

Python

Introduction

- General-purpose interpreted, interactive, object-oriented, and high-level programming language
- Created by Guido van Rossum during 1985-1990
- Source code is also available under the GNU General Public License (GPL)

Introduction

- **Python is Interpreted** – Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive** – You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Installing Python

- Open a terminal window and type "python" to find out if it is already installed and which version is installed. Python works on all major platforms.
- The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python <https://www.python.org/>
- You can download Python documentation from <https://www.python.org/doc/>. The documentation is available in HTML, PDF, and PostScript formats.
- Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

First Program

- `print "Hello, Python!"`
- In cmd
- In IDLE
- In notepad
- In any other 3rd party python environment/IDE
- Virtual in env

Programming A way to tell your computer to perform task/tasks

- Declarative Instructions
 - Identifiers, x=10,_xth12
 - Data types, number, decimal number, character, group of data, int, float, char “,”string”,[],{}
- Calculative Instructions
 - Operators +-/*
- Control flow instructions
 - Statements
 - loops

Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).
- Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Naming conventions for Python identifiers

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Python keywords

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

- Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

if True:

 print "Answer"

 print "True"

else:

 print "Answer"

 print "False"

Multi-Line Statements

- Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example –

```
total = item_one + \  
item_two + \  
item_three
```
- Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example –

```
days = ['Monday', 'Tuesday', 'Wednesday',  
'Thursday', 'Friday']
```

Quotation in Python

- Python accepts single ('), double (") and triple (''' or ''') quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines. For example, all the following are legal –
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is made up of
multiple lines and sentences."""

Comments

- A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

```
# First comment
```

```
print "Hello, Python!"
```

```
# second comment
```

- This produces the following result –
Hello, Python!

Multiple Statements on a Single Line

- The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block. Here is a sample snip using the semicolon –
`import sys; x = 'foo'; sys.stdout.write(x + '\n')`
- A group of individual statements, which make a single code block are called **suites** in Python.

if expression :

 suite

elif expression :

 suite

else :

 suite

Python Variables

- Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.
- Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.
- Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.
 - `counter = 100` # An integer assignment
 - `miles = 1000.0` # A floating point
 - `name = "John123"` # A string
 - `Bool1=True` #boolean
 - `a = b = c = 1` #multiple assignment
 - `a,b,c = 1,2,"john"` #multiple assignment
 - `X=True` #boolean

Python standard datatypes

- Boolean
 - True
 - False

Python Standard Datatypes

- Numbers: int float $123+125=248$; $a=123$
- String "Abab@#\$123-145";
 $x="123";y="125";z=x+y=123125$
 $"123"+"125"="123125"$
- List [1,2,3,4.5,"22ttt","z"]
- Tuple (4,66,"ggg") #read only data
- Dictionary{key:value,key2:value2}

Python Standard Datatypes

- Numbers 2^{-32} to 2^{32}
 - int (signed integers), $x=10$ or $x=-10$
 - long (long integers, they can also be represented in octal and hexadecimal), $x=51924361L$
 - float (floating point real values), $x=15.20$ or $x=-91.23$
 - complex (complex numbers), $x=3.14j$ or $x=-9+7j$, $j=\sqrt{-1}$
- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
- A complex number consists of an ordered pair of real floating-point numbers denoted by $x + yj$, where x and y are the real numbers and j is the imaginary unit.

Python Standard Datatypes

- String
- continuous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and working their way from - 1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.
 - `str = 'Hello World!'`
 - `print str` # Prints complete string #Hello World!
 - `print str[0]` # Prints first character of the string #H
 - `print str[2:5]` # Prints characters starting from 3rd to 5th #llo
 - `print str[2:]` # Prints string starting from 3rd character #llo World!
 - `print str * 2` # Prints string two times #Hello World!Hello World!
 - `print str + "TEST"` # Prints concatenated string #Hello World!TEST

Strings

- Accessing string elements:
 - `print str[0:5]`
 - `Print str [2:]`
- String Literals:/ non printable characters
 - `\a`
 - `\b`
 - `\f`
 - `\n`
 - `\r`
 - `\t`
 - `\v`

Strings

- Special Operators:
 - + gives string concatenation
 - * gives string repetition
 - [] gives character slice
 - [:] gives range of character slice
 - in and not in gives membership status
 - str="SmartYug" print('S' in str)
 - 'a' in str
 - 'x' not in str
 - r/R preceded before quotation marks neglects escape characters
 - print r"hello yes\no world"
 - print r"\n"
 - % performs string formatting: format specifiers
 - print "My name is %s and age is %d years!" % ('SmartYug', 1)
 - Supported format specifiers: c, s, i/d, u, o, x, f, e,

Strings Methods

- `str.center(total_width,fillchar)`
- `str.ljust(total_width,fillchar)`
- `str.rjust(total_width,fillchar)`
- `str.zfill(total_width)`
- `str.count(char/string,begin,end)`
- `str.endswith(char/string,begin,end)`
- `str.startswith(char/str,begin,end)`
- `str.find(char/string,begin,end)`
- `rfind(char/string,begin,end)`
- `str.index(char/string,begin,end)`
- `str.rindex(char/string,begin,end)`
- `str.isalnum()`
- `Str.isdecimal()`
- `Str.isidentifier()`, a-z,0-9, _
- `Str.isnumeric()`
- `Str.isprintable()`
- `str.isalpha()`
- `str.isdigit()`
- `str.islower()`
- `str.isupper()`
- `str.isspace()`
- `str.istitle()`
- `str.lstrip(char/str);default space/tab`
- `str.rstrip(char/str)`
- `str.strip(char/str)`
- `str.split(char/str,count)`
- `str.splitlines(count)`
- `str.maketrans(intab, outtab);used with`
- `str.translate(table,deletechar)`
- `Str.encode()`
- `str.replace(old,new,max)`
- `str.join(seq)`
- `str.expandtabs();8 is bydefault`
- `Str.format()`
- `Str.format_map`
- `str.capitalize()`
- `Str.casefold()`
- `str.upper()`
- `str.lower()`
- `str.swapcase()`
- `str.title()`
- `Type(str)`
- `len(str)`
- `max(str)`
- `min(str)`

Python Standard Datatypes







- List
- A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.
 - list = ['abcd', 786 , 2.23, 'john', 70.2]
 - tinylis = [123, 'john']
 - print list # Prints complete list `#['abcd', 786, 2.23, 'john', 70.2000000000000003]`
 - print list[0] # Prints first element of the list `#abcd`
 - print list[1:3] # Prints elements starting from 2nd till 4th `#[786, 2.23]`
 - print list[2:] # Prints elements starting from 3rd element `#[2.23, 'john', 70.2000000000000003]`
 - print tinylis * 2 # Prints list two times `#[123, 'john', 123, 'john']`
 - print list + tinylis `#['abcd', 786, 2.23, 'john', 70.2000000000000003, 123, 'john']`

Lists & Tuples










- Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	1 2 3	Iteration

Functions with list

Sr.No.	Function with Description
1	cmp(list1, list2)  Compares elements of both lists. <ul style="list-style-type: none">• Available in python 2 only
2	len(list)  Gives the total length of the list.
3	max(list)  Returns item from the list with max value.
4	min(list)  Returns item from the list with min value. <ul style="list-style-type: none">• Sum(list)
5	list(seq)  Converts a tuple into list.
5	tuple(seq)  Converts a list into tuple.

List Methods

Sr.No.	Methods with Description
1	<code>list.append(obj)</code>  Appends object obj to list
2	<code>list.count(obj)</code>  Returns count of how many times obj occurs in list
3	<code>list.extend(seq)</code>  Appends the contents of seq to list
4	<code>list.index(obj)</code>  Returns the lowest index in list that obj appears
5	<code>list.insert(index, obj)</code>  Inserts object obj into list at offset index
6	<code>list.pop(obj=list[-1])</code>  Removes and returns last object or obj from list
7	<code>list.remove(obj)</code>  Removes object obj from list
8	<code>list.reverse()</code>  Reverses objects of list in place
9	<code>list.sort([func])</code>  Sorts objects of list, use compare func if given

Python Standard Datatypes

- Tuple
- A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and **cannot be updated**. Tuples can be thought of as **read-only** lists.
 - tuple = ('abcd', 786 , 2.23, 'john', 70.2)
 - list = ['abcd', 786 , 2.23, 'john', 70.2]
 - tuple[2] = 1000 # Invalid syntax with tuple
 - list[2] = 1000 # Valid syntax with list

Python Standard Datatypes

- Dictionary
- consist of key-value pairs.
- A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([])
- Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.
- dict = {}
- dict['one'] = "This is one"
- dict[2] = "This is two"
- tinydict = {'name': 'john', 'code': 6734, 'dept': 'sales', "One": "This is one"}
- print dict['one'] # Prints value for 'one' key **#This is one**
- print dict[2] # Prints value for 2 key **#This is two**
- print tinydict # Prints complete dictionary **# {'dept': 'sales', 'code': 6734, 'name': 'john'}**
- print tinydict.keys() # Prints all the keys **# ['dept', 'code', 'name']**
- print tinydict.values() # Prints all the values **# ['sales', 6734, 'john']**

Dictionary

- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Dictionary

- More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.
- Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed

Dictionary

- If we attempt to access a data item with a key, which is not part of the dictionary, we get an error
- `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`
- `print "dict['Name']: ", dict['Name']`
- `print "dict['Age']: ", dict['Age']`
- `print "dict['Alice']: ", dict['Alice']`

- `dict['Name']: Zara`
- `dict['Age']: 7`
- `dict['Alice']:`
- Traceback (most recent call last):
 - File "test.py", line 4, in <module>
 - `print "dict['Alice']: ", dict['Alice'];`
- `KeyError: 'Alice'`

Updating & Deleting Dictionary

- `dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}`
- `dict['Age'] = 8; # update existing entry`
- `dict['School'] = "DPS School"; # Add new entry`

- `del dict['Name']; # remove entry with key 'Name'`
- `dict.clear(); # remove all entries in dict`
- `del dict ; # delete entire dictionary`

Dictionary Methods

- `cmp(dict1, dict2)` (python 2)
- `len(dict)`
- `str(dict)`
- `type(variable)`
- `dict.clear()`
- `dict.copy()`
- `dict.get(key, default=None)`
- `dict.has_key(key)`(python 2)
- `dict.items()`
- `dict.keys()`
- `dict.setdefault(key, default=None)`
- `dict.update(dict2)`
- `dict.values()`
- `dict.pop(key)`
- `Dict.popitem(LIFO)`
- `dict.fromkeys()`

Decision making

- `if(condition):`

- `if(condition):`

`else:`

- `if(condition):`

- `elif (condition):`

- `else:`

- Nested if

- If else ladder

Loops

- while expression:
 - statement(s)
- for iterating_var in sequence:
 - statements(s)
- For Iterating by Sequence Index, use range()
 - range(start,end,diff)
 - range(start, end)
 - range(end)

Loop Control Statements

- break statement
 - Terminates the loop statement and transfers execution to the statement immediately following the loop.
- continue statement
 - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- pass statement
 - The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Using else Statement with Loops

- If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Operators

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators =
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Operators

- Arithmetic Operators

+

-

*

/ quotient.... $5/2$ 2.5 2 1

% remainder $5\%2$

//: Floor Division: The division of operands where the result is the quotient in which the digits after the decimal point are removed. Eg $9//2=4$, $-9//2=-4$, $9.0//2=4.0$

**** $5**3=====5$ cube, $6**9$, 6 to the power 9**

Operators

- Comparison Operators.. If or while

==

!= or <>

>

<

>=

<=

Operators

- Assignment Operators

=

+=

-=

*=

/=

**=

//=

%=

Operators

- Logical Operators

- and

0	0	0
0	1	0
1	0	0
1	1	1

- or

0	0	0
0	1	1
1	0	1
1	1	1

- Not

0	1
1	0

Operators

- Bitwise Operators

&

|

0 absence of voltage

1 presence of voltage

A----- 65----- 0110 0101--- 1100 1010

A

~

^

<<

>>

0	0	0
0	1	1
1	0	1
1	1	0

Operators

- Membership Operators
- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators
 - in
 - not in

Operators

- Identity Operators
- Identity operators compare the memory locations of two objects. There are two Identity operators
 - is
 - is not

Data Type Conversion

- Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.
- There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

Data Type Conversion

- **int(x [,base])**
 - Converts x to an integer. base specifies the base if x is a string.
- **long(x [,base])(in python 2)**
 - Converts x to a long integer. base specifies the base if x is a string.
- **float(x)**
 - Converts x to a floating-point number.
- **complex(real [,imag])**
 - Creates a complex number.

Data Type Conversion

- **str(x)**
 - Converts object x to a string representation.
- **tuple(s)**
 - Converts s to a tuple.
- **list(s)**
 - Converts s to a list.
- **set(s)**
 - Converts s to a set.
- **dict(d)**
 - Creates a dictionary. d must be a sequence of (key,value) tuples.
- **frozenset(s)**
 - Converts s to a frozen set.

Data Type Conversion

- **ord(x)**
 - Converts a single character to its integer value.
- **hex(x)**
 - Converts an integer to a hexadecimal string.
- **oct(x)**
 - Converts an integer to an octal string.

File Input Output

- Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a **file** object.
- Before you can read or write a file, you have to open it using Python's built-in *open()* function. This function creates a **file** object, which would be utilized to call other support methods associated with it.
- `file object = open(file_name [, access_mode][, buffering])`
- **file_name** – The `file_name` argument is a string value that contains the name of the file that you want to access.
- **access_mode** – The `access_mode` determines the mode in which the file has to be opened, i.e., read, write, append, etc. It could be `r,w,a,rb,wb,ab,r+,w+,a+,rb+,wb+,ab+`
- **buffering** – If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size.

File Input Ouput

- `Filename=open("filename",mode)`
- `Filename.closed`
- `Filename.mode`
- `Filename.name`
- `Filename.close()`
- `Filename.write(string)`
- `Filename.read([countofbytes])`
- `Filename.readlines()`
- `Filename.tell();noofbytes`
- `Filename.seek(noofbytes[,from]);from` can be 0/1/2
- For more functions refer **os module**

File Input Output

'buffer',
'close',
'closed',
'detach',
'encoding',
'errors',
'fileno',
'flush',
'isatty',
'line_buffering',
'mode',
'name',
'newlines',
'read',
'readable',
'readline',
'readlines',
'reconfigure',
'seek',
'seekable',
'tell',
'truncate',
'writable',
'write',
'write_through',
'writelines'

Functions

- 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.

Python Inbuilt Functions

		Built-in Functions		
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code><u>ascii()</u></code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

Abs() (+)

- The abs() method returns the absolute value of the given number
- **abs() Parameters**
- The abs() method takes a single argument:
- **num** - number whose absolute value is to be returned. The number can be:
 - integer
 - floating number
 - complex number
- **Return value from abs()**
- The abs() method returns the absolute value of the given number.
- For integers - integer absolute value is returned
- For floating numbers - floating absolute value is returned
- For complex numbers - magnitude of the number is returned

All()

- The all() method returns True when all elements in the given iterable are true. If not, it returns False.
- **all() Parameters**
- The all() method takes a single parameter:
- **iterable** - any iterable ([list](#), [tuple](#), [dictionary](#), etc.) which contains the elements
- **Return Value from all()**
- The all() method returns:
- **True** - If all elements in an iterable are true
- **False** - If any element in an iterable is false

Truth table for all()

When	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	False
One value is false (others are true)	False
Empty Iterable	True

Any()

- The any() method returns True if any element of an iterable is True. If not, any() returns False.
- **any() Parameters**
- The any() method takes an iterable (list, string, dictionary etc.) in Python.
- **Return Value from any()**
- True if at least one
- element of an
- iterable is true
- False if all elements
- are false or if an
- iterable is empty

When	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	True
One value is false (others are true)	True
Empty Iterable	False

Ascii()

- The `ascii()` method returns a string containing a printable representation of an object. It escapes the non-ASCII characters in the string using `\x`, `\u` or `\U` escapes.
- `ascii(object)` **ascii() Parameters**
- The `ascii()` method takes an object (like: [strings](#), [list](#) etc).
- **Return Value from `ascii()`**
- It returns a string containing printable representation of an object.
- For example, `ö` is changed to `\xf6n`, `v` is changed to `\u221a`
- The non-ASCII characters in the string is escaped using `\x`, `\u` or `\U`.
- <https://rbutterworth.nfshost.com/Tables/common>

Bytes()

- The bytes() method returns a immutable bytes object initialized with the given size and data.
- bytes([source[, encoding[, errors]]])
- **source (Optional)** - source to initialize the array of bytes.
- **encoding (Optional)** - if source is a string, the encoding of the string.
- **errors (Optional)** - if source is a string, the action to take when the encoding conversion fails (Read more: [String encoding](#))
- The **source** parameter can be used to initialize the byte array in the following ways:

Different source parameters

Type	Description
String	Converts the string to bytes using str.encode() Must also provide encoding and optionally errors
Integer	Creates an array of provided size, all initialized to null
Object	Read-only buffer of the object will be used to initialize the byte array
Iterable	Creates an array of size equal to the iterable count and initialized to the iterable elements Must be iterable of integers between $0 \leq x < 256$
No source (arguments)	Creates an array of size 0

- **Return value from bytes()**
- The bytes() method returns a bytes object of the given size and initialization values.

Bytearray()

- The bytearray() method returns a bytearray object which is an array of the given bytes.
- bytearray([source[, encoding[, errors]]])
- The bytearray() method returns a bytearray object which is a mutable (can be modified) sequence of integers in the range $0 \leq x < 256$.
- If you want the immutable version, use [bytes\(\)](#) method.
- **source (Optional)** - source to initialize the array of bytes.
- **encoding (Optional)** - if source is a string, the encoding of the string.
- **errors (Optional)** - if source is a string, the action to take when the encoding conversion fails (Read more: [String encoding](#))
- The **source** parameter can be used to initialize the byte array in the following ways:

Different source parameters

Type	Description
String	Converts the string to bytes using str.encode() Must also provide encoding and optionally errors
Integer	Creates an array of provided size, all initialized to null
Object	Read-only buffer of the object will be used to initialize the byte array
Iterable	Creates an array of size equal to the iterable count and initialized to the iterable elements Must be iterable of integers between $0 \leq x < 256$
No source (arguments)	Creates an array of size 0.

- **Return value from bytearray()**
- The bytearray() method returns an array of bytes of the given size and initialization values.

Bool()

- The bool() method converts a value to Boolean (True or False) using the standard truth testing procedure.
- bool([value])
- It's not mandatory to pass a value to bool(). If you do not pass a value, bool() returns False.
- In general use, bool() takes a single parameter value.
- **Return Value from bool()**
- The bool() returns:
 - False if the *value* is omitted or false
 - True if the *value* is true
- **The following values are considered false in Python:**
- **None**
- **False**
- **Zero of any numeric type. For example, 0, 0.0, 0j**
- **Empty sequence. For example, (), [], "".**
- **Empty mapping. For example, {}**
- **objects of Classes which has __bool__() or __len__() method which returns 0 or False**
- All other values except these values are considered true.

Bin()

- The bin() method converts and returns the binary equivalent string of a given integer. If the parameter isn't an integer, it has to implement __index__() method to return an integer.
- bin(num) **bin() Parameters**
- The bin() method takes a single parameter:
- **num** - an integer number whose binary equivalent is to be calculated.
If not an integer, should implement __index__() method to return an integer.
- **Return value from bin()**
- The bin() method returns the binary string equivalent to the given integer.
- If not specified an integer, it raises a TypeError exception highlighting the type cannot be interpreted as an integer.

Complex()

- The `complex()` method returns a complex number when real and imaginary parts are provided, or it converts a string to a complex number.
- `complex([real[, imag]])` **complex() Parameters**
- In general, the `complex()` method takes two parameters:
- **real** - real part. If *real* is omitted, it defaults to 0.
- **imag** - imaginary part. If *imag* is omitted, it default to 0.
- If the first parameter passed to this method is a string, it will be interpreted as a complex number. In this case, second parameter shouldn't be passed.
- **Return Value from complex()**
- As suggested by the name, the `complex()` method returns a complex number.
- If the string passed to this method is not a valid complex number, `ValueError` exception is raised.

Chr()

- The chr() method returns a character (a string) from an integer (represents unicode code point of the character).
- chr(i) **chr() Parameters**
- The chr() method takes a single parameter, an integer *i*.
- The valid range of the integer is from 0 through 1,114,111.
- **Return Value from chr()**
- The chr() returns:
 - a character (a string) whose Unicode code point is the integer *i*
 - If the integer *i* is outside the range, ValueError will be raised.

Dir()

- The dir() method tries to return a list of valid attributes of the object.
- dir([object]) **dir() Parameters**
- The dir() takes maximum of one object.
- **object** (optional) - dir() attempts to return all attributes of this object.
- **Return Value from dir()**
- The dir() tries to return a list of valid attributes of the object.
- If the object has __dir__() method, the method will be called and must return the list of attributes.
- If the object doesn't have __dir__() method, this method tries to find information from the __dict__ attribute (if defined), and from type object. In this case, the list returned from dir() may not be complete.
- If object is not passed to the dir() method, it returns the list of names in the current local scope.

Divmod()

- The divmod() method takes two numbers and returns a pair of numbers (a tuple) consisting of their quotient and remainder.
- `divmod(x, y)` **divmod() Parameters**
- The divmod() takes two parameters:
- **x** - a non-complex number (numerator)
- **y** - a non-complex number (denominator)
- **Return Value from divmod()**
- The divmod() returns
- (q, r) - a pair of numbers (a [tuple](#)) consisting of quotient *q* and remainder *r*
- If x and y are integers, the return value from divmod() is same as (a // b, x % y).
- If either x or y is a float, the result is (q, x%y). Here, *q* is the whole part of the quotient.

Dict()

- The dict() constructor creates a dictionary in Python.
- The dict() doesn't return any value (returns None).

Enumerate()

- The enumerate() method adds counter to an iterable and returns it (the enumerate object).
- enumerate(iterable, start=0) **enumerate() Parameters**
- The enumerate() method takes two parameters:
- **iterable** - a sequence, an iterator, or objects that supports iteration
- **start** (optional) - enumerate() starts counting from this number. If *start* is omitted, 0 is taken as start.
- **Return Value from enumerate()**
- The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.
- You can convert enumerate objects to list and tuple using [list\(\)](#) and [tuple\(\)](#) method respectively.

Eval()

- The eval() function evaluates the specified expression, if the expression is a legal Python statement, it will be executed.
- `x = 'print(55)'`
`eval(x)`

Exec()

- The exec() function executes the specified Python code.
- The exec() function accepts large blocks of code, unlike the eval() function which only accepts a single expression
- `x = 'name =`
`"John"\nprint(name)'`
`exec(x)`

Float()

- The float(x) method returns a floating point number from a number or a string.
- **x (Optional)** - number or string that needs to be converted to floating point number
If it's a string, the string should contain decimal points
- The float() method returns:
- Equivalent floating point number if an argument is passed
- 0.0 if no arguments passed
- OverflowError exception if the argument is outside the range of Python float

Format()

- The built-in format() method returns a formatted representation of the given value controlled by the format specifier.
- format(value[, format_spec])
- **value** - value that needs to be formatted
- **format_spec** - The specification on how the value should be formatted.
- The format() method returns a formatted representation of a given value specified by the format specifier.
- [[fill]align][sign][#][0][width][,][.precision][type] where, the options are
- fill ::= any character
- align ::= "<" | ">" | "=" | "^"
- sign ::= "+" | "-" | " "
- width ::= integer
- precision ::= integer
- type ::= "b" | "d" | "e" | "E" | "f" | "F" | "o" | | "x" | "X"

frozenset()

- The frozenset() method returns an immutable frozenset object initialized with elements from the given iterable.
- Frozen set is just an immutable version of a Python **set** object.
- So lets see set first

set

- 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.
- 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.
- Creating an empty set is a bit tricky.
- Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the `set()` function without any argument.

set

- Sets are mutable. But since they are unordered, indexing have no meaning.
- We cannot access or change an element of set using indexing or slicing. Set does not support it.
- We can add single element using the `add()` method and multiple elements using the `update()` method. The `update()` method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.
- A particular item can be removed from set using methods, `discard()` and `remove()`.
- The only difference between the two is that, while using `discard()` if the item does not exist in the set, it remains unchanged. But `remove()` will raise an error in such condition.
- Similarly, we can remove and return an item using the `pop()` method.
- Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.
- We can also remove all items from a set using `clear()`.

Python Set Operations

- Union of A and B is a set of all elements from both sets. Union is performed using $|$ operator. Same can be accomplished using the method `union()`.
- Intersection of A and B is a set of elements that are common in both sets. Intersection is performed using $&$ operator. Same can be accomplished using the method `intersection()`.
- Difference of A and B ($A - B$) is a set of elements that are only in A but not in B . Similarly, $B - A$ is a set of element in B but not in A . Difference is performed using $-$ operator. Same can be accomplished using the method `difference()`.
- Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both. Symmetric difference is performed using $^$ operator. Same can be accomplished using the method `symmetric_difference()`.
- We can test if an item exists in a set or not, using the keyword `in`.
- Using a for loop, we can iterate though each item in a set.

Different Python Set Methods

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns the difference of two or more sets as a new set
<code>difference_update()</code>	Removes all elements of another set from this set
<code>discard()</code>	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
<code>intersection()</code>	Returns the intersection of two sets as a new set
<code>intersection_update()</code>	Updates the set with the intersection of itself and another
<code>isdisjoint()</code>	Returns <code>True</code> if two sets have a null intersection
<code>issubset()</code>	Returns <code>True</code> if another set contains this set
<code>issuperset()</code>	Returns <code>True</code> if this set contains another set
<code>pop()</code>	Removes and returns an arbitrary set element. Raise <code>KeyError</code> if the set is empty
<code>remove()</code>	Removes an element from the set. If the element is not a member, raise a <code>KeyError</code>
<code>symmetric_difference()</code>	Returns the symmetric difference of two sets as a new set
<code>symmetric_difference_update()</code>	Updates a set with the symmetric difference of itself and another
<code>union()</code>	Returns the union of sets in a new set
<code>update()</code>	Updates the set with the union of itself and others

Frozenset()

- While elements of a set can be modified at any time, elements of frozen set remains the same after creation.
- Due to this, frozen sets can be used as key in Dictionary or as element of another set. But like sets, it is not ordered (the elements can be set at any index).
- `frozenset([iterable])`
- The `frozenset()` method optionally takes a single parameter:
- **iterable (Optional)** - the iterable which contains elements to initialize the frozenset with.
Iterable can be set, dictionary, tuple, etc.
- The `frozenset()` method returns an immutable frozenset initialized with elements from the given iterable.
- Like normal sets, frozenset can also perform different operations like union, intersection, etc. Check using `dir` function.

Help()

- The help() method calls the built-in Python help system.
- help(object)
- The help() method takes maximum of one parameter.
- **object** (optional) - you want to generate the help of the given object
- The help() method is used for interactive use. It's recommended to try it in your interpreter when you need help to write Python program and use Python modules.
- If string is passed as an argument, name of a module, function, class, method, keyword, or documentation topic, and a help page is printed.
- If string is passed as an argument, the given string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed.
- If no argument is passed, Python's help utility (interactive help system) starts on the console. Use ctrl+d to quit or enter quit

Hex()

- The hex() function converts an integer number to the corresponding hexadecimal string.
- The hex() function takes a single argument.
- **x** - integer number (int object or it has to define __index__() method that returns an integer)
- The hex() function converts an integer to the corresponding hexadecimal number in string form and returns it.
- The returned hexadecimal string starts with prefix "0x" indicating it's in hexadecimal form.
- If you need to find hexadecimal representation of a float, you need to use float.hex() method.

Id()

- The id() function returns identity (unique integer) of an object.
- The id() function takes a single parameter *object*.
- The id() function returns identity of the object. This is an integer which is unique for the given object and remains constant during its lifetime.
- It's important to note that everything in Python is an object, even numbers and Classes.
- Hence, integer **5** has a unique id. The id of the integer 5 remains constant during the lifetime; doesn't matter if it is saved in a variable or not. Similar is the case for float **5.5** and other objects.

Input()

- The input() method reads a line from input, converts into a string and returns it.
- The input() method takes a single optional argument:
- **prompt (Optional)** - a string that is written to standard output (usually screen) without trailing newline
- The input() method reads a line from input (usually user), converts the line into a string by removing the trailing newline, and returns it.
- If EOF is read, it raises an EOFError exception.

Int()

- The int() method returns an integer object from any number or string.
- int(x=0, base=10)
- The int() method takes two arguments:
- **x** - Number or string to be converted to integer object. Default argument is **zero**.
- **base** - Base of the number in **x**. Can be 0 (code literal) or 2-36.
- The int() method returns:
- an integer object from the given number or string, treats default base as 10
- (No parameters) returns 0
- (If base given) treats the string in the given base (0, 2, 8, 10, 16)

Iter() and next()

- The iter() method returns an iterator for the given object.
- The iter() method creates an object which can be iterated one element at a time.
- These objects are useful when coupled with loops like for loop, while loop.
- **object** - object whose iterator has to be created (can be sets, tuples, etc.)
- The iter() method returns iterator object for the given object that loops through each element in the object.
- The next() function returns the next item from the iterator.
- next(iterator, default)
- **iterator** - next() retrieves next item from the *iterator*
- **default** (optional) - this value is returned if the iterator is exhausted (no items left)
- The next() returns the next item from the iterator.
- If the iterator is exhausted, it returns default value (if provided).
- If the *default* parameter is omitted and *iterator* is exhausted, it raises StopIteration exception

List()

- The list() constructor creates a list in Python.
- Python list() constructor takes a single argument
- **iterable (Optional)** - an object that could be a sequence (string, tuples) or collection (set, dictionary) or iterator object
- The list() constructor returns a mutable sequence list of elements.
- If no parameters are passed, it creates an empty list
- If iterable is passed as parameter, it creates a list of elements in the iterable

Len()

- The len() function returns the number of items (length) in an object.
- len(s)
- **s** - a sequence (string, bytes, tuple, list, or range) or a collection (dictionary, set or frozen set)
- The len() function returns the number of items of an object.
- Failing to pass an argument or passing an invalid argument will raise a TypeError exception.

Max()

- The `max()` function returns the item with the highest value, or the item with the highest value in an iterable.
- If the values are strings, an alphabetically comparison is done.
- Return the name with the highest value, ordered alphabetically `max(n1, n2, n3, ...)`
- Or: `max(iterable)`

Min()

- The `min()` function returns the item with the lowest value, or the item with the lowest value in an iterable.
- If the values are strings, an alphabetically comparison is done. Return the name with the lowest value, ordered alphabetically
- `in(n1, n2, n3, ...)`
- Or: `min(iterable)`

Oct()

- The `oct()` function converts an integer into an octal string.
- Octal strings in Python are prefixed with `0o`.

Pow()

- The `pow()` function returns the value of x to the power of y (x^y).
- If a third parameter is present, it returns x to the power of y , modulus z .
- `pow(x, y, z)`
- `x = pow(4, 3)`

Print()

- The print() function prints the specified message to the screen, or other standard output device.
- The message can be a string, or any other object, the object will be converted into a string before written to the screen.
- `print(object(s), separator=separator, end=end, file=file, flush=flush)`
- `print("Hello", "how are you?", sep="---")`

Parameter	Description
<i>object(s)</i>	Any object, and as many as you like. Will be converted to string before printed
<i>sep='separator'</i>	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
<i>end='end'</i>	Optional. Specify what to print at the end. Default is '\n' (line feed)
<i>file</i>	Optional. An object with a write method. Default is sys.stdout
<i>flush</i>	Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

Range()

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

Parameter	Description
<i>start</i>	Optional. An integer number specifying at which position to start. Default is 0
<i>stop</i>	Required. An integer number specifying at which position to end.
<i>step</i>	Optional. An integer number specifying the incrementation. Default is 1

Reversed()

- The `reversed()` function returns a reversed **iterator object**.
- `reversed(sequence)`
- Ex
- ```
alph = ["a", "b", "c", "d"]
ralph = reversed(alph)
for x in ralph:
 print(x)
```
- The `list.reverse()` method reverses a List.

# Round()

- The round() function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.
- The default number of decimals is 0, meaning that the function will return the nearest integer.
- round(*number, digits*)
- x = round(5.76543, 2)

# Slice()

- The slice() function returns a slice object.
- A slice object is used to specify how to slice a sequence. You can specify where to start the slicing, and where to end. You can also specify the step, which allows you to e.g. slice only every other item.
- *slice(start, end, step)*

| Parameter    | Description                                                                                 |
|--------------|---------------------------------------------------------------------------------------------|
| <i>start</i> | Optional. An integer number specifying at which position to start the slicing. Default is 0 |
| <i>end</i>   | An integer number specifying at which position to end the slicing                           |
| <i>step</i>  | Optional. An integer number specifying the step of the slicing. Default is 1                |

# Sorted()

- The sorted() function returns a sorted list of the specified iterable object.
- You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically.
- **Note:** You cannot sort a list that contains BOTH string values AND numeric values.
- `sorted(iterable, key=key, reverse=reverse)`

| Parameter       | Description                                                                                 |
|-----------------|---------------------------------------------------------------------------------------------|
| <i>iterable</i> | Required. The sequence to sort, list, dictionary, tuple etc.                                |
| <i>key</i>      | Optional. A Function to execute to decide the order. Default is None                        |
| <i>reverse</i>  | Optional. A Boolean. False will sort ascending, True will sort descending. Default is False |

# Str()

- The str() function converts the specified value into a string.
- `x = str(3.5)`



# Sum()

- The `sum()` function returns a number, the sum of all items in an iterable.
- `sum(iterable, start)`

| Parameter       | Description                                         |
|-----------------|-----------------------------------------------------|
| <i>iterable</i> | Required. The sequence to sum                       |
| <i>start</i>    | Optional. A value that is added to the return value |

- `a = (1, 2, 3, 4, 5)`  
`x = sum(a, 7)`
- 22

# Tuple()

The `tuple()` function creates a tuple object.

**Note:** You cannot change or remove items in a tuple.

## Syntax

```
tuple(iterable)
```

## Parameter Values

| Parameter       | Description                                            |
|-----------------|--------------------------------------------------------|
| <i>iterable</i> | Required. A sequence, collection or an iterator object |

## Example

Create a tuple containing fruit names:

```
x = tuple(('apple', 'banana', 'cherry'))
```

# Type()

- The type() function returns the type of the specified object
- Example:
- `a = ('apple', 'banana', 'cherry')`  
`b = "Hello World"`  
`c = 33`

```
x = type(a)
y = type(b)
z = type(c)
```

# Zip()

- The `zip()` function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.
- `zip(iterator1, iterator2, iterator3 ...)`
- `a = ("John", "Charles", "Mike")`  
`b = ("Jenny", "Christy", "Monica")`

```
x = zip(a, b)
```

# Functions

- 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*.
- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- 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`.

# Functions

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- 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`.
- By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

# Functions

- 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.
- `Def my_add(x,y):`
  - “this function adds 2 nos”
  - Return `x+y`
- `C=my_add(a,b)#pass by value`
- `d=my_add(10,20)`
- Passbyreference can be done using list

# Functions-Required Arguments

- 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.
- Eg `def printhi(name):`
  - Print “Hello, ”,name
- `Prinhi()`
- `TypeError: printhi() takes exactly 1 argument (0 given)`



# Functions-Keyword Arguments

- 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.
- `def printinfo( name, age ):`
  - `print "Name: ", name`
  - `print "Age ", age`
  - `return;`
- `printinfo( age=50, name="miki" )`

# Functions-Default Arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.
- `def printinfo( name, age = 18 ):`
  - "This prints a passed info into this function"
  - `print "Name: ", name`
  - `print "Age ", age`
  - `return;`
- `printinfo( age=50, name="miki" )`
- `printinfo( name="miki" )`

# Functions-Variable length Arguments

- 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.
- `def functionname([formal_args,] *var_args_tuple )`
- `def printinfo( arg1, *vartuple ):`
  - `print "Output is: "`
  - `print arg1`
  - `for var in vartuple:`
    - `print var`
  - `return;`
- `printinfo( 10 )`
- `printinfo( 70, 60, 50 )`

# Anonymous Function

- 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.
- 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.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- 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.

# Anonymous Function

- `sum = lambda arg1, arg2: arg1 + arg2;`
- `print "Value of total : ", sum( 10, 20 )`
- `print "Value of total : ", sum( 20, 20 )`
- Value of total : 30
- Value of total : 40

# Lambda function

- `result = lambda no_1, no_2 : no_1 + no_2`
- `print (result(10,20))`
  
- `result = lambda : "welcome"`
- `print(result())`
  
- `result = lambda name : ""`
- `print(result("XYZ"))`

# If else in lambda

- `result = lambda no : "even" if (no%2==0) else "odd"`
- `print(result(25))`
- `result = lambda no : "even" if (no%2==0) else "odd"`
- `print(result(24))`
  
- `location = "IN"`
- `result = lambda : "Namaste" if (location == "IN") else "Hello"`
- `print(result())`

# Nested lambda

- `def action(x):`
- `# Make and return function, remember x`
- `return (lambda newx: x + newx)`
- `print(action(100))`
  
- `def action(x):`
- `– x=x+x#200`
- `– Z=lambda newx: x + newx                   #250`
- `# Make and return function, remember x`
- `return (2*Z)#500`
- `y=(action(100))`
- `print(y(50))`



# Map function

- **map()** function returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)
- `map(fun, iter)`
- **fun** : It is a function to which map passes each element of given iterable.  
**iter** : It is a iterable which is to be mapped.
- **NOTE** : You can pass one or more iterable to the `map()` function.
- Returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)
- **NOTE** : The returned value from `map()` (map object) then can be passed to functions like `list()` (to create a list), `set()` (to create a set) .

# Map function

- `# Return double of n`
- `def addition(n):`
- `return n + n`
- 
- `# We double all numbers using map()`
- `numbers = (1, 2, 3, 4)`
- `result = map(addition, numbers)`
- `print(list(result))`

# Map with lambda

- `# Double all numbers using map and lambda`
- `numbers = (1, 2, 3, 4)`
- `result = map(lambda x: x + x, numbers)`
- `print(list(result))`
  
- `# Add two lists using map and lambda`
- `numbers1 = [1, 2, 3]`
- `numbers2 = [4, 5, 6]`
- `result = map(lambda x, y: x + y, numbers1, numbers2)`
- `print(list(result))`
  
- `# List of strings`
- `l = ['sat', 'bat', 'cat', 'mat']`
- `# map() can listify the list of strings individually`
- `test = list(map(list, l))`
- `print(test)`

# Filter function

- **filter() in python**
- The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
- **syntax:** filter(function, sequence)
- **function:** function that tests if each element of a sequence true or not.
- **sequence:** sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.
- **Retruns:** returns an iterator that is already filtered.

# Filter function

- # function that filters vowels
- def fun(variable):
- letters = ['a', 'e', 'i', 'o', 'u']
- if (variable in letters):
- return True
- else:
- return False
- # sequence
- sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
- # using filter function
- filtered = filter(fun, sequence)
- print('The filtered letters are:')
- for s in filtered:
- print(s)

# Filter with lambda

- # a list contains both even and odd numbers.
- seq = [0, 1, 2, 3, 5, 8, 13]
- # result contains odd numbers of the list
- result = filter(lambda x: x % 2 == 1, seq)
- print(list(result))
- # result contains even numbers of the list
- result = filter(lambda x: x % 2 == 0, seq)
- print(list(result))

# methods

- Float
- Int
- Char
- Str
- List
- Dict..

# Modules(Libraries)

- A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.



# Modules(Libraries)

- The Python code for a module named *aname* normally resides in a file named *aname.py*.
- Eg greetings.py
  - def greeting\_func( var ):
    - print "Hello : ", var
    - return
- import greetings
- greetings.greeting\_func("Zara")
- Hello : Zara

# Modules(Libraries)

- When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module.
- When you import a module, the Python interpreter searches for the module in the following sequences –
  - The current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
  - If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.
  - The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.
- A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

# Modules(Libraries)

- The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.
  - import sys
  - sys.path.append('/path/to/module')
- Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).
- A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

# Modules(Libraries)

- Python's *from* statement lets you import specific attributes from a module into the current namespace.
- Eg
- `from fib import fibonacci`
- This statement does not import the entire module `fib` into the current namespace; it just introduces the item `fibonacci` from the module `fib` into the global symbol table of the importing module.

# Modules(Libraries) Functions

- `Dir(module_name)`
- `Globals()`
- `Locals()`
- `Reload(module)`

# methods

- Dir modules also helps to learn methods associated with each data type

# Packages(group of libraries of same type)

- A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and sub-subpackages, and so on.
- Consider a file *Pots.py* available in *Phone* directory.
- Eg
- `def Pots(): print "I'm Pots Phone"`
- Similar way, we have another two files having different functions with the same name as above –
- *Phone/Isdn.py* file having function `Isdn()`
- *Phone/G3.py* file having function `G3()`
- Now, create one more file `__init__.py` in *Phone* directory –
- *Phone/\_\_init\_\_.py*
- To make all of your functions available when you've imported *Phone*, you need to put explicit import statements in `__init__.py` as follows for python 2–
- `from Pots import Pots`
- `from Isdn import Isdn`
- `from G3 import G3`

# Packages

- After you add these lines to `__init__.py`, you have all of these classes available when you import the Phone package.
- `import Phone`
- `Phone.Pots()`
- `Phone.Isdn()`
- `Phone.G3()`
- When the above code is executed, it produces the following result –
- I'm Pots Phone
- I'm 3G Phone
- I'm ISDN Phone
- In the above example, we have taken example of a single functions in each file, but you can keep multiple functions in your files. You can also define different Python classes in those files and then you can create your packages out of those classes.



# Numbers

- Numerical Data Types:
  - Int
  - Float
  - Long
  - complex

# Numbers Examples

Here are some examples of numbers

| int    | long                   | float      | complex    |
|--------|------------------------|------------|------------|
| 10     | 51924361L              | 0.0        | 3.14j      |
| 100    | -0x19323L              | 15.20      | 45.j       |
| -786   | 0122L                  | -21.9      | 9.322e-36j |
| 080    | 0xDEFA BCECBDAECBFBAEL | 32.3+e18   | .876j      |
| -0490  | 535633629843L          | -90.       | -.6545+0J  |
| -0x260 | -052318172735L         | -32.54e100 | 3e+26J     |
| 0x69   | -4721885298529L        | 70.2-E12   | 4.53e-7j   |

# Numbers Type Conversion

- Type **int(x)** to convert x to a plain integer.
- Type **long(x)** to convert x to a long integer.
- Type **float(x)** to convert x to a floating-point number.
- Type **complex(x)** to convert x to a complex number with real part x and imaginary part zero.
- Type **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

# Math Module

- `abs(x)`
- `Ceil(x)`
- `Cmp(x,y)`
- `Exp(x)`
- `Fabs(x)`
- `Floor(x)`
- `log(x)`
- `log10(x)`
- `Max(x,y)`
- `Min(x,y)`
- `Modf(x)`
- `Pow(x,y)`
- `Round(x[,n])`
- `Sqrt(x)`
- `Acos(x)`
- `asin(x)`
- `asin(x)`
- `atan(x)`
- `cos(x)`
- `hypot(x)`
- `sin(x)`
- `tan(x)`
- `degrees(x)`
- `radians(x)`

Pi and e are mathematical constants in math module

# Math module

| Name of the function   | Description                                                                            | Example                                                                                                                                            |
|------------------------|----------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ceil( x )</code> | It returns the smallest integer not less than $x$ , where $x$ is a numeric expression. | <code>math.ceil(-45.17)</code><br><b>-45.0</b><br><code>math.ceil(100.12)</code><br><b>101.0</b><br><code>math.ceil(100.72)</code><br><b>101.0</b> |

|                           |                                                                                   |                                                                                                                                                                  |
|---------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>degrees( x )</code> | It converts angle $x$ from radians to degrees, where $x$ must be a numeric value. | <code>math.degrees(3)</code><br><b>171.887338539</b><br><code>math.degrees(-3)</code><br><b>-171.887338539</b><br><code>math.degrees(0)</code><br><b>0.0</b>     |
| <code>radians(x)</code>   | It converts angle $x$ from degrees to radians, where $x$ must be a numeric value. | <code>math.radians(3)</code><br><b>0.0523598775598</b><br><code>math.radians(-3)</code><br><b>-0.0523598775598</b><br><code>math.radians(0)</code><br><b>0.0</b> |

# Math module

|                          |                                                                                          |                                                                                                                                                                                     |
|--------------------------|------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>floor( x )</code>  | It returns the largest integer not greater than $x$ , where $x$ is a numeric expression. | <code>math.floor(-45.17)</code><br><b>-46.0</b><br><code>math.floor(100.12)</code><br><b>100.0</b><br><code>math.floor(100.72)</code><br><b>100.0</b>                               |
| <code>fabs( x )</code>   | It returns the absolute value of $x$ , where $x$ is a numeric value.                     | <code>math.fabs(-45.17)</code><br><b>45.17</b><br><code>math.fabs(100.12)</code><br><b>100.12</b><br><code>math.fabs(100.72)</code><br><b>100.72</b>                                |
| <code>exp( x )</code>    | It returns exponential of $x$ : $e^x$ , where $x$ is a numeric expression.               | <code>math.exp(-45.17)</code><br><b>2.41500621326e-20</b><br><code>math.exp(100.12)</code><br><b>3.03084361407e+43</b><br><code>math.exp(100.72)</code><br><b>5.52255713025e+43</b> |
| <code>log( x )</code>    | It returns natural logarithm of $x$ , for $x > 0$ , where $x$ is a numeric expression.   | <code>math.log(100.12)</code><br><b>4.60636946656</b><br><code>math.log(100.72)</code><br><b>4.61234438974</b>                                                                      |
| <code>log10( x )</code>  | It returns base-10 logarithm of $x$ for $x > 0$ , where $x$ is a numeric expression.     | <code>math.log10(100.12)</code><br><b>2.00052084094</b><br><code>math.log10(100.72)</code><br><b>2.0031157171</b>                                                                   |
| <code>pow( x, y )</code> | It returns the value of $x^y$ , where $x$ and $y$ are numeric expressions.               | <code>math.pow(100, 2)</code><br><b>10000.0</b><br><code>math.pow(100, -2)</code>                                                                                                   |

# Math module

|                        |                                                                                                |                                                                                                                                                                                                  |
|------------------------|------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sqrt (x )</code> | It returns the square root of $x$ for $x > 0$ , where $x$ is a numeric expression.             | <code>math.sqrt(100)</code><br><b>10.0</b><br><code>math.sqrt(7)</code><br><b>2.64575131106</b>                                                                                                  |
| <code>cos (x)</code>   | It returns the cosine of $x$ in radians, <i>where <math>x</math> is a numeric expression</i>   | <code>math.cos(3)</code><br><b>-0.9899924966</b><br><code>math.cos(-3)</code><br><b>-0.9899924966</b><br><code>math.cos(0)</code><br><b>1.0</b><br><code>math.cos(math.pi)</code><br><b>-1.0</b> |
| <code>sin (x)</code>   | It returns the sine of $x$ , in radians, <i>where <math>x</math> must be a numeric value.</i>  | <code>math.sin(3)</code><br><b>0.14112000806</b><br><code>math.sin(-3)</code><br><b>-0.14112000806</b><br><code>math.sin(0)</code><br><b>0.0</b>                                                 |
| <code>tan (x)</code>   | It returns the tangent of $x$ in radians, <i>where <math>x</math> must be a numeric value.</i> | <code>math.tan(3)</code><br><b>-0.142546543074</b><br><code>math.tan(-3)</code><br><b>0.142546543074</b><br><code>math.tan(0)</code><br><b>0.0</b>                                               |

# Built in arithmetic functions

| Name                              | Description                                                                                                                                                                                              | Example                                                                           |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>abs (x)</code>              | It returns distance between <i>x</i> and zero, where <i>x</i> is a numeric expression.                                                                                                                   | <pre>&gt;&gt;&gt;abs(-45) 45 &gt;&gt;&gt;abs(119L) 119</pre>                      |
| <code>max( x, y, z, .... )</code> | It returns the largest of its arguments: where <i>x</i> , <i>y</i> and <i>z</i> are numeric variable/ expression.                                                                                        | <pre>&gt;&gt;&gt;max(80, 100, 1000) 1000 &gt;&gt;&gt;max(-80, -20, -10) -10</pre> |
| <code>min( x, y, z, .... )</code> | It returns the smallest of its arguments; where <i>x</i> , <i>y</i> , and <i>z</i> are numeric variable/ expression.                                                                                     | <pre>&gt;&gt;&gt; min(80, 100, 1000) 80 &gt;&gt;&gt; min(-80, -20, -10) -80</pre> |
| <code>cmp( x, y )</code>          | It returns the sign of the difference of two numbers: -1 if <i>x</i> < <i>y</i> , 0 if <i>x</i> == <i>y</i> , or 1 if <i>x</i> > <i>y</i> , where <i>x</i> and <i>y</i> are numeric variable/expression. | <pre>&gt;&gt;&gt;cmp(80, 100) -1 &gt;&gt;&gt;cmp(180, 100) 1</pre>                |



# Built in arithmetic functions

|                               |                                                                                                                                                                                        |                                                                                                                             |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <code>divmod (x,y )</code>    | Returns both quotient and remainder by division through a tuple, when x is divided by y; where x & y are variable/ expression.                                                         | <pre>&gt;&gt;&gt; divmod (14,5) (2,4) &gt;&gt;&gt; divmod (2.7, 1.5) (1.0, 1.20000)</pre>                                   |
| <code>len (s)</code>          | Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).                                                | <pre>&gt;&gt;&gt; a= [1,2,3] &gt;&gt;&gt;len (a) 3 &gt;&gt;&gt; b= 'Hello' &gt;&gt;&gt; len (b) 5</pre>                     |
| <code>round( x [, n] )</code> | <p>It returns float x rounded to n digits from the decimal point, <i>where x and n are numeric expressions.</i></p> <p>If n is not provided then x is rounded to 0 decimal digits.</p> | <pre>&gt;&gt;&gt;round(80.23456, 2) 80.23 &gt;&gt;&gt;round(-100.000056, 3) -100.0 &gt;&gt;&gt; round (80.23456) 80.0</pre> |

# Random Module

- `Choice(seq)`
- `Randrange([start,]stop[,step])`
- `Random()`
- `Seed([x])`
- `Shuffle(list)`
- `Uniform(x,y)`

# Random Module

| Name of the function                           | Description                                                        | Example                                                                                              |
|------------------------------------------------|--------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>random ( )</code>                        | It returns a random float $x$ , such that $0 \leq x < 1$           | <pre>&gt;&gt;&gt;random.random ( ) 0.281954791393 &gt;&gt;&gt;random.random ( ) 0.309090465205</pre> |
| <code>randint (a, b)</code>                    | It returns a int $x$ between $a$ & $b$ such that $a \leq x \leq b$ | <pre>&gt;&gt;&gt; random.randint (1,10) 5 &gt;&gt;&gt; random.randint (-2,20) -1</pre>               |
| <code>uniform (a,b)</code>                     | It returns a floating point number $x$ , such that $a \leq x < b$  | <pre>&gt;&gt;&gt;random.uniform (5, 10) 5.52615217015</pre>                                          |
| <code>randrange ([start,] stop [,step])</code> | It returns a random item from the given range                      | <pre>&gt;&gt;&gt;random.randrange(100 ,1000,3) 150</pre>                                             |

# Matplotlib

- 3<sup>rd</sup> party library
- Pip install matplotlib
- `matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- Generating visualizations with pyplot is very quick:
  - `import matplotlib.pyplot as plt`
  - `plt.plot([1, 2, 3, 4])`
  - `plt.ylabel('some numbers')`
  - `plt.show()`

# Exceptions

- 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.

# Exceptions

- For some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.
  - try: You do your operations here; .....
  - except *ExceptionI*: If there is ExceptionI, then execute this block.
  - except *ExceptionII,Argument*: If there is ExceptionII, then execute this block. ....Argument if written holds error string, the error number, and an error location.
  - Except exception1,exception2,.....exception n:If there is any exception from the given exception list, then execute this block.
  - else: If there is no exception then execute this block.
  - Finally: this would be executed always

# Standard Exceptions

- **Exception**
  - Base class for all exceptions
- **ZeroDivisionError**
  - Raised when division or modulo by zero takes place for all numeric types
- **OverflowError**
  - Raised when a calculation exceeds maximum limit for a numeric type.
- **FloatingPointError**
  - Raised when a floating point calculation fails.
- **StandardError**
  - Base class for all built-in exceptions except StopIteration and SystemExit.
- **StopIteration**
  - Raised when the next() method of an iterator does not point to any object.
- **SystemExit**
  - Raised by the sys.exit() function.
- **ArithmeticError**
  - Base class for all errors that occur for numeric calculation.
- **AttributeError**
  - Raised in case of failure of attribute reference or assignment.
- **EOFError**
  - Raised when there is no input from either the raw\_input() or input() function and the end of file is reached.

# Standard Exceptions

- **AssertionError**
  - Raised in case of failure of the Assert statement.
- **ImportError**
  - Raised when an import statement fails.
- **KeyboardInterrupt**
  - Raised when the user interrupts program execution, usually by pressing Ctrl+c.
- **LookupError**
  - Base class for all lookup errors.
- **IndexError**
  - Raised when an index is not found in a sequence.
- **KeyError**
  - Raised when the specified key is not found in the dictionary.
- **NameError**
  - Raised when an identifier is not found in the local or global namespace.
- **UnboundLocalError**
  - Raised when trying to access a local variable in a function or method but no value has been assigned to it.
- **EnvironmentError**
  - Base class for all exceptions that occur outside the Python environment.
- **IOError**
  - Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.



# Standard Exceptions

- **OSError**
  - Raised for operating system-related errors.
- **SyntaxError**
  - Raised when there is an error in Python syntax.
- **IndentationError**
  - Raised when indentation is not specified properly.
- **SystemError**
  - Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
- **SystemExit**
  - Raised when Python interpreter is quit by using the `sys.exit()` function. If not handled in the code, causes the interpreter to exit.
- **TypeError**
  - Raised when an operation or function is attempted that is invalid for the specified data type.
- **ValueError**
  - Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
- **RuntimeError**
  - Raised when a generated error does not fall into any category.
- **NotImplementedError**
  - Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

# Handling Exceptions

```
try:
```

```
 print 1/0
```

```
except ZeroDivisionError:
```

```
 print "Your answer is infinite"
```

# Handling Exceptions

try:

```
number = int(raw_input("Enter a number:"))
```

except ValueError:

```
print "Err.. numbers only"
```

else:

```
print "you entered number "
```

# Handling Exceptions

try:

```
num1, num2 =(input("Enter two numbers, separated by a comma : "))
result = num1 / num2
print("Result is", result)
```

except ZeroDivisionError:

```
print("Division by zero is error !!")
```

except SyntaxError:

```
print("Comma is missing. Enter numbers separated by comma like this 1, 2")
```

except:

```
print("Wrong input")
```

else:

```
print("Code executed succesfully! No exceptions")
```

finally:

```
print("Bye Bye!")
```

# Handling Exceptions

```
def kelvin_to_celcius(var):
 try:
 return int(var)+273.15
 except ValueError, Argument:
 print "Value Error: The argument does not contain numbers\n"
 return Argument

print kelvin_to_celcius("xyz");
```

# Exception as

- Python exception messages can be captured and printed using **as**
- try:
  - num1 = int(input("Enter number 1:"))
  - num2 = int(input("Enter number 2:"))
  - result = num1 / num2
  - print("Result is", result)
- except ZeroDivisionError as e:
  - print("Division by zero is error !!")
  - print(e)

# with

- **with** statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams.

- 1) without using with statement
- `file = open('file_path', 'w')`
- `file.write('hello world !')`
- `file.close()`

- # 2) without using with statement
- `file = open('file_path', 'w')`
- `try:`
- `file.write('hello world')`
- `finally:`
- `file.close()`

- # using with statement
- `with open('file_path', 'w') as file:`
- `file.write('hello world !')`
- Notice there is no need to call `file.close()` when using with statement. The with statement itself ensures proper acquisition and release of resources.
- An exception during the `file.write()` call in the first implementation can prevent the file from closing properly which may introduce several bugs in the code
- The second approach in the above example takes care of all the exceptions but using the with statement makes the code compact and much more readable.
- Thus, with statement helps avoiding bugs and leaks by ensuring that a resource is properly released when the code using the resource is completely executed. The with statement is popularly used with file streams, as shown above and with Locks, sockets, subprocesses and telnets etc.



# Raising Exceptions

- To raise your exceptions from your own methods you need to use raise keyword
  - `raise ExceptionClass("Your argument")`

# Raising Exceptions

```
def enterage(age):
 if age < 0:
 raise ValueError("Only positive integers are allowed")
 if age % 2 == 0:
 return "even"
 else:
 return "odd"

try:
 num = int(input("Enter your age: "))
 ans=enterage(num)
except ValueError:
 print("Only positive integers are allowed")
except:
 print("something is wrong")
else:
 print ("Your age is:",ans)
finally:
 print ("Age is just a number! Enjoy your life!")
```

# Using exceptions objects

- to access exception object in exception handler code, You can use the following code to assign exception object to a variable:
- try:
- # this code is expected to throw exception
- except ExceptionType as e:
- # code to handle exception

# Using exceptions objects

try:

```
number =(input("Enter a number: "))
```

```
print("The number entered is", number)
```

except NameError as xyz:

```
print("Exception:", xyz)
```

# User Defined Exceptions

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions. This can be done using classes
- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- **Class** – A **class** can be defined as a template/blueprint that describes the behavior/state that the **object** of its type support.
- An **object** is an instance of a **class**

# Class

- `class MyClass:`  
    `x = 5`
- `p1 = MyClass()`  
    `print(p1.x)`
- `Class class_name:`  
    `def funcname(1starg=class itself,2ndarg,3rdarg...):`
- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created:

# Class and Objects

- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- `class person:`
- `def __init__(self,name,age,contact,*arg):`
- `self.name=name`
- `self.age=age`
- `self.contact=contact`
- `self.address=arg[0]`
- `self.profession=arg[1]`
- Object Methods:
- `p1=person("xyz",10,12345,"surat","abc")`
- `print p1.name`
- `print p1.address`

# Class and Objects

```
class person:
 def __init__(self,name,age,contact,*arg):
 self.name=name
 self.age=age
 self.contact=contact
 self.address=arg[0]
 self.profession=arg[1]
 def full_name(self,fullname):
 print("Hello my name is " + self.name)
 self.name=fullname
```

Object Methods:

```
p1=person("xyz",10,12345,"surat","abc")
print p1.name
print p1.address
p1.fullname="xyz abc"
```



# Modify Object

- `p1.age = 40`
- `del p1.age`
- `del p1`

# Class Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

# Class Inheritance

- ```
class Person:  
    def __init__(self, fname,  
lname):  
        self.firstname = fname  
        self.lastname = lname  
  
    def printname(self):  
        print(self.firstname,  
self.lastname)  
  
#Use the Person class to  
create an object, and then  
execute the printname  
method:  
  
x = Person("John", "Doe")  
x.printname()
```
- ```
class Student(Person):
 pass
```
- Use the pass keyword when you do not want to add any other properties or methods to the class.
- Now the Student class has the same properties and methods as the Person class.
- Use the Student class to create an object, and then execute the printname method:  

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

# Class Inheritance

- The `__init__()` function is called automatically every time the class is being used to create a new object.
  - When you add the `__init__()` function in child, the child class will no longer inherit the parent's `__init__()` function.
  - The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.
- ```
class Student(Person):
```
 - ```
 def __init__(self,
```
  - ```
        fname,mname, lname):
```
 - ```
 Person.__init__(self,
```
  - ```
            fname, lname)
```
 - ```
 self.fname=fname
```
  - ```
            self.mname=mname
```
 - ```
 self.lname=lname
```
  - ```
        def printfunc(self):
```
 - ```
 print(self.fname+self.lname)
```
  - ```
xyz=Student("x","y","z")
```
 - ```
xyz.printfunc()
```

# Class Inheritance

- To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:
- ```
class Student(Person):  
    def __init__(self, fname, lname):  
        Person.__init__(self, fname, lname)
```
- Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:
- ```
class Student(Person):
 def __init__(self, fname, lname):
 super().__init__(fname, lname)
```

# Class Inheritance

- `class Student(Person):`
- `def __init__(self, fname, mname, lname):`
- `Person.__init__(self, fname, lname)`
- `self.fname=fname`
- `self.mname=mname`
- `self.lname=lname`
- `def printfunc(self):`
- `print(self.fname+self.lname)`
- `xyz=Student("x","y","z")`
- `xyz.printfunc()`

- `class Student(Person):`
  - `def __init__(self, fname, lname, year):`
    - `super().__init__(fname, lname)`
    - `self.graduationyear = year`
- `x = Student("Mike", "Olsen", 2019)`

# Class Inheritance

- `class Student(Person):`  
    `def __init__(self, fname, lname, year):`  
        `super().__init__(fname, lname)`  
        `self.graduationyear = year`  
  
    `def welcome(self):`  
        `print("Welcome", self.firstname,`  
`self.lastname, "to the class of",`  
`self.graduationyear)`

# User Defined Exceptions

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.
- In the try block, the user-defined exception which is defined as sub-class of any standard exception is raised and caught in the except block.



# User Defined Exceptions

```
class Error(ValueError):
 """Base class for other
 exceptions"""
 pass
class
ValueTooSmallError(Error):
 """Raised when the input
 value is too small"""
 pass
class
ValueTooLargeError(Error):
 """Raised when the input
 value is too large"""
 pass
```

```
while True:
 try:
 sem = int(input("Enter your semester: "))
 if sem < 1:
 raise ValueTooSmallError
 elif sem > 8:
 raise ValueTooLargeError
 break
 except ValueTooSmallError:
 print("This value is too small, try again!")
 print()
 except ValueTooLargeError:
 print("This value is too large, try again!")
 print()
 except:
 print("Something went wrong")
 else:
 print ("Welcome to %d semester. Your
subjects for this semester are:....."%(sem))
 finally:
 print ("Copyright of XYZ Technological
University")
```

# Python Inbuilt Functions

|                            |                          | Built-in Functions        |                           |                             |
|----------------------------|--------------------------|---------------------------|---------------------------|-----------------------------|
| <code>abs()</code>         | <code>delattr()</code>   | <code>hash()</code>       | <code>memoryview()</code> | <code>set()</code>          |
| <code>all()</code>         | <code>dict()</code>      | <code>help()</code>       | <code>min()</code>        | <code>setattr()</code>      |
| <code>any()</code>         | <code>dir()</code>       | <code>hex()</code>        | <code>next()</code>       | <code>slice()</code>        |
| <code>ascii()</code>       | <code>divmod()</code>    | <code>id()</code>         | <code>object()</code>     | <code>sorted()</code>       |
| <code>bin()</code>         | <code>enumerate()</code> | <code>input()</code>      | <code>oct()</code>        | <code>staticmethod()</code> |
| <code>bool()</code>        | <code>eval()</code>      | <code>int()</code>        | <code>open()</code>       | <code>str()</code>          |
| <code>breakpoint()</code>  | <code>exec()</code>      | <code>isinstance()</code> | <code>ord()</code>        | <code>sum()</code>          |
| <code>bytearray()</code>   | <code>filter()</code>    | <code>issubclass()</code> | <code>pow()</code>        | <code>super()</code>        |
| <code>bytes()</code>       | <code>float()</code>     | <code>iter()</code>       | <code>print()</code>      | <code>tuple()</code>        |
| <code>callable()</code>    | <code>format()</code>    | <code>len()</code>        | <code>property()</code>   | <code>type()</code>         |
| <code>chr()</code>         | <code>frozenset()</code> | <code>list()</code>       | <code>range()</code>      | <code>vars()</code>         |
| <code>classmethod()</code> | <code>getattr()</code>   | <code>locals()</code>     | <code>repr()</code>       | <code>zip()</code>          |
| <code>compile()</code>     | <code>globals()</code>   | <code>map()</code>        | <code>reversed()</code>   | <code>__import__()</code>   |
| <code>complex()</code>     | <code>hasattr()</code>   | <code>max()</code>        | <code>round()</code>      |                             |

# Callable()

- The callable() function returns True if the specified object is callable, otherwise it returns False.
- callable(*object*)
- def x():  
    a = 5

```
print(callable(x))
```

# ISINSTANCE()

- The `isinstance()` function returns `True` if the specified object is of the specified type, otherwise `False`
- `isinstance(object, type)`
- `x = isinstance(5, int)`
- `x = isinstance("Hello", (float, int, str, list, dict, tuple))`
- `class myObj:`  
    `name = "John"`

`y = myObj()`

`x = isinstance(y, myObj)`

# Issubclass()

- The `issubclass()` function returns `True` if the specified object is a subclass of the specified object, otherwise `False`.
- `issubclass(object, subclass)`
- `class myAge:`  
    `age = 36`

```
class myObj(myAge):
 name = "John"
 age = myAge
```

```
x = issubclass(myObj, myAge)
```

# Vars()

- The vars() function returns the `__dic__` attribute of an object.
- The `__dict__` attribute is a dictionary containing the object's changeable attributes.
- **Note:** calling the vars() function without parameters will return a dictionary containing the local symbol table.
- `vars(object)`
- ```
class Person:  
    name = "John"  
    age = 36  
    country = "norway"
```

```
x = vars(Person)
```

Memoryview()

- The `memoryview()` function returns a memory view object from a specified object.
- It can also return unicode for each byte in for specified location
- `x = memoryview(b"A1ello")`
- `y=memoryview(b'1')`
- `print(x,y[0])`
- `#return the Unicode of the first character`
- `print(x[0])`
- `#return the Unicode of the second character`
- `print(x[1])`

Delattr()

- The `delattr()` function will delete the specified attribute from the specified object.
- `delattr(object, attribute)`
- `class Person:`
 `name = "John"`
 `age = 36`
 `country = "Norway"`

`delattr(Person, 'age')`

getattr()

- The getattr() function returns the value of the specified attribute from the specified object.
- `getattr(object, attribute, default)`
 - *object* Required. An object.
 - *attribute* The name of the attribute you want to get the value from
 - *default* Optional. The value to return if the attribute does not exist
- `class Person:`
 `name = "John"`
 `age = 36`
 `country = "Norway"`

 `x = getattr(Person, 'age')`
- `x = getattr(Person, 'page', 'Sorry! Result not found')`

Hasattr()

- The `hasattr()` function returns `True` if the specified object has the specified attribute, otherwise `False`.
- `hasattr(object, attribute)`
- `class Person:`
 `name = "John"`
 `age = 36`
 `country = "Norway"`

```
x = hasattr(Person, 'age')
```

Setattr()

- The setattr() function sets the value of the specified attribute of the specified object.
- setattr(*object*, *attribute*, *value*)
 - *object* Required. An object.
 - *attribute* Required. The name of the attribute you want to set
 - *value* Required. The value you want to give the specified attribute
- class Person:
- name = "John"
- age = 36
- country = "Norway"

- setattr(Person, 'age', 40)
- # The age property will now have the value: 40
- x = getattr(Person, 'age')
- print(x)

if `__name__` == `"__main__"`

- Before executing code, Python interpreter reads source file and define few special variables/global variables.
If the python interpreter is running that module (the source file) as the main program, it sets the special `__name__` variable to have a value `"__main__"`. If this file is being imported from another module, `__name__` will be set to the **module's name**. Module's name is available as value to `__name__` global variable.
A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended.
- `# Python program to execute`
- `# main directly`
- `print "Always executed"`
-
- `if __name__ == "__main__":`
- `print "Executed when invoked directly"`
- `else:`
- `print "Executed when imported"`

if `__name__` == `"__main__"`

- # Python program to execute
- # function directly
- `def my_function()`
- `print "I am inside function"`
-
- # We can test function by calling it.
- `my_function()`
- Now if I want to use this as a module, I have comment out the `my_function` line. Again uncomment to use it directly and again comment to use as module

Direct:

- `def my_function()`
- `print "I am inside function"`
- `Def f2():`
 - `Print("Hi")`
- `if __name__ == "__main__":`
 - `my_function()`
 - `F2()`

As module:

- `import myscript`
- `myscript.my_function()`

Lib

- To enlist functions/constants/methods supported by any library use `dir()` function
- To understand the purpose of the functions offered by library read the manual of the library which would be available on <https://docs.python.org/3/library/>
- For understanding functions in third party libraries either read the manual on github or on the developer's website or go through online tutorial of the library

debugging

- Logical debugging can be done by placing print statements or storing data in list as per the program's structure
- Python IDLE's debugging tool can also be used for the same.

debugging

- A **bug** is an unexpected problem in your program. They can appear in many forms, and some are more difficult to fix than others. Some bugs are tricky enough that you won't be able to catch them by just reading through your program. Luckily, Python IDLE provides some basic tools that will help you debug your programs with ease!

debugging

- If you want to run your code with the built-in debugger, then you'll need to turn this feature on. To do so, select *Debug* → *Debugger* from the Python IDLE menu bar. In the interpreter, you should see [DEBUG ON] appear just before the prompt (>>>), which means the interpreter is ready and waiting.
- **Now open your file, define globals and locals, set breakpoints, run the program and GO for debugging**

debugging

- **Breakpoints**
- A **breakpoint** is a line of code that you've identified as a place where the interpreter should pause while running your code. They will only work when *DEBUG* mode is turned on, so make sure that you've done that first.
- To set a breakpoint, right-click on the line of code that you wish to pause. This will highlight the line of code in yellow as a visual indication of a set breakpoint. You can set as many breakpoints in your code as you like. To undo a breakpoint, right-click the same line again and select *Clear Breakpoint*.
- Once you've set your breakpoints and turned on *DEBUG* mode, you can run your code as you would normally. The debugger window will pop up, and you can start stepping through your code manually.

debugging

- In this window, you can inspect the values of your local and global variables as your code executes. This gives you insight into how your data is being manipulated as your code runs.
 - You can also click the following buttons to move through your code:
 - **Go:** Press this to advance execution to the next [breakpoint](#). You'll learn about these in the next section.
 - **Step:** Press this to execute the current line and go to the next one.
 - **Over:** If the current line of code contains a function call, then press this to step *over* that function. In other words, execute that function and go to the next line, but don't pause while executing the function (unless there is a breakpoint).
 - **Out:** If the current line of code is in a function, then press this to step *out* of this function. In other words, continue the execution of this function until you return from it.
 - Be careful, because there is no reverse button! You can only step forward in time through your program's execution.
-
- You'll also see four checkboxes in the debug window:
 - **Globals:** your program's global information
 - **Locals:** your program's local information during execution
 - **Stack:** the functions that run during execution
 - **Source:** your file in the IDLE editor

Python advance

- Python can be further used in following areas:
 - Web Development
 - Game Development
 - Database Access
 - Software Development
 - Machine Learning and AI
 - GUI Development
 - Network Programming
 - Science and Numeric Applications
 - Data Science and Data Analysis
 - Automation and Cloud Computing
- and many more