# Introduction to Programming (CSE100)

# Lecture 1 (Intro)

Lecturer - Raghava
Teaching fellow - Bhawna

Labs will be conducted on hackerrank
There will be 12 labs

Short quizzes will be held on the ALT app
An alternate google form will also be provided
GoogleForm

## Bonus Marks

There will be a few bonus exercises
They will be of only 10% only if other exercises are completed with a score > 80%

## Books

Intro to computation and programming using python - John Guttag

## Marks distribution

- Marks distribution (exact % will depend on students' behavior and performance):
  - 5-10% for "in-class exercises" (depending on response)
  - 10-15% for 2 pre-announced quizzes
  - 20-30% for Labs
  - 20-30% for Assignments
  - ~20% for mid-semester exam
  - ~20% to end-semester exam

# Lecture 2 (Basic Python)

- Expressions
- `print()` Statement
- `input()` Statement

# Lecture 3 (Operations)

- Variable name assignment
- Variable and scalar Data Type
  - Data Type can be scalar and compound/structured
  - Simple scalar are - `int` , `float` , `bool` , `None`
- Arithmetic Operations
- Boolean Operations
- Operation Priorities (lowest to hightest)
  - `or`
  - `and`
  - `not`
  - `==,!=, <=, >=, <, >`
  - `+, -`
  - `*, /, //, %`
  - `+x, -x`
  - `**`
  - `()`

# Lecture 4 (if-else blocks)

- `if`, `elif` and `else` statements
- **Blocks** of statements

# Lecture 5 (loops)

- `pass` statement
- `for` loop
- control flow (FlowCharts)

# Lecture 6 (More about loops)

- `while` loops
- iterable data structures
- `break` statement
  - Exits the innermost loop
- `continue` statement
  - Skips the code underneath and goes back to the top of the innermost loop to check the condition
- Nesting of loops

# Lecture 7 (Functions)

- Calling a function with `def`
- Declaration:

```
def fn_name(parameters):
  <fn_body>
```

- Local and global variables
  - `global` statements

# Lecture 8 (More about functions)

- Importance of function
  - Divide and conquer
  - Abstraction
  - Reusable
  - Modular
- Docstring
- Top-Down Approach

  > make dummy function, work on the main function first, and
  > write down the function later.

# Lecture 9 (Structured Datatype)

- Slicing

  `var[start:end:step]`
- `len()`
- `sum()`
- `list.append(a)` , `list.insert(index, a)` , `list.extend([a])`

# Lecture 10 (list)

- `list.sort()` sorts in ascending order
- `list.sort(reverse=True)` sorts in descending order
- `list.sort(key=len)` sorting of items is done as per the value len(item) of each item
- list comprehension

# Lecture 11

## Strings

- `ord("char")` to get the order of a character
- `chr(ord)` to get the character at the order
- `str.split()`
- `str.join(list)`

```
",".join(["one", "two"])
# "one,two"
```

- Huge number of methods available - all methods return new values - do not change original string
- Some we have seen, some others are:
  - s.count("str") # returns how many times "str" occurs in s
  - s.find("str")  # index from where str is found in s; -1 if not found
  - s.isalnum() # returns True if all are alpha numeric
  - s.isdigit() # True if all are digits, else false
  - s.capitalize()  # capitalizes the first char and converts all other characters in the string to lowercase
  - s.title() # Capitalizes the first letter of every word in the string and converts the rest in lowercase
  - s.isupper() and s.islower(). # Returns true if s is an uppercase string and lowercase string respectively.
  - s.startswith(str) and s.endswith(str). # Used for checking suffixes and prefixes

# Sets

Following operations are allowed on a set s (modifies the set s)

**s.add(item)** # will add item to s, if does not already exist

**s.remove(item)** # will remove item if it exists, error otherwise

**s.discard(item)** # no error if item does not exist

**s.clear()** # clears the set

**s.update(s2)** # s2 is a set, list, tuple: adds elts of s2 to s - duplicates dropped, i.e. does a union operation

**del s** # Completely deletes the set s

- **s1.union(s2)** # returns the union of s1 and s2
- Can also be done by s1 | s2
- Can have union of multiple sets, s1.union(s2, s3); s1|s2|s3

- **s1.intersection(s2)** # the intersection of s1 and s2
- Can also be done by: s1 & s2
- Can have intersection of multiple sets.

- **s1.difference(s2)** # items in s1 which are not in s2
- Can also be done by: s1 - s2

- `s1.symmetric_difference(s2) => (s1 | s2) - (s1 & s2)`
- `s1.isdisjoint(s2)` , True if the sets are disjoined
- `s1.issubset(s2)` , True if s1 is a sub-set of s2
- `s1.issuperset(s2)` , True if s1 is super-set of s2
- Relationship operation holds true for sets

# Lecture 12

- `forzenset()` - are immutable set
- tuple
  - `tuple.count()`
  - `tuple.index(ele)`
- `sorted(sequence)` return a sorted list irrespective of the data-type of the sequence
- Dictionary
  - `dict.get(key)` , return None if key doesn't exist
  - `dict.popitem()` removes the last element added to the dict
  - `dict.items()` return list of tuple of key and value
  - `dict.copy()`
  - `dict.fromkeys(<keys>,defValue)`

# Lecture 13

- Common function for iterables

There are some common functions which are useful - some apply to structured types, some to all objects

- type(x) # returns the type of x
- len(x) # returns the no of items in x
- all(x) # returns True if all elements of an iterable are true
  - all([1,1,1]) -> True ; all([1,0,0]) -> False
- any(x) # returns True if any elements of an iterable are true
  - any([1,0,0]) -> True ; any([0,0,0]) -> False
- reversed(x) # returns a reverse iterator
- sorted(x) # returns a new sorted list from the items in iterable

- `enumerate(iterable)` gives index and value simultaneously of an iterable object
- Hashing

## Set Implementation

- Sets have unique values (no duplicates) - so can use hashing
- In hashing, a contiguous memory, much larger than the number of items, is assigned
- An item value is converted by a hash function to an index in this large memory - index is used to access the item (which is fast)
- For using hashing on a collection of items, each item must be unique so its hashing returns a unique index
- E.g. consider {12345, 45678, 90876, 6789}, suppose hash function is item%100 (i.e. the last two digits)
- A contiguous memory for 100 items will be assigned (each item will have a flag - exists/not exists, and value)
- Item 12345, will be in 45th location, 45678 in 78th ..
- Hashing function is generally fast

- Membership checking for an item - hash the item to get index, check directly in the memory location - ***fast***
- Adding an item in set - check at index: hash(item) - if exists, then return, else, add the item in this - ***fast***
- Deleting an item - ***fast*** - just go to the index provided by hash
- So, set operations are fast - the cost is that it takes more memory (for a set of n items, the memory may be much larger than need for just keeping n items)

```
# Method 1
res1 = []
for i in tstl:
    if i not in res1:
       res1.append(i)

# Method 2
res2 = []
[res2.append(x) for x in tstl if x not in res2]

# Method 3
    res3 = list(set(tstl))
```

Which of these is most efficient?

Run these for lists of 10K size

Times (sec): 0.6, 0.4, 0.0007

**Method 3** is extremely efficient

# Lecture 14 (Encryption)

## Encryption

### Caesar Cipher

A very simple cipher where each letter is replaced by a letter which is a fixed number of positions away from the current letter

# Lecture 15 (FIle I/O)

- to open a file `file = open(fileName, importMode)`
- `file.close()` to close the file
- File I/O modes
  - read: `"r"` ; Error if file doesn't exist
  - write: `"w"` ; Creates a new file if file doesn't exist
  - append: `"a"` ; Creates a new file if file doesn't exist
- reading a file
  - `file.read()` reads the entire file in string
  - `file.read(byte)` read the specified no. of bytes
  - `file.readline()` read a single line of a file
  - `file.readlines()` makes a list of all the lines in the file
- writing a file
  - `file.write(str)` writes a string to the file
  - `file.writelines([strs])` writes all the strings in the iterable
- `with` keyword, this helps the programmer to close the file

```
with open(<fname>, <importMode>) as f:
    <codeBlock>
```

- ***There might be spaces before or after a line***

  use `strip()` to get rid of it

# File I/O (Dr Jalote)

## CSV

- We can use `csv` library

  ```
  import csv
  ```

- A csv file can be opened as a regular file

  ```
  f = open("a.csv", "r")
  ```

- Reading from csv

  ```
  csv_r = csv.reader(f, delimiter=',')
  # delimiter can be omitted
  # csv_r is a list of rows
  # a row is a list of strings
  ```

- DictReader

  It assumes that the first row contains the names of the items and use those as keys for reading

other rows

```
with open("demo.csv", "r") as f:
  for row in csv.DictReader(f):
    print(row)

""" demo.csv
RNo, Name
1, a
2, b
"""
""" output
{"RNo": "1", "Name": "a"}
{"RNo": "2", "Name": "b"}
"""
```

## Writing CSV : Method 1

```
import csv

header = ["Roll Number", "Name"]
data = [["r1","abc"],["r2", "pqr"], ["r3", "xyz"]]

with open("demo.csv", "w") as f:
    csv_writer = csv.writer(f) # Create the writer object
    csv_writer.writerow(header) # Write the header
    csv_writer.writerows(data) # Write the data
```

demo.csv

```
Roll Number, Name
r1, abc
r2, pqr
r3, xyz
```

## Writing CSV: Method 2

- Use DictWriter; methods available are
- writeheader(DW)
- writerow(DW, rowdict)
- writerows(DW, rowdicts)

**JSON**

- We need to import json library

  ```
  import json
  ```
- JSON file can be opened as regular file
- `json.load(file)` parses json data from file and return a dictionary
- Sometimes, JSON may be a string in your program, that can be converted to a dictionary object

  ```
  data = json.load(str)
  ```
- writing to a str

  ```
  json_str = json.dump(<dict>, indent=4)
  ```

- writing to a file

  ```
  json.dump(<dict>, file)
  ```

# Reading from Web using API

## HTTP

It is the protocol we use in the web browser and for humans and not for programs

## REST

REST is a framework made for programs to make requests

- REST: REpresentational State Transfer
- It is an architectural style for client-server interaction on the internet

## REST on HTTP

- Operations used on REST on HTTP:
  - GET- read/ retrieve
  - POST- create new resource
  - PUT- update existing resource
  - DELETE- delete a resource

## How to make an API request

- import request package
- Make a request to an API, save the response

```
import requests
resp = requests.get(<api-url>)
```

- We can also request with the parameter rather than typing it on the url

```
import requests
resp = requests.get(<api-url>, params=<dict>)
# dict is with parameter as specified keys
```

## Response attribute

- resp.status_code # the status code of teh request
- resp.headers # headers of the response - dictionary in json
- resp.txt # the text returned- a string; often json in a string
- resp.request # information about the request - like headers, etc

## status_code

- 200: OK, request successful
- 404: Bad request - request was incomplete
- 401: unauthorized - need credentials to access
- 404: Not found

## Response text

- we can directly extract the dictionary via
  `resp.json()`
- We can also use `json.load(string)`

# Authentication

- API Key

# Lecture 16 (Recursion)

- A recursive function in python is a function which calls itself
- All recursive functions must contain:
  - A base case:

    In which there is no recursive call to the function, and returns a clear value
  - Recursive part:

    Where the function itself is called but with different parameters
- When a function is called, a space separator (called frame) is allocated to this call with local vars (including parameters)

## Functions: Global and local var

- Global vars - those declared in the main scope i.e. outside of any fn - they can be accessed anywhere
- Local vars - those created in the function - do not have existence outside the function
- In a python function, variables only referenced are considered global unless a local var exists; if a variable is assigned a value, it is treated as local unless declared global
  - Allows use of global names of functions, etc (and global values)
  - Disallows accidental changing of a global variable
- If we have a var pointing to a mutable object, then assigning to an element is not changing the var, hence it is allowed
  - So, we can pass a list to a function, and have the fn change items
  - But we cannot pass a string/int - and have its value changed by the function (a function can return a new string/int)

- Care must be taken during recursion as all the frames created during recursion takes up memory and can lead to a StackOverflow error. To stop this python stops the recursion at the $1000^{th}$ recursion call.
- When a recursive function is called many times with the same values then we can use **Memoization** to save time and memory. Memoization is like creating your own cache.
- Cache is best implemented with python using dictionary

# Lecture 17 (Classes)

- Functions can be assigned to a variable
- When assigning the function the `()` are not used
- That var can be used to call the function
- Functions can be passed as an argument of another function
- Function can be an element of a list

- Functions can be returned

```
def add(x,y):
    return x+y
def mult(x,y):
    return x*y

def getfn(s):
    if s == "+":
        return add
    if s == "*":
        return mult

f = getfn("*")
g = getfn("+")
print("Result of Operation: ",
f(4,5) + g(4,5))
```

- Functions can be nested
- A function can also return a nested function
- Scope of nested functions

## Scope of Variables with Nested Functions

- With inner functions, we have nested scopes
  - Built-in: Those defined by python
  - Global scope: names defined in main - are accessible anywhere
  - Local scope:  names declared in a function (incl the parm); accessible anywhere within the function
  - Enclosing scope: names declared in the enclosing function
- Python searches for a name (a var, fn, obj,..) in this order:
  - Local scope first - if a name found here, used
  - Enclosing scope: if not found in local, look for in enclosing scope
  - Global scope: if not found in local or enclosing (after this, built-in)
- Defining/Assigning to variables
  - To assign to a global var within a function, it must be declared "global"
  - To assign to a variable in scope of an outer function, but not global, it must be declared "non local"

- Closure
  - A function which has inner function can return the inner function to the main, which has access to the local vars of the outer function
  - And when the outer function returns its frame is removed, making its local variable disappear
  - But the main function has access to the inner function which can invoke the local vars of the outer function

# Lecture 18 (Module)

- A module refers to a file containing python definition and statements
- A module defines a namespace of its functions, global vars, ...
- Generally we access modules using .<fn_name>
  - this allows different modules to have same names and making name conflict in importing module
- Main function

## if __name__ == “__main__”:

- When we directly run a python file, then the the built-in var "__name__" is set to "__main__"
- On import stmt, the executables of the imported module are executed (function definitions are noted), __name__ is the name of the module
- So, when we run a program file, __name__ is __main__, but when we import it, __name__ is the module name
- We can use this to define a python program which can be run as a script when needed, and imported when desired:
    - For executable statements (which we want to run as a script), first check if __name__ == “__main__” and then execute them.
- Any code placed under this is only executed when the python program is run directly and is ignored when the python file is imported as a module.

# Lecture 19 (Error handling)

- **Compile time error** - Syntax error
- **Run time Error are Exceptions**

## Built-in exceptions

- **IndexError**: when index is out of bound
- **KeyError**: when key not found in dictionary
- **NameError**: when local var is used but no value is assigned
- **ZeroDivisionError**: Divide by 0 encountered
- **ValueError**: Operation receives a value when at is not appropriate
- **IOError**: wrong file name or path while reading
- **RuntimeError**: A general error when no other case fits

try-except block is used to handle errors

# Lecture 21 (OOP)

- We can create new data types using OOP
- A data type defines a collection of data objects and a set of predefined operations on those objects
- To support the operations, whatever information needed is maintained by implementation of the type

# Defining a class

```
class <className>:
  <body>
```

we can declare object of a class by

```
object1 = <className>()
```

# Classes contains

- methods- functions which define the operations on this new type
- attributes- these are variables in its scope, accessed from within the class to implement the methods of the class
- attributes are defined in methods in the class, usually using the `__init__`

# `__init__` Method

- `__init__` typically will get some values as params and use them to define the state of the object being created, i.e., assign values to attributes for the object
- If no `__init__` provided, creation of an empty object (whose state can be defined with state by calling methods)

# Methods

## Class and Objects – calling Methods

- A class has no state, it is just a definition - the state is within objects, just like list itself has no state - only a list object we create has state
- An object can be created and initialized through the method __init__()
- On an object of a class, the methods that are defined in the class are the set of operations that can be performed - like on sets we can perform the operations defined on sets
- Operations defined in a class can be invoked on an object using

  object_variable.operation (args)

- When a method is invoked on an object, the function defined in class executes on the state of the object on which it is invoked

## Method Calls and self

- There can be many objects of a class - each has its own state
- To execute the methods defined in class, we need to identify which object to perform them on
- When a method is invoked on an object, the reference to the object is passed implicitly to the method as an argument
- In the method, it is the first parameter **self** - the ref to the object
- All attributes of an object are accessed as self.attribute
- In methods, those variables associated with self define the class attributes - other variables defined method are local
- All objects have these attributes - each attribute has its own state
- In a method on an object - must access attributes through "self"

# Execution with classes

- When a program is executed, the class definition is noted by the interpreter, but no method is looked at
- When an object is created, then `__init__()` method is visited and executed

# Lecture 22 (OOP)

## Example: Postfix Expression Evaluation using Stack

- Stack is a common type - used in many applications
- Arithmetic expressions can also be written in postfix (also called reverse polish) notation - no parenthesis is required
  - The operator is mentioned after the two operands
  - The result of the operation replaces the two operands and op

    `(a+b)*c => ab+c*  ; a/b+c/d => ab/cd/+;  (a+b)*(c+d)  => ab+cd+*`
- Evaluating postfix notation is efficient - best way is to use a stack

  *If operand -> push it on a stack*

  *If operator -> pop last two values, apply operator, push result on stack*

  *When expression ended -> the only value on stack is the answer*
- We need to create a stack data type for this

- Evaluating a postfix operation using stack

## Postfix eval using stack...

```
class Stack:
    def __init__(self):
        self.top = 0
        self.data = [None]*20
    def push(self, item):
        self.data[self.top] = item
        self.top += 1
    def pop(self):
        if self.top < 1:
            print("Error - popping empty stack")
        item = self.data[self.top-1]
        self.top -= 1
        return(item)
    def isempty(self):
        return self.top <= 0
```

```
def postfix(l):
    estk = Stack()
    for i in l:
        if isinstance(i,float) or isinstance(i, int):
            estk.push(i)
        if isinstance(i, str):
            x = estk.pop()
            y = estk.pop()
            if i == "*":
                estk.push(x*y)
            elif i == "+":
                estk.push(x+y)
            elif …

l = [2, 3, 1, "*", "+", 9, "-"]
print(postfix(l))
```

# Dunder method

- With predefined objects we can perform standard operations, i.e., print(), len()
- Dunder method provides a way to make standard ops available for custom objects.
- these functions name start with __

# The __str__ method

- How do we print an object - for lists, sets, etc - the print() prints it in some format (we can also do str(list/set/..) to get a string equivalent)
- For a class object o, if you print(o), you will get the type of the object and a memory location where it is. For example,

```
#<__main__.Complex object at 0x7fbfb42ae4c0>
```

- To print and object we can provide suitable method to print relevant info - and then invoke it on the object
- Or we can provide a method __str__(self) to return a string - containing info about the state of the object. With __str__(), when print(o) is called, __str__() is invoked, which returns a string that gets printed
- __str__() is also called when str(obj) is called

- Similarly, for `__len__`

# Some std fns that work on Class Objects

- There is a function *is* - which takes two vars and returns True if both vars point to the same object
  - This will work for user defined classes also - as it just needs to check the value of the two vars
- There is another function isinstance(*obj*, *type*) - this returns True if the object *obj* is of the type *type*
  - This will also work on class type objects
- These operations works for user defined class objects also

- For some operations

## Dunder methods for some common ops

| Operation | Dunder method |
|---|---|
| + | object.__add__(self, other) |
| - | object.__sub__(self, other) |
| * | object.__mul__(self, other) |
| == | object.__eq__(self, other) |
| != | object.__ne__(self, other) |
| >= | object.__ge__(self, other) |

- Example

```
def __add__(self, b):
    return self.sum + b.sum
```

## Copying objects

- Class objects are mutable
- => o2 = o1 will make the pointer of both the objects same so the changes in one will be reflected in both of them
- We can copy objects by using copy module

```
import copy

o2 = o1.copy # For creating a shallow copy (doesn't copy nested objects)
o2 = o1.copy # For creating deep copy (created full copy rather than pointer for nes
```

# Lecture 23 (OOPS) - Jalote Sir

# Composition of classes

- Composition of classes require properly defining classes, so that it can be easily used in other classes
- We need to know about the attributes of composing classes but not how it functions

# Searching

- For a set or dictionary, they internally use hashing, therefore searching is fast as it just directly check if the element is present

## Searching for an item in list

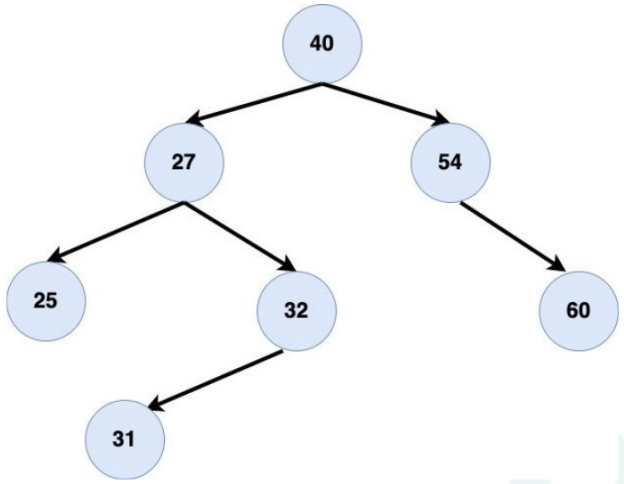
- If list is not sorted then we will have to compare all the items in the list - slow up to len(list) comparisons
- if the list is sorted then the search is faster, using Binary search algorithm

# Sorting

- Bubble algorithm
  - Check a list and push the biggest item to the end
    - By checking two elements in the list and the bigger item is swapped to be the right one
  - make the list 1 element shorter and do the same
- Note: Use inbuilt sort function as it is way more efficient
- When there is too many search and appending and removing then use binary tree
  - **Binary Tree**

## Example – Binary Search Tree

- Binary search tree - a root with value, and links to a tree on left and right
- Left sub-tree has values smaller than the root
- Right subtree has values greater than the root
- Searching for an element with binary search tree is very efficient - recursion is natural here
- Printing sorted items is easy




# Lecture 24 (Inheritance)

- In this a new class borrows attributes and methods of an existing class and add some more features to it
- With inheritance we define a new class using an existing class definition and specifying some modifications/enhancements to it
- The new class is called derived class, child class or subclass and the onw which it inherits is called the base call or parent/super class
- Syntax

```
class BaseClass:
    <body of baseClass>
    # Attributes and methods

  class DerivedClass (BaseClass):
    <body of derivedClass>
```

- On inheriting, all the attributes and methods of BaseClass are available to this class
- Derived class is said to extend the base class
- When an attribute is called from the derived class then python first searches the derived class and if not found then it goes to the derived class
- Liskov's substitution principle says that objects of superclass can be replaced by objects of a subclass without breaking the application

    - This requires the objects of subclass to behave in the same way as the objects of the superclass
- In subclass code we can use super() to refer to the parent class
- super() provides a proxy object through which all the methods of base class can be called

# Converting objects of base type to derived type

- Derived class are "specialized" version of base class
- We can change a base class object to derived class object by using `__class__` attribute
- it is best to avoid this type of conversions

# Abstract class / Interfaces

- A base class can define some abstract methods: they define the interface but there is no body to execute it
- An abstract class is one which has some abstract methods
- Abstract classes require subclasses to redefine the abstract method with actual implementation
- Often used for the purpose of defining standard API/ interfaces
-