustrate
# filter() with lambda()
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0) , li))
```

```
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

10.2 Map:

Use of lambda() with map()

The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item. Example:

```
# Python code to illustrate
# map() with lambda()
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

10.3 Reduce:

Reduce is module that we have to import from the function tools

Like

```
from functools import reduce()
```

The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module.

Example:

```
# Python code to illustrate
# reduce() with lambda()
# to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

Output:

```
193
```

All Operations in one example:

1. a = lambda y : y*10

```
print(a(5))

# def a(y):
    # return y*10
```

2. b = lambda a,b : a+b

```
c = b(5,6)

print("SUM IS:",c)
```

3. d = lambda a,b : a-b

```
e = d(5,6)

print("SUB IS:",e)
```

4. f = lambda a,b : a*b

```
g = f(5,6)

print("MUL IS:",g)
```

5. h = lambda a,b : a/b

```
i = h(5,6)

print("DIV IS:",i)
```

6. x = lambda a, b, c : a + b * c

```
print(x(5, 6, 2))
```

7. def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))

8. def myfunc(n):
    return lambda a : a * n

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

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

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

9. from functools import reduce

```
nums = [1,2,3,4,5,6,7,8,9,10]

evens = list(filter(lambda n : n%2==0,nums))

print(evens)

doubles = list(map(lambda n : n*2,evens))

print(doubles)

sum = (reduce(lambda a,b:a+b,doubles))

print(sum)
```

11. OOPS (OBJECT ORIENTED PROGRAMMING CONCEPTS)

11.1 Types of Oop's Concepts:

Python Object Oriented Programming:

In this article, you'll learn about the Object Oriented Programming (OOP) in Python and their fundamental concept with examples.

Introduction to OOPs in Python:

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.

Polymorphism	A concept of using common operation in different ways for different data input.

11.2 Instance method:

Instance Methods:

The first method on MyClass, called method, is a regular instance method. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, self, which points to an instance of MyClass when the method is called (but of course instance methods can accept more than just one parameter).

Through the self parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, instance methods can also access the class itself through the self.__class__ attribute. This means instance methods can also modify class state.

Class:

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

```
class Parrot:
```

```
    pass
```

Here, we use class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object:

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, obj is object of class Parrot.

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

Example 1: Creating Class and Object in Python:

```
class Parrot:

    # class attribute
    species = "bird"

    # instance attribute
    def __init__(self, name, age):
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

When we run the program, the output will be:

```
Blu is a bird

Woo is also a bird

Blu is 10 years old
```

```
Woo is 15 years old
```

In the above program, we create a class with name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

Then, we access the class attribute using __class __.species. Class attributes are same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

To learn more about classes and objects, go to Python Classes and Objects

Methods:

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python:

```
class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

    def dance(self):
        return "{} is now dancing".format(self.name)

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

When we run program, the output will be:

```
Blu sings 'Happy'

Blu is now dancing
```

In the above program, we define two methods i.e sing() and dance(). These are called instance method because they are called on an instance object i.e blu.

11.3 Inheritance method:

Inheritance:

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python:

```
# parent class
class Bird:

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

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

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

# child class
class Penguin(Bird):

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

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

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

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

When we run this program, the output will be:

```
Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster
```

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim()method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

11.4 Polymorphism method:

Polymorphism:

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

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

Example 5: Using Polymorphism in Python:

```
class Parrot:

    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:

    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")
```

```
# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
flying_test(blu)
flying_test(peggy)
```

When we run above program, the output will be:

```
Parrot can fly

Penguin can't fly
```

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

11.4(a) Method Overriding:

Overriding is a very important part of OOP since it is the feature that makes inheritance exploit its full power. Through method overriding a class may "copy" another class, avoiding duplicated code, and at the same time enhance or customize part of it. Method overriding is thus a strict part of the inheritance mechanism.

overriding methods of a class:

Create a parent class Robot. Then a class that inherits from the class Robot. We add a method action that we override:

```
class Robot:
	def action(self):
		print('Robot action')

class HelloRobot(Robot):
	def action(self):
		print('Hello world')

r = HelloRobot()
r.action()
```

Instance r is created using the class HelloRobot, that inherits from parent class Robot.

The HelloRobot class inherits the method action from its parent class, but its overridden in the class itself.

Execute the program to see:
abstract-base-classes.md

```
Hello world
```

Overriding methods:

The method is overwritten. This is only at class level, the parent class remains intact.
If we add another class that inherits, let's see what happens.

Try the program below and find out why it outputs differently for the method action:

```
class Robot:
        def action(self):
                print('Robot action')

class HelloRobot(Robot):
        def action(self):
                print('Hello world')

class DummyRobot(Robot):
        def start(self):
                print('Started.')

r = HelloRobot()
d = DummyRobot()

r.action()
d.action()
```

11.4(b) Method Overloading:

Python Method Overloading:

Like other languages (for example method overloading in C++) do, python does not supports method overloading. We may overload the methods but can only use the latest defined method.

```
# First product method.
# Takes two argument and print their
# product
def product(a, b):
    p = a * b
    print(p)


# Second product method
```

```
# Takes three argument and print their
# product
def product(a, b, c):
    p = a * b*c
    print(p)

# Uncommenting the below line shows an error
# product(4, 5)

# This line will call the second product method
product(4, 5, 5)
```

Output:

```
100
```

In the above code we have defined two product method, but we can only use the second product method, as python does not supports method overloading. We may define many method of same name and different argument but we can only use the latest defined method. Calling the other method will produce an error. Like here calling will produce an error as the latest defined product method takes three arguments.

However we may use other implementation in python to make the same function work differently i.e. as per the arguments.

filter_none

```
# Function to take multiple arguments
def add(datatype, *args):

    # if datatype is int
```

```
    # initialize answer as 0
    if datatype =='int':
        answer = 0

    # if datatype is str
    # initialize answer as ''
    if datatype =='str':
        answer =''

    # Traverse through the arguments
    for x in args:

        # This will do addition if the
        # arguments are int. Or concatenation
        # if the arguments are str
        answer = answer + x

    print(answer)

# Integer
```

```
add('int', 5, 6)


# String

add('str', 'Hi ', 'Geeks')
```

Output:

```
11
Hi Geek
```

11.5 Encapsulation:

Encapsulation:

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.

Example 4: Data Encapsulation in Python:

```
class Computer:

    def __init__(self):
        self.__maxprice = 900

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

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

c = Computer()
c.sell()

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

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

When we run this program, the output will be:

```
Selling Price: 900

Selling Price: 900

Selling Price: 1000
```

In the above program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

11.6 Abstraction:

Data Abstraction:

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonym because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities. Abstracting something means to give names to things so that the name captures the core of what a function or a whole program does.

11.7 Constructors:

Python Constructor:

A constructor is a special type of method (function) which is used to initialize the instance members of the class.

Constructors can be of two types.

1. Parameterized Constructor
2. Non-parameterized Constructor

Constructor definition is executed when we create the object of this class. Constructors also verify that there are enough resources for the object to perform any start-up task.

Creating the constructor in python:

In python, the method **__init__** simulates the constructor of the class. This method is called when the class is instantiated. We can pass any number of arguments at the time of creating the class object, depending upon **__init__** definition. It is mostly used to initialize the class attributes. Every class must have a constructor, even if it simply relies on the default constructor.

Consider the following example to initialize the Employee class attributes.

Example:

```
1.  class Employee:
2.      def __init__(self,name,id):
3.          self.id = id;
4.          self.name = name;
5.      def display (self):
6.          print("ID: %d \nName: %s"%(self.id,self.name))
7.  emp1 = Employee("John",101)
8.  emp2 = Employee("David",102)
9.
10. #accessing display() method to print employee 1 information
11.
12. emp1.display();
13.
14. #accessing display() method to print employee 2 information
15. emp2.display();
```

Output**:**

```
ID: 101
Name: John
ID: 102
```

Name: David

Example: Counting the number of objects of a class:

```
class Student:
    count = 0
    def __init__(self):
        Student.count = Student.count + 1
s1=Student()
s2=Student()
s3=Student()
print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor Example:

```
class Student:
    # Constructor - non parameterized
    def __init__(self):
        print("This is non parametrized constructor")
    def show(self,name):
        print("Hello",name)
student = Student()
student.show("John")
```

Output:

```
This is non parametrized constructor
Hello John
```

Python Parameterized Constructor Example:

```
class Student:
    # Constructor - parameterized
    def __init__(self, name):
        print("This is parametrized constructor")
        self.name = name
    def show(self):
        print("Hello",self.name)
student = Student("John")
student.show()
```

Output:

This is parametrized constructor

Hello John

11.8 Iterators:

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.

11.9 Generators:

Generators in Python

Prerequisites: Yield Keyword and Iterators

There are two terms involved when we discuss generators.

1. **Generator-Function :** A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def

contains yield, the function automatically becomes a generator function.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def simpleGeneratorFun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for value in simpleGeneratorFun():
    print(value)
```

Output :

```
1
2
3
```

2.Generator-Object : Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a "for in" loop (as shown in the above program).

```
# A Python program to demonstrate use of
# generator object with next()

# A generator function
def simpleGeneratorFun():
    yield 1
    yield 2
```

```
    yield 3

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next
print(x.next()); # In Python 3, __next__()
print(x.next());
print(x.next());
```

Output :

```
1
2
3
```

So a generator function returns an generator object that is iterable, i.e., can be used as an Iterators .

As another example, below is a generator for Fibonacci Numbers.

```
# A simple generator for Fibonacci Numbers
def fib(limit):

    # Initialize first two Fibonacci Numbers
    a, b = 0, 1

    # One by one yield next Fibonacci Number
    while a < limit:
```

```
        yield a
        a, b = b, a + b

# Create a generator object
x = fib(5)

# Iterating over the generator object using next
print(x.next()); # In Python 3, __next__()
print(x.next());
print(x.next());
print(x.next());
print(x.next());

# Iterating over the generator object using for
# in loop.
print("\nUsing for in loop")
for i in fib(5):
    print(i)
```

Output :

```
0
1
1
2
3

Using for in loop
0
1
1
```

```
2
3
```

Applications : Suppose we to create a stream of Fibonacci numbers, adopting the generator approach makes it trivial; we just have to call next(x) to get the next Fibonacci number without bothering about where or when the stream of numbers ends.
A more practical type of stream processing is handling large data files such as log files. Generators provide a space efficient method for such data processing as only parts of the file are handled at one given point in time. We can also use Iterators for these purposes, but Generator provides a quick way (We don't need to write __next__ and __iter__ methods here).

Refer below link for more advanced applications of generators in Python.
http://www.dabeaz.com/finalgenerator/

11.10 Closures:

Python Closures:

Before seeing what a closure is, we have to first understand what are nested functions and non-local variables.

Nested functions in Python

A function which is defined inside another function is known as nested function. Nested functions are able to access variables of the enclosing scope.
In Python, these non-local variables can be accessed only within their scope and not outside their scope. This can be illustrated by following example:

```
# Python program to illustrate

# nested functions

def outerFunction(text):

    text = text


    def innerFunction():

        print(text)
```

```
    innerFunction()


if __name__ == '__main__':
    outerFunction('Hey!')
```

As we can see innerFunction() can easily be accessed inside the outerFunction body but not outside of it's body. Hence, here, innerFunction() is treated as nested Function which uses **text** as non-local variable.

Python Closures

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

```
# Python program to illustrate

# closures

def outerFunction(text):

    text = text


    def innerFunction():

        print(text)
```

```
        return innerFunction # Note we are returning function
WITHOUT parenthesis


    if __name__ == '__main__':
        myFunction = outerFunction('Hey!')
        myFunction()
```

Output:

```
omkarpathak@omkarpathak-Inspiron-3542:
~/Documents/Python-Programs/$ python Closures.py
Hey!
```

11.11 Decorators:

Decorators in Python:

In Python, functions are the first class objects, which means that –

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

```
@gfg_decorator
def hello_decorator():
    print("Gfg")

'''Above code is equivalent to -

def hello_decorator():
    print("Gfg")

hello_decorator = gfg_decorator(hello_decorator)'''
```

In the above code, gfg_decorator is a callable function, will add some code on the top of some another callable
function, hello_decorator function and return the wrapper function.

Decorator can modify the behavior:

```
# defining a decorator
def hello_decorator(func):

    # inner1 is a Wrapper function in
    # which the argument is called

    # inner function can access the outer local
    # functions like in this case "func"
    def inner1():
        print("Hello, this is before function execution")
```

```
        # calling the actual function now
        # inside the wrapper function.
        func()

        print("This is after function execution")

    return inner1


# defining a function, to be called inside wrapper
def function_to_be_used():
    print("This is inside the function !!")


# passing 'function_to_be_used' inside the
# decorator to control its behavior
function_to_be_used = hello_decorator(function_to_be_used)


# calling the function
function_to_be_used()
```

Output:

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

REST WILL BE EXPLAINED VIRTUALLY ,BECAUSE IT TAKE LONGER TIME TO EXPLAIN..

PREPARED BY "ANJAN KUMAR REDDY"