```python
####################### TOPIC 1 - INTRODUCTION TO PYTHON ######################

# 1. FEATURES OF PYTHON

# - Scripting Language

# Python is a general-purpose programming language that is often applied in
# scripting roles. The theoretical difference between the scripting languages
# and compiled languages is that they do not require the compilation step and
# are rather interpreted. Generally, compiled programs run faster than
# interpreted programs because they are first converted to native machine code.
# Also, compilers read and analyze the code only once, and report the errors
# collectively that the code might have, but the interpreter will read and
# analyze the code statements each time it meets them and halts at that very
# instance if there is some error.

# - Developer's Productivity

# Python has shined as a tool that allows programmers to get more done with
# less effort. It is deliberately optimized for speed of development – its
# simple syntax, dynamic typing, lack of compile steps and built-in toolset
# allow programmers to develop programs in a fraction of the time needed when
# using some other tools. The net effect is that Python typically boosts
# developer productivity many times beyond the levels supported by traditional
# languages.

# - Program Portability

# Most Python programs run unchanged on all major computer platforms. Porting
# Python code between Linux and Windows, for example, is usually just a matter
# of copying a script's code between machines. Moreover, Python offers multiple
# options for coding portable graphical user interfaces, database access
# programs, web-based systems and more. Even operating system interfaces,
# including program launches and directory processing, are as portable in Python
# as they can possibly be.

# - Support Libraries

# Python comes with a large collection of prebuilt and portable functionality,
# known as the standard library. This library supports a variety of
# application-level programming tasks, from text pattern matching to network
# scripting. In addition, Python can be extended with both homegrown libraries
# and a vast collection of third-party application support software. Python's
# third-party domain offers tools for website construction, numeric programming,
# serial port access, game development and much more. The NumPy extension,
# for instance, has been described as a free and more powerful equivalent to
# the MATLAB numeric programming system.
```

```python
# - Easy Integration with other Languages

# Python scripts can easily communicate with other parts of an application,
# using a variety of integration mechanisms. Such integrations allow Python to
# be used as a product customization and extension tool. Today, Python code can
# invoke C and C++ libraries, can be called from C and C++ programs, can
# integrate with Java and .NET components and much more.

# ------------------------------------------------------------------------------

# 2. WHAT CAN BE DONE WITH PYTHON?

# Almost everything, but major things are:

# - GUIs

# Python comes with a standard object-oriented interface to the Tk GUI API
# called tkinter that allows Python programs to implement portable GUIs with a
# native look and feel. Python/tkinter GUIs run unchanged on Microsoft Windows,
# Linux and the Mac OS (both Classic and OS X). High-level GUIs can be made
# with even more sophisticated tools such as PyQt, PyGTK and PyWin32.

# - Internet Scripting

# Python comes with standard Internet modules that allow Python programs to
# perform a wide variety of networking tasks, in client and server modes.
# Scripts can communicate over sockets;
# extract form information sent to server-side CGI scripts;
# transfer files by FTP;
# parse, generate, and analyse XML files;
# send, receive, compose, and parse email;
# fetch web pages by URLs;
# parse the HTML and XML of fetched web pages;
# communicate over XML-RPC, SOAP, and Telnet; and more.
# In addition, full-blown web development framework packages for Python, such as
# Django, TurboGears, Flask, etc. support quick construction of full-featured
# and production-quality websites with Python.

# - Database Programming

# For traditional database demands, there are Python interfaces to all commonly
# used relational database systems—Sybase, Oracle, Informix, ODBC, MySQL,
# PostgreSQL, SQLite and more. The Python world has also defined a portable
# database API for accessing SQL database systems from Python scripts, which
# looks the same on a variety of underlying database systems.
# Python's standard 'pickle' module provides a simple object persistence system.
# It allows programs to easily save and restore entire Python objects to files
# and file-like objects. On the Web, you'll also find a third-party open source
# system named ZODB that provides a complete object-oriented database system
# for Python scripts, and others (such as SQLObject and SQLAlchemy) that map
# relational tables onto Python's class model.
```

```
# - ML, AI, Data Science, Numerics, Scientific Computing, etc.

# The NumPy numeric programming extension for Python mentioned earlier includes
# such advanced tools as an array object, interfaces to standard mathematical
# libraries and much more. By integrating Python with numeric routines coded
# in a compiled language for speed, NumPy turns Python into a sophisticated
# yet easy-to-use numeric programming tool that can often replace existing code
# written in traditional compiled languages such as FORTRAN or C++.

# - Game Development

# Game programming and multimedia in Python can be done with the Pygame system.

# - Robotics

# Serial port communication on Windows, Linux, and more with the 'PySerial'
# extension and Robot control programming with the 'PyRo' toolkit are few
# examples of commonly used libraries in Robotics.

# ---------------------------------------------------------------------------------

# 3. INTERACTIVE PROMPT

# Introducing Interactive Prompt/ REPL:

# The most platform neutral way to start an interactive interpreter session is
# usually just to type 'python' at your operating system's prompt, without any
# arguments.

# The interactive prompt runs code and echoes results as you go, but it doesn't
# save your code in a file. Although this means you won't do the bulk of your
# coding in interactive sessions, the interactive prompt turns out to be a great
# place to both experiment with the language and test program files on the fly.
# Although the interactive prompt is simple to use, there are a few tips that
# beginners should keep in mind:

# - Type Python commands only.
    # First of all, remember that you can only type Python code at the Python
    # prompt, not system commands. There are ways to run system commands from
    # within Python code (e.g., with os.system), but they are not as direct as
    # simply typing the commands themselves.

# - print statements are required only in files.
    # Because the interactive interpreter automatically prints the results of
    # expressions, you do not need to type complete print statements
    # interactively. This is a nice feature, but it tends to confuse users when
    # they move on to writing code in files: within a code file, you must use
    # print statements to see your output because expression results are not
    # automatically echoed. Remember, you must say print in files, but not
    # interactively.
```

```
# - Watch out for prompt changes for compound statements.
    # You should know that when typing lines 2 and beyond of a compound
    # statement interactively, the prompt may change. In the simple shell window
    # interface, the interactive prompt changes to ... instead of >>> for lines
    # 2 and beyond; in the IDLE interface, lines after the first are
    # automatically indented.

# - Terminate compound statements at the interactive prompt with a blank line.
    # By contrast, blank lines are not required in files and are simply ignored
    # if present. If you don't press Enter twice at the end of a compound
    # statement when working interactively, you'll appear to be stuck in a limbo
    # state, because the interactive interpreter will do nothing at all — it's
    # waiting for you to press 'Enter' again!


# REPL AS CALCULATOR:

    2+2
    50-5*6
    (50-5*6)/4
    8/5         # Division always returns floating point numbers
    17/3
    17//3       # Floor division discards the fractional part
    17%3
    2**7
    4*3.75-2    # Mixed type operations are supported

# FOR EXECUTING PYTHON STATEMENTS:

    print("Hello World")

    # Print statement automatically puts a new line character in the end.
    print ('Hello')
    print ('World')
    # However, if we want both to print on same line, we put one additional
    # argument:
    print('Hello', end=' ')
    print('World')

    # The separator between the arguments to print() function in Python is
    # space by default which can be modified and can be made to any character,
    # integer or string as per our choice.
    print('09','12','2016', sep='-')

# ------------------------------------------------------------------------------
```

```
########################## TOPIC 2 - BASIC DATA TYPES #########################

# 1. VARIABLES and KEYWORDS:

    # When we create a data item, we can either assign it to a variable or
    # insert it into a collection. The name of the variable should start with
    # any character from Unicode set except the digits (0-9), operators
    # (+, -, *, /, etc) and period '.'. Digits can occur in between.
    width = 20
    height = 5*9
    width*height

    # No identifier (i.e. variable name) should be same as Python's Keywords.
    # There are 33 Keywords in Python. They are:

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

    # You can print all the 33 keywords and check if a string is a keyword or
    # not by using 'keyword' module.
    import keyword
    print(keyword.kwlist)                # Output: Returns a list of keywords
    print(keyword.iskeyword('elif'))     # Output: True

    # Apart from these 33 Keywords, one should also take this precaution:
    # Don't use the names of any of Python's predefined identifiers for
    # your own identifiers. For eg. 'int', 'float', 'Ellipsis', etc.
    # To list all the python's built-in identifiers, type the following in REPL:
    dir(__builtins__)
    # There are about 130 names in the list. Those that begin with a capital
    # letter are the names of Python's built-in exceptions; the rest are
    # function and data type names.

    # In interactive mode, the last printed expression is assigned to the
    # variable '_'.
    tax = 12.5 / 100
    price = 100.50
    price * tax
    price + _

# ---------------------------------------------------------------------------
```

```
# 2. OPERATORS:

# ARITHMETIC: + - * / % // **
# RELATIONAL: > < == != >= <=
# LOGICAL: and, or, not.

    # Python provides three logical operators: and, or, and not.
    # Both 'and' and 'or' use short-circuit logic. This means:

    # 'and' returns the first false value. If not found, returns last.
    # 'or' returns the first true value. If not found, returns last.
    3 and 0 and 5                   # Output: 0
    3 and 5 and 10                  # Output: 10
    'large' and '' and 'small'      # Output: ''
    'large' or '' or 'small'        # Output: 'large'

# BITWISE: | & ^ ~ << >>
# ASSIGNMENT: = += -= *= /= %= //= **= &= |= ^= >>= <<=
# IDENTITY: is, is not.

    # 'is' and 'is not' are the identity operators both are used to check
    # if two objects are located on the same part of the memory.
    # Two variables that are equal does not imply that they are identical.

    a1 = 3
    b1 = 3
    a2 = 'Sagar'
    b2 = 'Sagar'
    a3 = [1,2,3]
    b3 = [1,2,3]
    print(a1 is not b1)     # False
    print(a2 is b2)         # True
    print(a3 is b3)         # False, since lists are mutable.

# MEMBERSHIP: in, not in.

    # 'in' and 'not in' are used to test whether a value or variable is in a
    # sequence.
    x = 'Sagar'
    y = [1, 3, 5, 7]
    z = {3:'blue', 1:'brown'}

    print('g' in x)         # True
    print(5 not in y)       # False
    print(3 in z)           # True
    print('brown' in z)     # False

# --------------------------------------------------------------------------------
```

```
# 3. NUMBERS: Int, Bool, Float, Complex, (Decimal)

# a) Integers:

# No limit on range of Python Integers:
    len(str(2**1000000))
    # Output: 301030. (2 raised to 1 million has 301030 digits.)

# USAGE OF BUILT-IN FUNCTIONS:
    # - NUMERIC FUNCTIONS: abs(x), divmod(x, y), pow(x, y, [z]), round(x, n)
    # - CONVERSION FUNCTIONS:

        # Conversion from various bases to decimal.
        a = 1980            # Decimal Notation
        b = 0b11110111100   # Binary Notation
        c = 0x7bc           # Hexadecimal Notation
        d = 0o3674          # Octal Notation
        print(a, b, c, d)
        # Output: 1980 1980 1980 1980

        # Conversion from decimal to various bases.
        bin(a)              # Output: 0b11110111100
        hex(a)              # Output: 0x7bc
        oct(a)              # Output: 0o3674
        print(a, bin(a), hex(a), oct(a))
        # Output: 1980 0b11110111100 0x7bc 0o3674

        # int() function:
        # Syntax: int(x, base)
        # - Converts object x to an integer. If the optional base argument is
        # given it should be an integer between 2 and 36 inclusive.
        # - Raises ValueError on failure or TypeError if x's data type does not
        # support integer conversion.

        x = int('5')    # 5 is assigned to x as integer
        x = int('2.5')
        # Output: ValueError: invalid literal for int() with base 10: '2.5'
        x = int([2, 3])
        # Output: TypeError: int() argument must be a string, a bytes-like
        # object or a number, not 'list'
        x = int('3674', 8)
        print(x)    # 1980

# b) Booleans:

    # There are two built-in Boolean objects: True and False.
    a = True
    b = False
    a and b     # False
    a or True   # True
```

```python
# Boolean types are the Output of the comparison expressions.
# COMPARISON OPERATORS: < > == != <= >=

# In Python, chained comparisons are allowed. Eg. 'X < Y < Z' will result
# in same boolean type as 'X < Y and Y < Z'

# In python: 0 and 0.0 both represent 'False'.
0 == False      # Output: True
0.0 == False    # Output: True

# "", {}, [], () - are neither 'True' nor 'False'.
"" == False     # Output: False
"" == True      # Output: False

# However, they can be typecasted to bool and then they represent 'False'.
bool("") == False   # Output: True
bool([]) == False   # Output: True

# Rest every value of every data type represents 'True'.
bool(0.0)
# Output: False
bool(-0.156)
# Output: True

# True and False in python are same as 1 and 0. Example:
print(True + True + True)       # Output: 3
print(5*True)                   # Output: 5
print(True - True - True)       # Output: -1
print(True + False)             # Output: 1
```

# c) Floating Point Numbers:

```python
# Like integers, even floating point numbers can be converted to hex()
# but not to bin() and oct().

print(5.6.hex())
# Output: 0x1.6666666666666p+2
# The exponent is indicated using p (“power”) rather than e since e is a
# valid hexadecimal digit. This simply shows 5.6 is an object. We will see
# this point later – when we will understand that everything in Python, is an
# object.

# Floating point numbers are needed while doing mathematical operations like
# sqrt(), sin(), cos(), asin(), sinh(), etc. Python outputs 15 digits after
# decimal point by default.

# Doubt: What if I want precision less than 15 digits?
# Ans: Either use string formatting (covered in Topic 6) or use 'Decimal'
#      type, covered later in this section.
```

```python
import math
math.pi              # Pre-defined Mathematical Constant
# Output: 3.141592653589793
math.sqrt(85)        # Mathematical Function
# Output: 9.219544457292887

# We dont know what else 'math' module has to offer. So we can use help.
help(math)      # Press 'Q' to end and return to REPL

math.factorial(6)  # 720
# This is too verbose. If we need to evaluate - nCk ie. choose k from n,
# Writing 'math.' again and again is tedious. Here Python allows us to
# import specific function into current namespace from 'math' module.
from math import factorial
n = 5
k = 3
factorial(n)/(factorial(k)*factorial(n-k))

# During Math operations, two special float values can come as output:
# inf (infinity) and nan (not a number). These two fall under 'float'
# category and not 'int' as we can see below:

float('inf')
# Output: inf
float('-inf')
# Output: -inf
float('nan')
# Output: nan
int('inf')
# Output: ValueError: invalid literal for int() with base 10: 'inf'.

# Python 3.2 and higher allows checking for finiteness.
pos_inf = float('inf')
math.isfinite(pos_inf)
# Output: False
math.isfinite(0.0)
# Output: True

# Comparison operators work as expected for positive and negative infinity.
import sys
sys.float_info.max
# Output: 1.7976931348623157e+308 (this is system dependent)
pos_inf = float('inf')
neg_inf = float('-inf')
pos_inf > sys.float_info.max
# Output: True
neg_inf < -sys.float_info.max
# Output: True
pos_inf == sys.float_info.max * 1.0000001
# Out: True
```

```python
    neg_inf == -sys.float_info.max * 1.0000001
    # Out: True
    -5.0 * pos_inf == neg_inf
    # Out: True
    -5.0 * neg_inf == pos_inf
    # Out: True
    pos_inf * neg_inf == neg_inf
    # Out: True

    # In Math, we can't evaluate 0*inf, inf/inf, inf-inf, etc.
    # Such expressions result into 'NaN'.
    0.0 * pos_inf
    # Out: nan
    pos_inf / pos_inf
    # Out: nan

    # NaN is never equal to anything, not even to itself.
    # We can test for it is with the isnan method.
    float('nan') == float('nan')
    # Output: False
    float('nan') != 5.2
    # Output: True
    float('nan') != float('nan')
    # Output: True
    math.isnan(float('nan'))
    # Output: True

# d) Complex Numbers:

    z = -89.5 + 2.125j
    z.real, z.imag
    # Output: (-89.5, 2.125)

    # Except for //, %, divmod(), and the three-argument pow(),
    # all the numeric operators and functions mentioned with integers
    # can be used with complex numbers as well. In addition to that,
    # one more builtin method i.e. conjugate() can be used with complex types:
    z.conjugate()
    # Output: -89.5 - 2.125j

    # There are many functions for complex types in 'cmath' module.
    import cmath
    cmath.phase(z)
    cmath.polar(z)
    cmath.sin(z)
    cmath.sinh(z)
```

```python
# e) Decimal

    # So far we have seen 4 core numeric types - int, float, bool and complex.
    # These will suffice most of the number crunching that programmers will ever
    # need. Here is an exotic numeric type - Decimal, which merits a quick look
    # here.

    # As you may or may not already know, floating-point math is less than
    # exact, because of the limited space used to store values. For example, the
    # following should yield zero, but it does not. The result is close to zero,
    # but there are not enough bits to be precise here:
    print(0.1 + 0.1 + 0.1 - 0.3)
    # Output: 5.55111512313e-17

    # However, with decimals, the result can be dead-on:
    from decimal import Decimal
    Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
    # Output: Decimal('0.0')

    # We can set precision of the decimal result we want.
    Decimal(1)/Decimal(7)
    # Output: Decimal('0.142857142857142857142857142 9')
    decimal.getcontext().prec = 4
    Decimal(1)/Decimal(7)
    # Output: Decimal('0.1429')

# --------------------------------------------------------------------------------


# 4. STRINGS

# DECLARATION:

    # Both quotation marks are allowed – eg. 'Sagar' or "Sagar".

    # '\' can be used to escape quotes.
    'doesn\'t'
    # If you don't want characters prefaced by '\' to be interpreted as special
    # characters, you can use raw strings by adding an r before the first quote:
    print('C:\some\name')   # here \n means newline!
    print(r'C:\some\name')  # note the r before the quote

    # Triple quotes give multi-line strings.
    print("""\
    Usage: thingy [OPTIONS]
        -h                        Display this usage message
        -H hostname               Hostname to connect to
    """)
```

```python
    # Triple quotes are usually used for declaring 'Docstrings'.
    # Python documentation strings (or docstrings) provide a convenient way of
    # associating documentation with Python modules, functions, classes and
    # methods.

    # What should a docstring look like?
    # - The doc string line should begin with a capital letter and end with a
    #   period.
    # - The first line should be a short description.
    # - If there are more lines in the documentation string, the second line
    #   should be blank, visually separating the summary from the rest of the
    #   description.
    # - The following lines should be one or more paragraphs describing the
    #   object’s calling conventions, its side effects, etc.

    # The docstrings can be accessed using the __doc__ method of the object or
    # using the help function.
    def myfunction():
        """This is a docstring and explains what my function does - Nothing!"""
        return None

    print(myfunction.__doc__)
    # Output: This is a docstring and explains what my function does - Nothing!
    help(myfunction)
    # Output:  Help on function myfunction in module __main__:
    #
    #      myfunction()
    #          This is a docstring and explains what my function does - Nothing!


# STRING CONCATENATION AND REPETITION:

    # Strings can be concatenated with the + operator and repeated with *.
    3 * 'un' + 'ium'    # Output - 'unununium'

    # Two or more string literals (i.e. the ones enclosed between quotes)
    # next to each other are automatically concatenated.
    'Py' 'thon'
    # This feature is particularly useful when you want to break long strings:
    text = ('Put several strings within parentheses '
            'to have them joined together.')
    text
    # Output: 'Put several strings within parentheses to have them joined together.
'

    # This only works with two literals though, not with variables or expressions.
    # If you want to concatenate variables or a variable and a literal, use +.
    prefix = 'Py'
    prefix 'thon' # can't concatenate a variable and a string literal
    # Output: SyntaxError: invalid syntax
```

```python
# SEQUENCE OPERATIONS:

    S = "Spam"
    len(S)
    S[0]    # 'S'
    S[1]    # 'p'
    S[-2]   # 'a'

    S[6]    # IndexError: String Index out of range
    S[-4]   # IndexError: String Index out of range

    # S[a:b] means slice from a upto b, a is inclusive and b is not.
    S[1:3]  # 'pa'
    S[1:]   # 'pam'
    S[:3]   # 'Spa'
    S[:-1]  # 'Spa'
    S[3:1]  # '' - Order is incorrect. Returns empty string.
    S[:]    # 'Spam'

    # S[a:b:c] means get characters from a to b at intervals of c.
    S = "malayalam"
    S[0:6:2]        # 'mly'
    S[::-1]         # Easy way to reverse a string
    S == S[::-1]    # Check if S is a palindrome
    # Output: True


# IMMUTABILITY:

    # Strings are immutable in Python, i.e. We cannot change the contents
    # of the string
    S[3] = 'n'      # Attempt to make S = "Span"
    # Output: TypeError: 'str' object does not support item assignment

    S.replace('m', 'n') # This replaces all occurences of 'm' with 'n' in S
    # Output: 'Span'
    # But this 'Span' is a temporary nameless object. If you print contents of
    # S, it's still - Spam!
    S
    # Output: 'Spam'

    # So what's the way to change it to 'Span'? By creating a new string object
    # and assigining it to S.
    S = S[:3] + 'n'
    S
    # Output: 'Span'
```

```python
# TYPE-SPECIFIC METHODS:

    # We saw usage of replace() method over string type. What other functions we
    # can use on string?
    dir(S)
    # You probably won’t care about the names with underscores in this list
    # until later in the book, when we study operator overloading in classes —
    # they represent the implementation of the string object and are available
    # to support customization. The names without the underscores in this list
    # are the callable methods on string objects. The dir function simply gives
    # the methods’ names. To ask what they do, you can pass them to the help():
    help(S.replace)

    S = "Spam"
    S.upper()           # 'SPAM'
    S.isalpha()         # True

    line = "aaa,bbb,ccc,ddd"
    line.split(',')     # Output: ['aaa', 'bbb', 'ccc', 'ddd']
    line = "aa,bb,cc,dd\n"
    line.rstrip()       # Removes whitespace from right end.
    # Output: 'aa,bb,cc,dd'
    line.rstrip().split(',')    # Combining 2 methods
    # Output: ['aa', 'bb', 'cc', 'dd']

    # Here is the list of common built-in string functions:

    # 1. str.isdecimal() - Returns true if all characters are decimal
    # 2. str.isalnum() - Returns true if all characters are alphanumeric
    # 3. str.isalpha() - Returns true if all characters are alphabets
    # 4. str.max() and str.min() - Returns maximum and minimum alphabetical
    #    character.
    # 5. str.lower() and str.upper() - Converts the string into respective cases.
    # 6. str.find(substr, start, end) : Default values of start = 0 and
    #    end = length-1. The find() method returns the lowest index of the
    #    substring if it is found in given string. If its is not found then
    #    it returns -1.
    # 7. str.rfind(substr, start, end)
    #    rfind() method returns the highest index of the substring if found in
    #    given string. If not found then it returns -1.
    # 8. str.split(seperator, max_limit) and str.rsplit(sep, max_limit):
    #    - separator : The is a delimiter. The string splits at this specified
    #                  separator. If is not provided then any white space is a
    #                  separator.
    #    - max_limit : It is a number, which tells us to split the string into
    #                  maximum of provided number of times. If it is not provided
    #                  then there is no limit.
```

```
    # 9. str.count(substr, start, end)
    #    Returns the number of (nonoverlapping) occurrences of substring in string.
    # 10. str.join(list/tuple) :
    #     The join() method is a string method and returns a string in which
    #     the elements of sequence have been joined by str separator.
    # 11. str.replace(replace_what, replace_with, max_replacements):

# Additional Note: Strings, Lists, Tuples, Dictionaries and Sets are also
# called Iterables. We will see more on iterables and iterators later.

# ---------------------------------------------------------------------------------


# 5. LISTS

# INTRODUCTION:

    squares = [1, 4, 9, 16, 25]
    squares     # Output: [1, 4, 9, 16, 25]

    # Lists also support operations like concatenation
    squares + [36, 49, 64, 81, 100]
    # Output - [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

    # The built-in function len() also applies to lists
    letters = ['a', 'b', 'c', 'd']
    len(letters)    # Output: 4

    L = [123, 'spam', 1.23]     # No type constraints

    # We can convert - string into a list of characters using list()
    S = 'shrubbery'
    L = list(S)
    L   # Output: ['s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y']
    # To get the string back from list, we can use join() method.
    A = ''.join(L)
    A   # Output: 'shrubbery'


# SEQUENCE OPERATIONS:

    # Like strings (and all other built-in sequence types), lists can be indexed
    # and sliced.

    squares[0]      # '1'
    squares[-1]     # '25'
    squares[-3:]    # [9, 16, 25]
    squares[99]     # IndexError: list index out of range
```

```python
# MUTABILITY:

    # Unlike strings, which are immutable, lists are a mutable type,
    # i.e. it is possible to change their content.
    cubes = [1, 8, 27, 65, 125]
    cubes[3] = 64  # replace the wrong value
    cubes
    # Output - [1, 8, 27, 64, 125]

# NESTING:
    a = ['a', 'b', 'c']
    n = [1, 2, 3]
    x = [a, n]       # No type constraints.
    x                # [['a', 'b', 'c'], [1, 2, 3]]
    x[0]             # ['a', 'b', 'c']
    x[0][1]          # 'b'

# TYPE-SPECIFIC METHODS:

# a) Adding elements to the list:

    # Using append():

    # Add an item to the end of the list. Equivalent to a[len(a):] = [x].
    L = []
    L.append(1)
    L.append(2)
    L.append(4)
    L   # [1, 2, 4]

    # Using extend():

    # Extend the list by appending all the items from the iterable.
    # Equivalent to a[len(a):] = iterable.
    L.extend([5, 6, 7])
    L   # [1, 2, 4, 5, 6, 7]
    # Note: append() and extend() can only add elements in the end.

    # Using insert():

    # Insert an item at a given position. The first argument is the index of the
    # element before which to insert, so a.insert(0, x) inserts at the front of
    # the list, and a.insert(len(a), x) is equivalent to a.append(x).
    L.insert(2, 3)
    L   # [1, 2, 3, 4, 5, 6, 7]
```

```python
# b) Removing elements from the list:

    # Using pop():

    # Remove the item at the given position in the list, and return it. If no
    # index is specified, a.pop() removes and returns the last item in the list.
    L.pop()
    L   # [1, 2, 3, 4, 5, 6]
    L.pop(5)
    L   # [1, 2, 3, 4, 5]

    # Using remove():

    # Remove the first item from the list whose value is equal to x.
    # It raises a ValueError if there is no such item.
    L.remove(3)
    L   # [1, 2, 4, 5]

    # Using clear():

    # Remove all items from the list and makes it a null list.
    L.clear()
    L   # []

    # Using del:

    L = [1, 2, 3, 4]
    # We can delete elements or entire list by using 'del' keyword as well.
    # Syntax: del sequence[index]  or del sequence
    del L[2]

    # Deleting the list even deletes the reference.
    del L
    L
    # Output: NameError: name 'L' is not defined

# c) Other utility functions:

    # 1. list.index(x, [start], [end])
    # Return zero-based index in the list of the first item whose value is
    # equal to x. Raises a ValueError if there is no such item.

    # 2. list.count(x)
    # Return the number of times x appears in the list.

    # 3. list.reverse()
    # Reverse the elements of the list in place.

    # 4. list.sort()
    # Sort the items of the list in place.
```

```
    # 5. list.min() and list.max()
    # Returns minimum and maximum valued elements in the list respectively.

# range() FUNCTION:

    range(0, 5)          # Returns range object and not the list
    # The result remains the same even if we use range() inside print() as:
    print(range(0, 5))
    # To get the list, we need to typecast it, i.e.
    list(range(0, 5))
    # We shall revisit this point of range() again at two places: firstly when
    # we shall be looking at the 'for loop' and secondly during 'generators'.

    list(range(4))       # List of 0 to n-1 for range(n)
    # Output: [0, 1, 2, 3]
    list(range(-6, 7, 2))    # -6 to 7 by shift of 2
    # Output: [-6, -4, -2, 0, 2, 4, 6]
    [[x, x**2, x/2] for x in range(-6, 7, 2) if x > 0]
    # Output: [[2, 4, 1], [4, 16, 2], [6, 36, 3]]

# COMPREHENSIONS:

# a) Introduction:

    # List comprehensions provide a concise way to create lists. Common
    # applications are to make new lists where each element is the result
    # of some operations applied to each member of another sequence or
    # iterable, or to create a subsequence of those elements that satisfy
    # a certain condition.

    # For example, assume we want to create a list of squares, like:
    squares = []
    for x in range(10):
        squares.append(x**2)

    squares     # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

    # We can create a similar list in more consice manner:
    squares = [x**2 for x in range(10)]

    # A list comprehension consists of brackets containing an expression
    # followed by a for clause, then zero or more for or if clauses.
    # The result will be a new list resulting from evaluating the expression
    # in the context of the for and if clauses which follow it.

    # For example, this listcomb combines the elements of two lists if they
    # are not equal:
    listcomb = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

    listcomb    # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```python
    # The above listcomb comprehension is equivalent to:
    combs = []
    for x in [1,2,3]:
        for y in [3,1,4]:
            if x != y:
                combs.append((x, y))
    combs   # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
    # Note how the order of the for and if statements is the same in both
    # these snippets.

    # Here are few examples of list comprehensions:

    vec = [-4, -2, 0, 2, 4]
    [x*2 for x in vec]          # create a new list with the values doubled
    [x for x in vec if x >= 0]  # filter the list to exclude negative numbers
    [abs(x) for x in vec]       # apply a function to all the elements

    vec = [[1,2,3], [4,5,6], [7,8,9]]
    [num for elem in vec for num in elem]   # flatten a list using two 'for'
    # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

    from math import pi
    [str(round(pi, i)) for i in range(1, 6)]
    # Output: ['3.1', '3.14', '3.142', '3.1416', '3.14159']

    # List comprehensions become handy while working with matrices.

    M = [[1, 2, 3],     # A 3 × 3 matrix, as nested lists
        [4, 5, 6],      # Code can span lines if bracketed
        [7, 8, 9]]

    col2 = [row[1] for row in M]
    col2    # [2, 5, 8]
    [row[1] + 1 for row in M]                 #  Add 1 to each item in col 2
    # Output: [3, 6, 9]
    [row[1] for row in M if row[1] % 2 == 0]    # Filter out odd items
    # Output: [2, 8]
    diag = [M[i][i] for i in [0, 1, 2]]         # Collect a diagonal from matrix
    diag        # [1, 5, 9]


# b) Nested List Comprehensions:

    # Consider the following example of a 3x4 matrix:
    matrix = [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12],
    ]
```

```python
    # The following list comprehension will transpose rows and columns:
    [[row[i] for row in matrix] for i in range(4)]
    # Output: [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

    # As we saw in the previous section, the nested listcomp is evaluated in the
    # context of the for that follows it, so this example is equivalent to:
    transposed = []
    for i in range(4):
        transposed.append([row[i] for row in matrix])

    # which, in turn, is the same as:
    transposed = []
    for i in range(4):
        transposed_row = []
        for row in matrix:
            transposed_row.append(row[i])
        transposed.append(transposed_row)

    # Similar type of comprehensions are supported with dictionaries, sets and
    # tuples as well.

# SEQUENCE UNPACKING:

# Sequence assignments normally require exactly as many names in the target on
# the left as there are items in the subject on the right. We get an error if
# the lengths disagree.

seq = [1, 2, 3, 4]
a, b, c, d = seq
print(a, b, c, d)   # Output: 1 2 3 4

a, b = seq
# Output: ValueError: too many values to unpack

# However, we can use a single starred name in the target to match more
# generally.

a, *b = seq         # a = 1 and b = [2, 3, 4]
a*, b = seq         # a = [1, 2, 3] and b = 4
a, *b, c = seq      # a = 1, b = [2, 3] and c = 4
a, b, *c = seq      # a = 1, b = 2, c = [3, 4]

# This Sequence unpacking works for any type of sequence, not just lists.
a, *b, c = "spam"   # a = 's', b = ['p', 'a'] and c = 'm'

# Although extended sequence unpacking is flexible, some boundary cases are
# worth noting.
```

```
# First, the starred name may match just a single item, but is always
# assigned a list:
a, b, c, *d = seq
print(a, b, c, d)       # Output: 1 2 3 [4]

# Second, if there is nothing left to match the starred name, it is assigned an
# empty list, regardless of where it appears. In the following, a, b, c, and d
# have matched every item in the sequence, but Python assigns e an empty list
# instead of treating this as an error case:

a, b, c, d, *e = seq
print(a, b, c, d, e)    # Output: 1 2 3 4 []

a, b, *e, c, d = seq
print(a, b, c, d, e)    # Output: 1 2 3 4 []

# Finally, errors can still be triggered if there is more than one starred name,
# if there are too few values.

a, *b, c, *d = seq
# Output: SyntaxError: two starred expressions in assignment

# random MODULE:

    import random
    random.random()                  # Outputs random float point number
    random.randint(1, 10)            # Outputs a random integer in given range
    random.choice([1, 2, 3, 4])      # Selects an element randomly from the list
    random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
    suits = ['clubs', 'diamonds', 'spades', 'hearts']
    random.shuffle(suits)

# ------------------------------------------------------------------------------

# 6. DICTIONARIES

# MAPPING OPERATIONS:

    # Dictionaries are coded in curly braces and consist of a series of
    # “key: value” pairs.
    D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
    D['food']    # 'Spam'
    D['quantity'] += 1  # Mutable
    D   # {'food': 'Spam', 'color': 'pink', 'quantity': 5}

    # Creates new key-value pairs using assignment
    D = {}
    D['name'] = 'Bob'
    D['job'] = 'dev'
    D['age'] = 40
    D        # Output: {'age': 40, 'job': 'dev', 'name': 'Bob'}
```

```python
    # We can also make dictionaries by passing 'name = value' syntax to 'dict()'
    # function or by zipping together two sequences of keys and values obtained
    # at runtime.
    bob1 = dict(name='Bob', job='dev', age='40')
    bob1    # Output: {'age': 40, 'job': 'dev', 'name': 'Bob'}
    bob2 = dict(zip(['name', 'job', 'age'], ['Bob', 'dev', 40]))
    bob2    # Output: {'age': 40, 'job': 'dev', 'name': 'Bob'}
    # Notice how left to right order of dictionary keys is scrambled. Mappings
    # are not positionally ordered because we dont access dictionary values
    # by position but by keys.

# NESTING:

    rec = {
            'name': {'first': 'Bob', 'last': 'Smith'},
            'job': ['dev', 'mgr'],
            'age': 40.5
        }

    # All appropiate operations are valid on values of dictionary. For example,
    rec['job'].append('janitor')
    rec
    # Output: {'age': 40.5, 'job': ['dev', 'mgr', 'janitor'],
    #           'name': {'last': 'Smith', 'first': 'Bob'}}

    rec['exp']  # Output: KeyError: 'exp'

# SORTING KEYS:

    # We saw that the keys are printed in different order than that in which
    # they were created.
    D = {'a':1, 'b':2, 'c':3}
    D       # Output: {'a':1, 'c':3, 'b':2}

    # If we need to impose an ordering on dictionary's items, one common
    # solution is to grab a list of keys, sort that list and then step through
    # the dictionary using for loop.
    k = list(D.keys())          # ['a', 'b', 'c']
    k.sort()
    for key in k:
        print(key, '=', D[key])
    # Output: a = 1
    #         b = 2
    #         c = 3

    # This was a 3-step process. This could be done in 1 step as:
    for key in sorted(D):
        print(key, '=', D[key])

    # In above example, we saw - how to make a list of 'keys' in a dictionary:
    k = list(D.keys())
```

```python
    # Similarly, we can extract 'values' in a dictionary as:
    v = list(D.values())
    # We can also create a list of tuples of key-value pairs as:
    p = list(D.items())

# ADDING and DELETING items:

    # Using update():

    # It merges the keys and values of one dictionary into another,
    # blindly overwriting values of the same key:
    D1 = {'spam': 2, 'ham': 1, 'eggs': 3}
    D2 = {'toast':4, 'muffin':5}
    D1.update(D2)
    D1      # {'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}

    # Using pop():

    D.pop('muffin')         # 5
    D.pop('toast')          # 4
    D       # {'eggs': 3, 'ham': 1, 'spam': 2}

# COMPREHENSIONS:

    # One nice use of dictionary comprehensions is to invert dictionaries.

    country_to_capital = {
        'UK':'London',
        'Brazil':'Brazilia',
        'Morocco':'Rabat',
        'Sweden':'Stockholm'
    }

    capital_to_country = {
        capital:country for country, capital in country_to_capital.items()
    }

    # It is important to understand how this comprehension is working.
    # 'country_to_capital.items()' returns a tuple of key-value pairs.
    # 'country, capital' uses tuple unpacking on each tuple element and gets
    # the corresponding values.
    # 'capital:country' generates the inverted dictionary.

    # Note: If there are duplicate keys, while handling comprehensions (or
    # usually anytime) latter keys overwrite the former ones. Also, we shouldn't
    # use comprehensions alot and obfuscate the code. Readability of code is
    # still the main purpose. Because code is written once, but read again and
    # again.
# ------------------------------------------------------------------------------
```

```python
# 7. TUPLES

# INTRODUCTION:

    # The tuple object is roughly like a list that cannot be changed.

    T = (1, 2, 3, 4)
    len(T)          # Output: 4
    T + (5, 6)      # Output: (1, 2, 3, 4, 5, 6)
    T               # Output: (1, 2, 3, 4).  Because tuples are immutable.

    T[0]            # 1
    T.index(4)      # 3. Meaning: Element '4' is at what index?
    T.count(4)      # 1. Meaning: How many times does 4 appear?

    # Like Lists and Dictionaries, Tuples also support mixed types and nesting.
    # They cant be grown or shrunk as they are immutable.
    T.append(5)
    # Output: AttributeError: 'tuple' object has no attribute 'append'

    # Tuples can be formed from lists using tuple() method:
    T = tuple([1, 2, 3, 4])
    T   # (1, 2, 3, 4)

    # Some of the functions which can be used on tuples:
    # - len()
    # - max() and min()
    # - sum()
    # - sorted()

# SOME FACTS ABOUT TUPLES:

    # a) Tuples can't contain a single element.

    h = (391)
    h               # Output: 391
    type(h)         # Output: <class 'int'>

    # Here, the parentheses are considered as the precedence order operator.
    # Thus, 391 evaluates to an integer.

    # If we attempted to make tuple using tuple() function, we get TypeError.
    t = tuple((132))    # Or you can also try: t = tuple(132)
    # TypeError: 'int' object is not iterable

    # However there is a trick with which you can create a single element tuple,
    # i.e. by including a trailing comma.
    t = (391,)
    t           # Output: (391,)
    type(t)     # Output: <class 'tuple'>
```

```python
    # b) Parentheses can be omitted while defining tuples

    p = 1, 1, 1, 4, 9
    p           # Output: (1, 1, 1, 4, 9)

    # This feature is often useful when returning multiple values from the
    # function. Following is an example function which does nothing by itself
    # and is made just to demonstrate how to return multiple values from func.

    def minmax(items):
        return min(items), max(items)

    minmax([83, 33, 84, 32, 85, 31, 86])
    # Output: (31, 86)

    # We can also collect the multiple return values because of tuple unpacking:
    lower, upper = minmax([83, 33, 84, 32, 31, 86])
    lower       # Output: 31
    upper       # Output: 86

    # This tuple unpacking leads to Python's famous 1 line swap statement:
    x, y = y, x

# WHY TUPLES?:

    # If we provide collection of objects to any program or method, there is a
    # possibility that it might get changed. But passing a tuple won't let that
    # happen. It is like passing 'const array' or 'const string' in C/C++ rather
    # than just array or string. In short, tuples provide a sort of integrity
    # constraint that is convenient in programs larger than we usually write.

# ------------------------------------------------------------------------------

# 8. SETS

# INTRODUCTION:

    # An unordered collection of unique and immutable objects that supports
    # operations corresponding to mathematical set theory. Sets are iterable,
    # can grow and shrink on demand, and may contain a variety of object types.
    x = set('spammer')
    x                       # Output: {'m', 's', 'r', 'a', 'p', 'e'}
    y = {'h', 'a', 'm'}     # Creating a set with literals.

    x & y                   # Intersection
    # Output: {'m', 'a'}
    x | y                   # Union
    # Output: {'m', 's', 'h', 'r', 'a', 'p', 'e'}
    x - y                   # Set Difference
    # Output: {'s', 'e', 'p', 'r'}
```

```python
    x ^ y                   # Symmetric Difference
    # Output: {'s', 'h', 'r', 'p', 'e'}
    x > y                   # Is X superset of Y?
    # Output: False
    x < y                   # Is X subset of Y?
    # Output: False

# SET FUNCTIONS:

# a) Adding elements to a set

    # Using add():

    # Only one element at a time can be added to the set by using add() method,
    # loops are used to add multiple elements at a time with the use of add()
    # method.

    # Note: Lists cannot be added into sets, whereas tuples can be added.
    # Reason (advanced): Because lists are mutable, hence not hashable;
    #                    whereas tuples are immutable, and hence hashable.
    s = set()    # s = {} doesn't work as this will create 's' as empty dictionary.
    s.add(8)
    s.add(9)
    s.add((5, 6))
    s   # {8, 9, (5, 6)}

    # Using update():

    # For addition of two or more elements update() method is used.
    # The update() method accepts lists, strings, tuples as well as other sets
    # as its arguments. In all of these cases, duplicate elements are avoided.
    set1 = set([ 4, 5, (6, 7)])
    set1.update([10, 11])
    set1    # {10, 11, 4, 5, (6, 7)}

# b) Accessing a set

    # Set items cannot be accessed by referring to an index, since sets are
    # unordered the items has no index. But you can loop through the set items
    # using a for loop, or ask if a specified value is present in a set, by
    # using the 'in' keyword.
    for i in set1:
        print(i, end=" ")       # Output: 4 5 (6, 7) 10 11

    print(11 in set1)      # Output: True

# c) Removing elements from set

    # Using remove() or discard():
```

```
    # Elements can be removed from the Set by using built-in remove() function
    # but a KeyError arises if element doesn't exist in the set.
    # To remove elements from a set without KeyError, use discard(), if the
    # element doesn't exist in the set, it remains unchanged.
    set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
    set1.remove(5)
    set1.remove(6)
    set1.discard(8)
    set1.discard(9)
    set1        # {1, 2, 3, 4, 7, 10, 11, 12}

    # Using pop():

    # For an unodered set, there's no such way to determine which element is
    # popped by using the pop() function.
    set1.pop()  # 1
    set1        # {2, 3, 4, 7, 10, 11, 12}

    # Note (advanced): Internally, set uses BSTs. So pop() removes the elements
    # in order from leaf node to root. Hence pop removes minimum element in this
    # case. However, pop() will remove minimum element in a set - should not be
    # generalized.

    # Using clear():
    # Used to clear entire set and convert it into a null set.

# USAGE:

    # Sets are useful for tasks such as - filtering out duplicates,
    # order-neutrality, equality tests without sorting, etc.
    list(set([1, 2, 1, 3, 1, 2]))       # Filtering out duplicates
    # Output: [1, 2, 3]
    set('spam') == set('maps')          # Order-neutrality Equality Test
    # Output: True

# ---------------------------------------------------------------------------

# 9. NONE

    # This is a special constant used to denote a null value or a void.
    # Note: 0 or any empty container (e.g empty list) do not compute to None.
    0 == None       # Output: False
    [] == None      # Output: False

    # It is an object of its own datatype – NoneType.
    a = None
    a == None       # Output: True

# ---------------------------------------------------------------------------
```

```python
########################### TOPIC 3 - DYNAMIC TYPING ###########################

# 1. MISSING DECLARATIONS:

    # When we type a = 3 in an interactive session or program file,
    # for instance, how does Python know that 'a' should stand for
    # an integer? For that matter, how does Python know what 'a' is at all?

    # Once you start asking such questions, you’ve crossed over into
    # the domain of Python’s dynamic typing model. In Python, types are
    # determined automatically at runtime, not in response to declarations
    # in your code. This means that you never declare variables ahead of time.

    # When we say, a = 3, Python will perform three distinct steps to carry out
    # the request.
    # 1. Create an object to represent the value 3.
    # 2. Create the variable a, if it does not yet exist.
    # 3. Link the variable a to the new object 3.

    # These links from variables to objects are called references in Python —
    # that is, a reference is a kind of association, implemented as a pointer
    # in memory. Readers with a background in C may find Python references
    # similar to C pointers (memory addresses). In fact, references are
    # implemented as pointers, and they often serve the same roles, especially
    # with objects that can be changed in-place (more on this later). However,
    # because references are always automatically dereferenced when used, you
    # can never actually do anything useful with a reference itself; this is a
    # feature that eliminates a vast category of C bugs. You can think of Python
    # references as C 'void*' pointers, which are automatically followed
    # whenever used.

    a = 3                   # a is an int
    a = "sagar"             # a is a string now
    a = 3.142               # a is a float now

    # The above series of statements are possible because -  data type lives
    # with objects and not with references. Variable names have no type - they
    # can be referenced to any object type. Objects, on the other hand, know
    # what type they are — each object contains a header field that tags the
    # object with its type. The integer object 3, for example, will contain the
    # value 3, plus a designator that tells Python that the object is an integer.

    # This leads to another question - when we reassign a variable, what happens
    # to the value it was previously referencing? For example, after the
    # following statements, what happens to the object 3?

    a = 3
    a = "Spam"
```

```python
    # The answer is that - In Python, whenever a name is assigned to a new object,
    # the space held by the prior object is reclaimed. This automatic
    # reclamation of objects' space is known as garbage collection.

    # Internally, Python accomplishes this feat by keeping a counter in every
    # object that keeps track of the number of references currently pointing to
    # that object. As soon as (and exactly when) this counter drops to zero,
    # the object's memory space is automatically reclaimed. In the preceding
    # listing, we're assuming that each time 'a' is assigned to a new object,
    # the prior object's reference counter drops to zero, causing it to be
    # reclaimed.

    # The most immediately tangible benefit of garbage collection is that it
    # means you can use objects liberally without ever needing to free up space
    # in your script. Python will clean up unused space for you as your program
    # runs. In practice, this eliminates a substantial amount of bookkeeping
    # code required in lower-level languages such as C and C++.

# ------------------------------------------------------------------------------

# 2. SHARED REFERENCES:

    a = 3
    b = a
    a = 4

    # In first step, integer object '3' is created and a is reference of
    # that object. In next step, another reference b is linked with the same
    # object '3'. So in memory there is still only one object with value 3 and
    # 'a' and 'b' are shared references. In last step, new object '4' is created
    # and a refer to this object. So b still refers to object '3'.

    # This is the normal behaviour of variables in most programming languages.
    # And as long as we are dealing with 'immutable' data types, we need not
    # worry about shared references and we may consider references as normal
    # variables. However, there are objects and operations that perform in-place
    # object changes. For instance, an assignment to an offset in a list
    # actually changes the list object itself in-place, rather than generating
    # a brand new list object. For objects that support such in-place changes,
    # you need to be more aware of shared references, since a change from one
    # name may impact others.

    L1 = [2, 3, 4]          # A mutable object
    L2 = L1                 # Making a new reference to same object
    L1[0] = 1               # Changing L1 in-place

    L1                      # [1, 3, 4] - L1 has changed.
    L2                      # [1, 3, 4] - L2 has also changed!

    # If we don't want such behaviour, we must create a copy of the object
    # instead of creating references.
```

```
    L1 = [2, 3, 4]
    L2 = L1[:]           # Making a copy of L1
    L1[0] = 1

    L1                   # [1, 3, 4] - L1 has changed.
    L2                   # [2, 3, 4] - L2 remains unchanged.

    # We can check if two references refer to same memory location or not using
    # 'is' and 'is not' operators.

    a = 'Spam'
    b = a
    a == b   # True i.e. contents of objects are same whom a and b are referring to
.
    a is b   # True i.e. They refer to same memory

    a = [1, 2]
    b = a
    a == b   # True i.e. contents of objects are same whom a and b are referring to
    a is b   # True i.e. They refer to same memory.

    # Note one more difference in 'mutable' and 'immutable' objects,
    # in following example.

    a = 'Spam'
    b = 'Spam'
    a == b      # True
    a is b      # True

    a = [1, 2]
    b = [1, 2]
    a == b      # True
    a is b      # False

# --------------------------------------------------------------------------------

# 3. COPIES ARE SHALLOW

# We first create a nested list for our demonstration.
a = [[1, 2], [3, 4]]

# We create a copy and assign it to b.
b = a[:]

# We can confirm 'a' and 'b' are distinct objects as:
a is b              # Output: False
a == b              # Output: True

# Now replace a[0] by new list.
a[0] = [8, 9]
```

```python
a               # Output: [[8, 9], [3, 4]]
b               # Output: [[1, 2], [3, 4]]

# Till now you must be convinced that a and b are distinct objects and
# everything just works fine. But this is what happens when you do this:
a[1].append(5)
a               # Output: [[8, 9], [3, 4, 5]]
b               # Output: [[1, 2], [3, 4, 5]]

# Even b[1] is also changed. The reason is, b = a[:] creates the copy of
# the top-most reference list. What I mean by this is - Imagine a nested list
# as, the list of references where each reference points to the corresponding
# element. So even though 'a' and 'b' are distinct lists of referenecs, those
# references lead you to the same list elements. You can confirm this as:

a[0] is b[0]    # Output: False
# This is because we have created a new list [8, 9] and a[0] now stores the
# reference of this newly created list. Whereas,
a[1] is b[1]    # Output: True

# Solution (advanced): If you don't want this issue, use deepcopy() method
# available in 'copy' module.

# --------------------------------------------------------------------------------

# 4. REPETITIONS ARE SHALLOW

# Repetition repeats the reference without copying the value.

s = [[-1, +1]]*4
s       # Output: [[-1, 1], [-1, 1], [-1, 1], [-1, 1]]

# Let's say we do append() on third element.
s[2].append(0)
s       # Output: [[-1, 1, 0], [-1, 1, 0], [-1, 1, 0], [-1, 1, 0]]

# All of the sublists are altered. This might seem little annoying at first,
# but the python implementation was designed in this way (i.e. by working with
# references) to optimize it's execution to the best possible level.

# Solution: Don't use repetition. Use for loop and run it 4 times appending
# [-1, 1] each time in the list.

# --------------------------------------------------------------------------------
```

```
######################### TOPIC 4 - CONTROL STATEMENTS #######################

# 1. IF statement:

    x = int(input("Please enter an integer: "))
    if x < 0:
        x = 0
        print("Negative numbers not allowed. Hence changed to 0.")
    elif x == 0:
        print("You entered zero!")
    elif x < 100:
        print("Your number is less than 100.")
    else
        print("Your number is huge!")

    # There can be zero or more elif parts, and the else part is optional.
    # The keyword ‘elif’ is short for ‘else if’, and is useful to avoid
    # excessive indentation. An if … elif … elif … sequence is a substitute
    # for the switch or case statements found in other languages.

# ----------------------------------------------------------------------------

# 2. WHILE loop:

    # Let's write the code for 'Fibonaci numbers' to demonstrate few points:

    a, b = 0, 1
    while a < 10:
        print(a, end = ' ')
        a, b = b, a+b
    print()

    # This example introduces some new features:

    # The first line contains a multiple assignment: the variables a and b
    # simultaneously get the new values 0 and 1. On the second last line this
    # is used again, demonstrating that the expressions on the right-hand side
    # are all evaluated first before any of the assignments take place.
    # The right-hand side expressions are evaluated from the left to the right.

    # The while loop executes as long as the condition (here: a < 10) remains
    # true. In Python, like in C, any non-zero integer value is true; zero is
    # false. The condition may also be a string or list value, in fact any
    # sequence; anything with a non-zero length is true, empty sequences are
    # false.

    # print() is used to go to the newline.

# ----------------------------------------------------------------------------
```

```python
# 3. FOR loop:

    # The for statement in Python differs a bit from what you may be used to
    # in C. Rather than always iterating over an arithmetic progression of
    # numbers or giving the user the ability to define both the iteration step
    # and halting condition, Python’s for statement iterates over the items of
    # any sequence (a list or a string), in the order that they appear in the
    # sequence. For example (no pun intended):

    # Measure some strings:
    words = ['cat', 'window', 'defenestrate']
    for w in words:
        print(w, len(w))

    # If you do need to iterate over a sequence of numbers, the built-in
    # function range() comes in handy. To iterate over the indices of a
    # sequence, you can combine range() and len() as follows:

    a = ['Mary', 'had', 'a', 'little', 'lamb']
    for i in range(len(a)):
        print(i, a[i])

    # In most such cases, however, it is convenient to use the enumerate()
    # function.
    for i, v in enumerate(a):
        print(i, v)                 # Prints same results

    # A strange thing happens if you just print a range:
    print(range(10))
    # In many ways the object returned by range() behaves as if it is a list,
    # but in fact it isn’t. It returns an object of 'range' class. Range object
    # returns the successive items of the desired sequence when you iterate
    # over it, but it doesn’t really make the list, thus saving space.
    # Thus, to get the list from range(), we have an easy solution:
    list(range(4))

    # When looping through dictionaries, the key and corresponding value can be
    # retrieved at the same time using the items() method.
    knights = {'gallahad': 'the pure', 'robin': 'the brave'}
    for k, v in knights.items():
        print(k, ': ', v)

    # To loop over two or more sequences at the same time, the entries can be
    # paired with the zip() function.
    questions = ['name', 'quest', 'favorite color']
    answers = ['lancelot', 'the holy grail', 'blue']
    for q, a in zip(questions, answers):
        print('What is your {0}?  It is {1}.'.format(q, a))
```

```python
    # To loop over a sequence in reverse, first specify the sequence in a
    # forward direction and then call the reversed() function.
    for i in reversed(range(1, 10, 2)):
        print(i, end=' ')
    print()

    # To loop over a sequence in sorted order, use the sorted() function which
    # returns a new sorted list while leaving the source unaltered.
    basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
    for f in sorted(set(basket)):
        print(f)

# -----------------------------------------------------------------------------

# 4. BREAK and CONTINUE statements:

    # The break statement causes an immediate exit from a loop. Because the code
    # that follows it in the loop is not executed if the break is reached, you
    # can also sometimes avoid nesting by including a break.

    while True:
        name = input('Enter name:')
        if name == 'stop': break
        age = input('Enter age: ')
        print('Hello', name, '=>', int(age) ** 2)

    # The continue statement causes an immediate jump to the top of a loop.
    # The next example uses continue to skip odd numbers. This code prints
    # all even numbers less than 10 and greater than or equal to 0.
    # Remember, 0 means false and % is the remainder of division operator,
    # so this loop counts down to 0, skipping numbers that aren’t multiples
    # of 2 (it prints 8 6 4 2 0):

    x = 10
    while x:
        x = x-1                     # Or, x -= 1
        if x % 2 != 0: continue     # Odd? -- skip print
        print(x, end=' ')

# -----------------------------------------------------------------------------

# 5. ELSE clause with loops:

    # Loop statements may have an else clause; it is executed when the
    # loop terminates through exhaustion of the iterable (with for) or
    # when the condition becomes false (with while), but not when the
    # loop is terminated by a break statement.
```

```
    for n in range(2, 10):
        for x in range(2, n):
            if n % x == 0:
                print(n, 'equals', x, '*', n//x)
                break
        else:
            # loop fell through without finding a factor
            print(n, 'is a prime number')

    # Output:
    # 2 is a prime number
    # 3 is a prime number
    # 4 equals 2 * 2
    # 5 is a prime number
    # 6 equals 2 * 3
    # 7 is a prime number
    # 8 equals 2 * 4
    # 9 equals 3 * 3

# --------------------------------------------------------------------------------

# 6. PASS Statement:

    # The pass statement does nothing. It can be used when a statement
    # is required syntactically but the program requires no action.

    # This is commonly used for creating minimal classes:
    class MyEmptyClass:
        pass

    # Another place pass can be used is as a place-holder for a function
    # or conditional body when you are working on new code, allowing you
    # to keep thinking at a more abstract level.
    def initlog(*args):
        pass

# --------------------------------------------------------------------------------
```

```python
############################## TOPIC 5 - FUNCTIONS ##############################

# 1. WHY USE FUNCTIONS?
    # - Maximizing code reuse and minimizing redundancy
    # - Procedural decomposition

# ------------------------------------------------------------------------------

# 2. DEF statement

    # The def statement creates a function object and assigns it to a name.
    # Its general format is as follows:
    def <name>(arg1, arg2,... argN):
        <statements>
        return <value>
    # Technically, a function without a return statement returns the None object
    # automatically, but this return value is usually ignored.

    # Here are some basic points regarding functions:

    # a) In Python, 'def' is an executable statement.

    # The Python 'def' is a true executable statement: when it runs, it creates
    # a new function object and assigns it to a name. We will see exactly -
    # what does it really mean by - 'def is an executable statement' when we
    # will see about 'Default Arguments' later.

    # Because it's a statement, a 'def' can appear anywhere — even nested in
    # other statements. For example:
    if test:
        def func(): # Define func this way
        ...
    else:
        def func(): # Or else this way
        ...
    ...
    func() # Call the version built during runtime

    # Because the definition of a function is decided at runtime, there's
    # nothing special about the function name. What's important is the object
    # to which it refers:
    othername = func     # Assign function object
    othername()          # Equivalent of calling func again

    # b) All functions in Python are polymorphic.

    # Functions work on arbitrary types, as long as they support the expected
    # object interface.
    def intersect(seq1, seq2):
        res = []                  # Start empty
        for x in seq1:            # Scan seq1
```

```python
        if x in seq2:       # Common item?
            res.append(x)   # Add to end
    return res

s1 = "SPAM"
s2 = "SCAM"
x = intersect(s1, s2)               # ['S', 'A', 'M']
x = intersect([1, 2, 3], (1, 4))    # [1]
# This time, we passed in different types of objects to our function —
# a list and a tuple (mixed types) — and it still picked out the common
# items. Because you don't have to specify the types of arguments ahead of
# time, the intersect function happily iterates through any kind of sequence
# objects you send it, as long as they support the expected interfaces.

# --------------------------------------------------------------------------------

# 3. ARGUMENT PASSING

    # Consider the following situation:

    m = [9, 15, 24]

    def modify(k):
        k.append(39)
        print("k = ", k)

    modify(m)           # Output: k = [9, 15, 24, 39]
    print(m)            # Output: [9, 15, 24, 39]

    # Thus, we can see even 'm' is changed!

    # Now consider one more situation:

    m = "Sagar"

    def modify(k):
        k = k[:2] + 'm' + k[-2:]
        print(k)

    modify(m)           # Output: Samar
    print(m)            # Output: Sagar

    # In this case, 'm' remains unchanged.

    # From above two situations, we can draw two conclusions:
    # a) Immutable arguments are effectively passed 'by value'.
    # b) Mutable arguments are effectively passed 'by reference'.
```

```python
    # Finally, we consider one last situation:

    def f(d):
        return d            # This function simply returns whatever it gets

    c = [9, 10]
    e = f(c)
    e is c          # Output: True

    c = "Sagar"
    e = f(c)
    e is c          # Output: True

    # In this case, same objects were returned and no new copies were created.

    # So one last conclusion which we can draw from above situation is:
    # c) 'return' works on 'return by reference' instead of 'return by value',
    #    irrespective of it is returning mutable object or immutable object.

# ----------------------------------------------------------------------------

# 4. DEFAULT, POSITIONAL and KEYWORD ARGUMENTS

# INTRODUCTION:

    # Consider the following function:

    def banner(message, border = '-'):
        line = border*len(message)
        print(line)
        print(message)
        print(line)

    banner("Norwegian Blue")
    # 1 positional argument, and other argument taken by default as '-'

    banner("Sun, Moon & Stars", '*')
    # 2 positional arguments which must be provided in order

    banner("Sun, Moon & Stars", border='#')
    # 1 positional argument and 1 keyword argument.
    # keyword arguments should be specified only after the positional arguments
    # have been specified in order.

    banner(border='+', message="Hello, from World!")
    # 2 keyword arguments. Now order is not important.
```

```python
# 'DEF' is an EXECUTABLE STATEMENT:

    # Let's see what we meant earlier by - 'def' is an executable statement.

    # Consider the following function which shows current time.

    import time
    def show_time(arg = time.ctime()):
        print(arg)

    show_time()         # Output: Sun Dec 29 01:58:32 2019

    # Try executing same function, few seconds or minutes later.
    show_time()         # Output: Sun Dec 29 01:58:32 2019
    show_time()         # Output: Sun Dec 29 01:58:32 2019

    # We may see that show_time() is suggesting that - 'time has stopped
    # progressing' - which we know is definitely not the case. So how can
    # we explain the output of these calls? By giving following explanation:

    # The 'def' statement is executed. And when 'def' was being executed,
    # default argument took the value of the current time and was never
    # evaluated again. So we can say - 'def' is executable statement, and is
    # not like a blue-print which classes provide for objects, and default
    # arguments are evuaulated only once.

# DEFAULT ARGS SHOULD BE IMMUTABLE:

    def add_spam (menu = []):
        menu.append('spam')
        return menu

    breakfast = ['bacon', 'eggs']
    add_spam(breakfast)
    breakfast           # Output: ['bacon', 'eggs', 'spam']

    lunch = ['baked beans']
    add_spam(lunch)
    lunch               # Output: ['baked beans', 'spam']

    # By this moment, our function works as expected. But look what happens
    # when we rely just on default arguments.

    add_spam()          # returns ['spam']
    add_spam()          # returns ['spam', 'spam']
    add_spam()          # returns ['spam', 'spam', 'spam']

    # This is completely opposite to the intention of programmer of keeping
    # default argument as empty list. Programmer wanted that everytime,
    # add_spam() is called with no arguments, it should return just ['spam'].
```

```python
    # The solution to this problem is simple, but not very obvious:
    # 'Always use immutable objects as default arguments.'
    # So we modify the above function as:

    def add_spam (menu = None):
        if menu is None:
            menu = []
        menu.append('spam')
        return menu

    add_spam()          # returns ['spam']
    add_spam()          # returns ['spam']

    # Now everything looks fine.
# --------------------------------------------------------------------------------

# 5. VARARGS

    # Functions can use special arguments preceded with one or two * characters
    # to collect an arbitrary number of extra arguments. (a feature called as
    # 'variadic arguments' or 'varargs')

    # Callers can also use the * syntax to unpack argument collections into
    # discrete, separate arguments.

    # Before we explain the above two points, first look at the syntax:

    # a) As a caller:
    # func(*sequence)   Pass all objects in sequence as individual positional
    #                   arguments
    # func(**dict)      Pass all key-value pairs in dict as individual keyword
    #                   arguments

    # b) At function definition:
    # def func(*name)   Matches and collects remaining positional arguments in
    #                   a tuple
    # def func(**name)  Matches and collects remaining keyword arguments in
    #                   a dictionary

    # Now let's look at what they mean:

    # A function can be called with an arbitrary number of arguments.
    # These arguments will be wrapped up in a tuple. Before the variable number
    # of arguments, zero or more normal arguments may occur. Any formal
    # parameters which occur after the *args parameter are 'keyword-only'
    # arguments, meaning that they can only be used as keywords rather than
    # positional arguments. For example:

    def concat(*args, sep = '/'):
        return sep.join(args)
```

```python
    concat("earth", "mars", "venus")
    # returns 'earth/mars/venus'
    concat("earth", "mars", "venus", sep=".")
    # returns 'earth.mars.venus'

    # The reverse situation occurs when the arguments are already in a list or
    # tuple but need to be unpacked for a function call requiring separate
    # positional arguments. For instance, the built-in range() function expects
    # separate start and stop arguments. If they are not available separately,
    # write the function call with the * operator to unpack the arguments out
    # of a list or tuple:

    list(range(3, 6))            # normal call with separate arguments
    # returns [3, 4, 5]

    args = [3, 6]
    list(range(*args))           # call with arguments unpacked from a list
    # returns [3, 4, 5]

    # In the same fashion, dictionaries can deliver keyword arguments with the
    # ** operator:

    def parrot(voltage, state='a stiff', action='voom'):
        print("-- This parrot wouldn't", action, end=' ')
        print("if you put", voltage, "volts through it.", end=' ')
        print("E's", state, "!")

    d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
    parrot(**d)

    # Output: '-- This parrot wouldn't VOOM if you put four million volts'
    #         'through it. E's bleedin' demised !'

# -------------------------------------------------------------------------------

# 6. LAMBDA EXPRESSIONS

# LAMBDA BASICS:

    # Besides the def statement, Python also provides an expression form that
    # generates function objects. It's called 'Lambda expression'. Like def,
    # this expression creates a function to be called later, but it returns the
    # function instead of assigning it to a name. This is why lambdas are
    # sometimes known as anonymous (i.e., unnamed) functions.

    # The lambda’s general form is the keyword lambda, followed by one or more
    # arguments (exactly like the arguments list you enclose in parentheses in
    # a def header), followed by an expression after a colon:

    lambda argument1, argument2,... argumentN :expression using arguments
```

```python
    # But there are a few differences that make lambdas useful in specialized
    # roles:

    # a) lambda is an expression, not a statement.

    # Because of this, a lambda can appear in places a def is not allowed by
    # Python’s syntax—inside a list literal or a function call’s arguments.

    # b) lambda’s body is a single expression, not a block of statements.

    # Because it is limited to an expression, a lambda is less general than a
    # def — you can only squeeze so much logic into a lambda body without using
    # statements such as if. This is by design, to limit program nesting: lambda
    # is designed for coding simple functions and def handles larger tasks.

    # Consider the following list:
    words = ['strawberry', 'fig', 'apple', 'maple', 'banana', 'cherry']
    # Now we need to sort this list such that rhyming words are grouped
    # together. The idea is to sort words in reverse.
    sorted(words, key=lambda word: word[::-1])

# ------------------------------------------------------------------------------

# 7. GENERATOR FUNCTIONS

# Now before we look at 'Generator' functions, let's look first at -
# Iterables and Iterators.

# ITERABLES AND ITERATORS:

    # In the previous section, we mentioned that for loop can work on any
    # sequence type including lists, tuples, strings, etc. Actually, the for
    # for loop turns out to be even more generic than this — it works on any
    # iterable object.

    # One of the easiest ways to understand what this means is to look at how
    # it works with a built-in type such as the file. File objects have a
    # method called readline, which reads one line of text from a file at a
    # time — each time we call the readline method, we advance to the next line.
    # At the end of the file, an empty string is returned, which we can detect
    # to break out of the loop:

    f = open('script1.py')
    f.readline()                # 'import sys\n'
    f.readline()                # 'print(sys.path)\n'
    f.readline()                # 'x = 2\n'
    f.readline()                # 'print(2 ** 33)\n'
    f.readline()                # ''

    # However, files also have a method named __next__ that has a nearly
    # identical effect — it returns the next line from a file each time it is
```

```python
# called. The only noticeable difference is that __next__ raises a built-in
# StopIteration exception at end-of-file instead of returning an empty
# string:

f = open('script1.py')
f.__next__              # 'import sys\n'
f.__next__              # 'print(sys.path)\n'
f.__next__              # 'x = 2\n'
f.__next__              # 'print(2 ** 33)\n'
f.__next__              # StopIteration

# Any object with a __next__ method to advance to a next result, which
# raises StopIteration at the end of the series of results, is considered
# iterable in Python.

# The best way to read a text file line by line today is to allow the for
# loop to automatically call __next__ to advance to the next line on each
# iteration. The file object's iterator will do the work of automatically
# loading lines as you go.

for line in open('script1.py'):
    print(line.upper(), end='')

# Output: IMPORT SYS
#         PRINT(SYS.PATH)
#         X = 2
#         PRINT(2 ** 33)

# Notice that the print uses end='' here to suppress adding a \n, because
# line strings already have one.

# The iterators work quicker and consume less memory compared to
# non-iterator equivalent code. For eg. it's also possible to read a file
# line by line with a while loop:

f = open('script1.py')
while True:
    line = f.readline()
    if not line: break
    print(line.upper(), end='')

# However, this may run slower than the iterator-based for loop version,
# because iterators run at C language speed inside Python, whereas the
# while loop version runs Python byte code through the Python virtual
# machine.

# Iterable objects such as strings, lists, sets, etc. can be passed to the
# built-in iter() function to get an iterator. Iterator is like the pointer
# which points to the elements of the iterable object. Iterator object can
# be passed to the built-in function next() to fetch the next item.
```

```python
    # Syntax: iterator = iter(iterable)
    #         item = next(iterator)

    iterable = ['Spring', 'Summer', 'Autumn', 'Winter']
    iterator = iter(iterable)
    next(iterator)            # Output: Spring

    # The above call is same as calling next() as below:
    iterator.__next__()       # Output: Summer

    next(iterator)            # Output: Autumn
    next(iterator)            # Output: Winter
    next(iterator)            # Output: StopIteration

    # When the for loop begins, it obtains an iterator from the iterable object
    # by passing it to the iter built-in function; the object returned by iter()
    # has the required next method. This initial step is not required for files,
    # because a file object is its own iterator. That is, files have their own
    # __next__ method.

    f = open('script1.py')
    iter(f) is f              # Output: True
    L = [1, 2, 3]
    iter(L) is L              # Output: False

# GENERATOR FUNCTIONS:

    # Before defining 'generators', let us first see some features of these
    # functions:

    # a) All generators are iterators.
    # b) Generators are lazily evaluated. This means - the next value in the
    #    sequence is computed on demand. We will see how.
    # c) These are used when we need to model infinite sequences or data streams
    #    with no definite end. For eg. Sensor readings which keep on coming
    #    from sensors as long as they are on, math sequences and series, large
    #    file processing, etc.

    # Generators are defined as python function which uses 'yield' keyword
    # atleast once in it's definition.

    # To understand generators, lets write the most basic generator function.

    def gen123():
        yield 1
        yield 2
        yield 3

    # We call generators on iterator objects. We create iterators as follows:

    g = gen123()              # Here 'g' is generator object.
```

```python
    # Since generators are like iterators, they can be passed to next() method.

    next(g)         # returns 1
    next(g)         # returns 2
    next(g)         # returns 3
    next(g)         # Error: StopIteration

    # Instead of using next() again and again, we can use for loop.

    for v in gen123():
        print(v)

    # Every generator object formed using same generator function has separate
    # address. This is due to the fact that each iterator can be advanced
    # indenpendently.
    h = gen123()
    i = gen123()
    h               # returns some address of generator object
    i               # returns some address of generator object
    h is i          # False

    # Let us write a generator funciton with more statements. Following
    # generator function produces lucas series on demand.

    def lucas():
        yield 2
        a = 2
        b = 1
        while True:
            yield b
            a, b = b, a+b

    # But what is the benefit of using generators?

    # Let's say we want to calculate the sum of first 1 million squares.
    # What we can do is - create a list of 1 million squares and then add
    # them using for loop. This will take around 400 MB memory and considerable
    # time. But what we can do is this -

    sum(x*x for x in range(1, 1000001))

    # So if we use generator function like range(), it will generate the
    # next value each time and sum will keep on adding it. Thus the memory usage
    # in this case will be insignificant. This is the benefit of lazy evaluation.

# ------------------------------------------------------------------------------
```

```python
# 8. SCOPE

# LEGB RULE:

    # In Python, names are resolved using LEGB rule. Let's see the scopes first
    # and then we shall see each one of them in detail.

    # a) L - Local: These are the names assigned in any way within a function
    #        (def or lambda) and not declared 'global' in that function.
    # b) E - Enclosing: Names in the local scope of any and all enclosing
    #        functions, from inner to outer. (Think of this as - Nested scope)
    # c) G - Global: Names assigned at the top-level of the module file or
    #        declared global within the file.
    # d) B - Built-in: Names preassigned in the built-in names module.

# BUILT-IN scope:

    # Built-in scope is bit simpler than what you may think. The built-in scope
    # is implemented as a standard library module named builtins, but that name
    # itself is not placed in the built-in scope, so you have to import it in
    # order to inspect it. Once you do, you can run a dir call to see which
    # names are predefined.

    import builtins
    dir(builtins)       # Output: Returns a list of built-in words.

    # The names in this list constitute the built-in scope in Python; roughly
    # the first half are built-in exceptions, and the second half are built-in
    # functions. Also in this list are the special names None, True, and False,
    # though they are treated as reserved words. Because Python automatically
    # searches this module last in its LEGB lookup, you get all the names in
    # this list 'for free' i.e., you can use them without importing any modules.

    # Thus, there are really two ways to refer to a built-in function — by
    # taking advantage of the LEGB rule, or by manually importing the builtins
    # module:

    zip                 # The normal way
    # Output: <class 'zip'>

    import builtins     # The hard way
    builtins.zip
    # Output: <class 'zip'>

    # The second approach is sometimes useful when local scope overrides
    # variables of same name in both - the built-in scope and global scope.

    # A function can, for instance, create a local variable called open by
    # assigning to it:
```

```
    def hider():
        open = 'spam'           # Local variable, hides built-in
        ...
        f = open('data.txt')    # This won't open a file now in this scope!

    # However, this will hide the built-in function called open that lives in
    # the built-in (outer) scope. It's also usually a bug, and a nasty one at
    # that, because Python will not issue a warning message about it.

# GLOBAL scope:

    # Variables which are declared with the 'global' keyword lie in the
    # global scope. 'global' allows us to change names that live outside a
    # def at the top level of a module file. For example:

    X = 88
    def func():
        global X
        X = 99      # This changes X globally
    func()          # Call the function
    print(X)        # Prints 99.

    # But we should try to avoid using 'global' as much as possible. Changing
    # globals can lead to well-known software engineering problems: because the
    # variables' values are dependent on the order of calls to arbitrarily
    # distant functions, programs can become difficult to debug. For example:

    X = 99

    def func1():
        global X
        X = 88

    def func2():
        global X
        X = 77

    # What will the value of X be here?
    # Really, that question has no meaning unless it's qualified with a point
    # of reference in time—the value of X is timing-dependent, as it depends on
    # which function was called last (something we can't tell from this file
    # alone).

    # So the basic doubt would come - then
    # What's the need of 'global' variables?
    # When is it actually useful?

    # One of the possible answers could be - during multithreading, networking,
    # etc. Multithreading runs function calls in parallel with the rest of the
    # program. Because all threaded functions run in the same process,
    # global scopes often serve as shared memory between them. Threading is
```

```
    # commonly used for long-running tasks in GUIs, to implement nonblocking
    # operations in general and to leverage CPU capacity.

    # For now, though, especially if you are relatively new to programming,
    # avoid the temptation to use globals whenever you can — try to communicate
    # with passed-in arguments and return values instead.

# NESTED/ENCLOSING scope:

    # Before talking about 'enclosing scope', we shall first see - how we nest
    # functions and how nested functions are called. For the sake of explaining
    # the flow of function calls, let us take a fairly simple example:

    def f1():
        X = 88
        def f2():           # Define f2 inside f1
            print(X)        # f2 remembers the value X = 88
        return f2           # Return f2 but don't call it

    action = f1()           # f1 returns f2 to action.
    action()                # This literally calls f2() and prints 88.

    # Since after f1() is called and return statement is executed, the scope of
    # variable X ends. However, calling action() allows us to call f2() and we
    # get our required result. The overall effect of this feels as if - f2()
    # has trapped the value of X or has remembered it.

    # This idea of 'rememberance of the value' will get clear in the following
    # concept of 'factory functions'.

    # 'Factory functions' refers to a function object that remembers values in
    # enclosing scopes regardless of whether those scopes are still present in
    # the memory. Although classes are usually best at remembering state,
    # because they make it explicit with attribute assignments, such functions
    # provide an alternative when things to remember are very small and we dont
    # want to create classes for every trivial thing. For example:

    def power(N):
        def action(X):
            return X**N
        return action

    square = power(2)
    square
    # Output: <function power.<locals>.action at 0x000001725BB94E58>

    square(3)       # Output: 9
    square(4)       # Output: 16
```

```python
# The most unusual part of this is that the nested function remembers
# integer 2, i.e. the value of the variable N in power, even though power
# has returned and exited by the time we call action. In effect, N from the
# enclosing local scope is retained as state information attached to action,
# and we get back its argument squared.

# To make things more clearer, let's call power function again.
cube = power(3)
cube(3)         # Output: 27
# Now let's see what happened to our square function.
square(3)       # Output: 9

# This works because each call to a factory function like this gets its own
# set of state information. In our case, the function we assign to name cube
# remembers 3, and square remembers 2, because each has its own state
# information retained by the variable N in power.

# This is an advanced technique that you're unlikely to see very often in
# most code, except among programmers with backgrounds in functional
# programming languages. On the other hand, enclosing scopes are often
# employed by lambda functions because they are expressions, they are
# almost always nested within a def.

# We are now ready to discuss about - the enclosing scope!

# We explored the way that nested functions can reference variables in an
# enclosing function’s scope, even if that function has already returned.
# It turns out that, we can also change such enclosing scope variables, as
# long as we declare them in 'nonlocal' statements. With this statement,
# nested defs can have both read and write access to names in enclosing
# functions.

# The 'nonlocal' statement is a close cousin to 'global', covered earlier.
# Like 'global', 'nonlocal' declares a name that will be changed in an
# enclosing scope. However, this won't change the variables which are
# defined outside all the defs in the global scope of the module. Thus,
# we can say that - 'nonlocal' statement has meaning only inside a function.

# Let's try to run through some examples:

def tester(start):
    state = start
    def nested(label):
        print(label, state)
    return nested

F = tester(0)
F('spam')       # Output: spam 0
F('ham')        # Output: ham 0
F('eggs')       # Output: eggs 0
```

```python
# Here we wish to get the 'state' variable in nested() auto-incremented.
# Changing a name in an enclosing def’s scope is not allowed by default.
# This means - we can't do the following:

def tester(start):
    state = start
    def nested(label):
        print(label, state)
        state += 1
    return nested

# Doing so will result into an error:
F = tester(0)
F('spam')
# UnboundLocalError: local variable 'state' referenced before assignment

# If we declare state in the tester scope as nonlocal within nested, we
# get to change it inside the nested function, too.

def tester(start):
    state = start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1          # This increments state on each call
    return nested

F = tester(0)
F('spam')       # Output: spam 0
F('ham')        # Output: ham 1
F('eggs')       # Output: eggs 2

# Some cautious cases to look out for:

# a) Unlike the global statement, nonlocal names really must have previously
# been assigned in an enclosing def’s scope when a nonlocal is evaluated,
# or else you’ll get an error—you cannot create them dynamically by
# assigning them anew in the enclosing scope:

def tester(start):
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested

# SyntaxError: no binding for nonlocal 'state' found

# b) 'nonlocal' restricts the scope lookup to just enclosing defs;
# nonlocals are not looked up in the enclosing module’s global scope or the
# built-in scope outside all defs, even if they are already there:
```

```
    state = 42
    def tester(start):
        def nested(label):
            nonlocal state
            print(label, state)
            state += 1          # This increments state on each call
        return nested

    # SyntaxError: no binding for nonlocal 'spam' found

# LOCAL SCOPE:

    # This is a scope inside a function. Once function execution completes, when
    # called, the variables in this scope aren't accessible. For example:

    def square(x)
        index = 2
        return x**index

    # Here 'x' and 'index' both are local to function square.

# --------------------------------------------------------------------------------

# 9. MORE FUNCTION TOOLS

# MAP

    # One of the more common things programs do with lists and other sequences
    # is apply an operation to each item and collect the results. For example:

    counters = [1, 2, 3, 4]
    updated = []
    for x in counters:
        updated.append(x+10)
    updated     # [11, 12, 13, 14]

    # But because this is such a common operation, Python actually provides a
    # built-in that does most of the work for you. The map function applies a
    # passed-in function to each item in an iterable object and returns a list
    # containing all the function call results. For example:
    def inc(x): return x + 10
    list(map(inc, counters))    # returns [11, 12, 13, 14]

    # Because map expects a function to be passed in, it also happens to be one
    # of the places where lambda commonly appears:
    list(map((lambda x: x+3), counters))    # returns [4, 5, 6, 7]

    # Moreover, map can be used in more advanced ways than shown here. For
    # instance, given multiple sequence arguments, it sends items taken from
    # sequences in parallel as distinct arguments to the function:
```

```
    list(map(pow, [1, 2, 3], [2, 3, 4]))    # # 1**2, 2**3, 3**4
    # returns [1, 8, 81]

    # The map call is similar to the list comprehension expressions. But map
    # applies a function call to each item instead of an arbitrary expression.
    # Because of this limitation, it is a somewhat less general tool. However,
    # in some cases map may be faster to run than a list comprehension (e.g.,
    # when mapping a built-in function), and it may also require less coding.

    # Python's built-in ord() function returns the ASCII integer code of a
    # single character (the chr() built-in is the converse – it returns the
    # character for an ASCII integer code).

    res = list(map(ord, 'spam'))
    res         # Output: [115, 112, 97, 109]

    # Here is our version of map, which is almost similar to internal
    # implementation. However, internal implemented map() is faster compared
    # to our for loop version. The purpose is to understand what map() does.

    def mymap(func, seq):
        res = []
        for x in seq:
            res.append(func(x))
        return res

# FILTER

    # filter() function is used to filter out items based on test function.
    # For example, following expression filters out positive numbers:

    list (filter((lambda x: x > 0), [-2, 3, -5, 5, 1, 4, -3]))
    # [3, 5, 1, 4]

    # Here is our for loop implementation of filter() function to understand
    # what it does internally.

    def myfilter(func, seq):
        res = []
        for x in seq:
            if func(x): res.append(func(x))
        return res

# REDUCE

    # reduce() in Python 3.x lives in functools module. It accepts an iterator
    # to process, but it's not an iterator itself – it returns a single result.
    # Here are two reduce calls that compute the sum and product of the items
    # in a list:
```

```
    from functools import reduce
    reduce((lambda x, y: x + y), [1, 2, 3, 4])      # returns 10
    reduce((lambda x, y: x*y), [1, 2, 3, 4])        # returns 24

    # The following function emulates most of the built-in's behavior and helps
    # demystify its operation in general:

    def myreduce(func, seq):
        tally = seq[0]
        for next in seq[1:]:
            tally = func(tally, seq)
        return tally

# -------------------------------------------------------------------------------
```

```
########################### TOPIC 6 - INPUT and OUTPUT #########################

# 1. input() FUNCTION

# INTRODUCTION:

    # This function first takes the input from the user and then evaluates
    # the expression, which means Python automatically identifies whether
    # user entered a string or a number or list. If the input provided is
    # not correct then either syntax error or exception is raised by python.

    val = input("Enter your value: ")
    print(val)

    # How input() works?
    # - When input() function executes program flow will be stopped until the
    #   user has given an input.
    # - The text or message display on the output screen to ask a user to enter
    #   input value is optional i.e. the prompt, will be printed on the screen
    #   is optional.
    # - Whatever you enter as input, input function convert it into a string.
    #   If you enter an integer value still input() function convert it into a
    #   string. You need to explicitly convert it into an integer in your code
    #   using typecasting.

    num = input ("Enter number :")
    print(num)
    name1 = input("Enter name : ")
    print(name1)

    print ("type of number", type(num))
    print ("type of name", type(name1))

    # Enter number :56
    # 56
    # Enter name : Sagar
    # Sagar
    # type of number <class 'str'>
    # type of name <class 'str'>

# TYPECASTING:

# a) Typecasting the input to Integer:

    num1 = int(input())
    num2 = int(input())
    print(num1 + num2)
```

```python
# b) Typecasting the input to Float:

    num1 = float(input())
    num2 = float(input())
    print(num1 + num2)

# c) Typecasting the input to String:

    # input
    string = str(input())
    print(string)

# MULTIPLE INPUTS:

    # In Python user can take multiple values or inputs in one line by
    # two methods.
    # - Using split() method
    # - Using List comprehension

# a) Using split() method

    # taking two inputs at a time
    x, y = input("Enter a two value: ").split()
    print("Number of boys: ", x)
    print("Number of girls: ", y)

    # taking three inputs at a time
    x, y, z = input("Enter a three value: ").split()
    print("Total number of students: ", x)
    print("Number of boys is : ", y)
    print("Number of girls is : ", z)

# b) Using List Comprehension

    # taking two input at a time
    x, y = [int(x) for x in input("Enter two value: ").split()]
    print("First Number is: ", x)
    print("Second Number is: ", y)

    # taking three input at a time
    x, y, z = [int(x) for x in input("Enter three value: ").split()]
    print("First Number is: ", x)
    print("Second Number is: ", y)
    print("Third Number is: ", z)

    # taking multiple inputs at a time
    x = [int(x) for x in input("Enter multiple value: ").split()]
    print("Number of list is: ", x)

# ------------------------------------------------------------------------------
```

```python
# 2. STRING FORMATTING:

    # There are several ways to format the output:
    # - Using formatted string literals
    # - Using str.format()
    # - Manually string formatting

# a) Using formatted string literals:

    # Formatted string literals (also called f-strings for short) let you
    # include the value of Python expressions inside a string by prefixing the
    # string with f or F and writing expressions as {expression}.

    # Example 1:
    year = 2016
    event = 'Referendum'
    print(f'Results of the {year} {event}')

    # Example 2:

    # Passing an integer after the ':' will cause that field to be a minimum
    # number of characters wide. This is useful for making columns line up.
    table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
    for name, phone in table.items():
        print(f'{name:10} ==> {phone:10d}')

    # Example 3:

    # The general syntax for a format placeholder is:
    #       %[width][.precision]type
    print(f"Total students: %3d\n Average Marks: %2.2f" %(240, 78.658333))

    # Example 4:

    # When the right argument is a dictionary, then the formats in the string
    # must include a parenthesised mapping key into that dictionary inserted
    # immediately after the '%' character.
    print(f'%(language)s has %(number)03d quote types.'%{'language': "Python", "num
ber": 2})

# b) Using str.format():

    # The brackets and characters within them (called format fields) are
    # replaced with the objects passed into the str.format() method.
    print('We are the {} who say "{}!"'.format('knights', 'Ni'))

    # A number in the brackets can be used to refer to the position of the
    # object passed into the str.format() method.
    print('{0} and {1}'.format('spam', 'eggs'))
    print('{1} and {0}'.format('spam', 'eggs'))
```

```python
    # If keyword arguments are used in the str.format() method, their values
    # are referred to by using the name of the argument.
    print('This {food} is {adjective}.'.format(food='spam', adjective='absolutely h
orrible'))

    # Positional and keyword arguments can be arbitrarily combined.
    print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred', other='Ge
org'))

    # If you have a really long format string that you don’t want to split up,
    # it would be nice if you could reference the variables to be formatted by
    # name instead of by position. This can be done by simply passing the dict
    # and using square brackets '[]' to access the keys.
    table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
    print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; Dcab: {0[Dcab]:d}'.format(tabl
e))

    # This could also be done by passing the table as keyword arguments with
    # the ‘**’ notation.
    table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
    print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))

    # Example 1:

    # Formatting of Integers
    String1 = "{0:b}".format(16)
    print("\nBinary representation of 16 is ")
    print(String1)

    # Formatting of Floats
    String1 = "{0:e}".format(165.6458)
    print("\nExponent representation of 165.6458 is ")
    print(String1)

    String1 = "{0:.2f}".format(1/6)
    print("\none-sixth is : ")
    print(String1)

    # Example 2:

    # A string can be left() or center(^) justified with the use of format
    # specifiers, separated by colon(:).

    String1 = "|{:<10}|{:^10}|{:>10}|".format('Left','Center','Right')
    print("\nLeft, center and right alignment with Formatting: ")
    print(String1)

    # Example 3:
    for x in range(1, 11):
        print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

```python
# c) Manual string formatting:

    # In this, formatting is done by using string functions. For example,
    # The str.rjust() method of string objects right-justifies a string
    # in a field of a given width by padding it with spaces on the left.
    # There are similar methods str.ljust() and str.center().

    for x in range(1, 11):
        print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
        print(repr(x*x*x).rjust(4))

    # There is another method, str.zfill(), which pads a numeric string
    # on the left with zeros.

    '12'.zfill(5)                # '00012'
    '-3.14'.zfill(7)             # '-003.14'
    '3.14159265359'.zfill(5)     # '3.14159265359'

# ------------------------------------------------------------------------------

# 3. TIPS FOR COMPETITIVE PROGRAMMERS (Advanced Topic)

# a) Normal Method:

    # The usual input() and print() functions are very slow.

    # Consider a question of finding the sum of N numbers given by the user.
    # INPUT FORM:
    # Input a number N.
    # Input N numbers separated by a single space in a line.

    # Basic Code will look like this
    n = int(input())
    arr = [int(x) for x in input().split()]
    summation = sum(arr)
    print(summation)

# b) Using stdin and stdout:

    # A bit faster method using inbuilt stdin and stdout.
    # - sys.stdin is a File object. This file will be standard input buffer.
    # - stdout.write('D\n') is faster than print('D').

    # import inbuilt standard input output
    from sys import stdin, stdout

    # input via readline method
    n = stdin.readline()
    # array input similar method
    arr = [int(x) for x in stdin.readline().split()]
    summation = sum(arr)
```

```python
    # could use inbuilt summation = sum(arr)

    # print answer via write().
    # Write() writes only strings. So we need to convert
    # any data other data into string first.
    stdout.write(str(summation))

# c) Adding a buffered pipe IO:

    # Simply, adding the buffered IO code before your submission code to
    # make the output faster.
    # - io.BytesIO object performs file input-output
    # in form of bytes. BytesIO objects have an internal pointer and for
    # every call to read(n) the pointer advances.
    # - The atexit module provides a simple interface to register functions
    # to be called when a program closes down normally.
    # - The sys module also provides a hook, sys.exitfunc, but only one
    # function can be registered there.

    # ----------------------- template begins ---------------------------
    # import libraries for input/ output handling on generic level
    import atexit, io, sys

    # A stream implementation using an in-memory bytes buffer. It inherits
    # BufferedIOBase.
    buffer = io.BytesIO()
    sys.stdout = buffer

    # print via here
    @atexit.register
    def write():
        sys.__stdout__.write(buffer.getvalue())
    # ----------------------- template ends -----------------------------

    n = int(input())
    arr = [int(x) for x in input().split()]
    summation = sum(arr)
    print(summation)

-----------------------------------------------------------------------------------
```

```
############################# TOPIC 7 - MODULES ################################

# 1. INTRODUCTION TO MODULES

# DIFFERENCE BETWEEN MODULES AND SCRIPTS:

    # Suppose we create a file - myfile.py; we can use this file in two ways:
    # a) as a module - that can be imported in python scripts or REPL
    # b) python script - which runs on OS's shell to do some task
    # Suppose the structure of myfile.py is -

    from math import factorial

    def combination(n, k):
        if n > k:
            return int(factorial(n)/(factorial(k)*factorial(n-k)))
        else:
            print("n can't be smaller than k.\n")

    def permutation(n, k):
        if n > k:
            return int(factorial(n)/factorial(n-k))
        else:
            print("n can't be smaller than k.\n")

    # a) On REPL, we can use this file as module by making an import statement.

    import myfile               # Notice how we omit .py extension during import
    # We can now execute our functions using -
    myfile.combination(5, 3)
    # We can import specific parts from the module as -
    from myfile import permutation
    permutation(8, 5)

    # b) But on OS's shell i.e. cmd or terminal, when we run this file -
    #    this script does nothing.
    python myfile.py

    # This is usual because we know - all we have done is defined the function
    # but never called them. So let's call them in script!
    # Add following two lines at the end of myfile.py
    print(combination(5, 3), end=" ")
    print(permutation(8, 5))

    # Run this script from OS's shell, as:
    python myfile.py
    # Output: 10 6720

    # But the issue is - when we import this file as module on REPL,
    # we get an output '10 6720', even if we have not called any function
    # on REPL. On REPL,
```

```python
    import myfile
    # Output: 10 6720

    # If we import the file again, nothing happens.
    import myfile           # No output like - '10 6720'

    # It seems, as if print() statements were not executed. Thus, we can say
    # that - in python, a file is imported only once.

    # Our aim is now to make a module from which we can usefully import
    # functions without running the function calls, as well as a file which
    # can be run as the script.

# '__name__' AND '__main__':

    # Python runtime environment defines special attribute names, which are
    # delimited by double underscores. One such attribute is: __name__

    # This __name__ evaluates to __main__ or the actual module name depending
    # on how the enclosing module is being used. To see how, add the following
    # statement as last line of your module:
    print(__name__)

    # On REPL, when we do - import myfile, we get additional line as output,
    # which is - 'myfile'. If we run this module as script - python myfile.py,
    # we get additional line as output - which is '__main__'.

    # Hence, we modify our module as:

    from math import factorial

    def combination(n, k):
        if n > k:
            return int(factorial(n)/(factorial(k)*factorial(n-k)))
        else:
            print("n can't be smaller than k.\n")

    def permutation(n, k):
        if n > k:
            return int(factorial(n)/factorial(n-k))
        else:
            print("n can't be smaller than k.\n")

    if __name__ == "__main__":
        print(combination(5, 3), end=" ")
        print(permutation(8, 5))

    # Now we can safely import our module without unduly executing our
    # function calls and execute the calls only when run as a script.

# ------------------------------------------------------------------------------
```

```
######################## TOPIC 8 - CLASSES and OBJECTS #########################

# 1. INTRODUCTION TO CLASSES

    # Classes provide a means of bundling data and functionality together.
    # Creating a new class creates a new type of object, allowing new instances
    # of that type to be made. Each class instance can have attributes attached
    # to it for maintaining its state. Class instances can also have methods
    # (defined by its class) for modifying its state.

    # Let's try to make a model for aircraft flights and learn the concepts
    # parallely. Create a file - airtravel.py

    class Flight:
        pass

    # This is a minimal class, and does nothing. Now, go to REPL, and do this:

    from airtravel import Flight
    Flight          # Output: <class 'Flight'>
    f = Flight()
    type(f)         # Output: <class 'Flight'>

    # Here 'f' is the object of class Flight. For every class, python provides
    # it's own constructor (i.e. We don't define constructors in class
    # definition) to create objects. Now, let's add some few more statements in
    # our class.

    class Flight:
        def number(self):
            return 'SN060'

    # The first argument of all the methods (i.e. functions of class) is self.
    # This is because:

    # We call this function using object as:
    f = Flight()
    f.number()          # Output: SN060         (Form 1)

    # However, Python interprets above function call as follows:

    Flight.number(f)    # Output: SN060         (Form 2)

    # The above form is also correct and is equivalent to Form 1. However,
    # we will use Form 1 only as hardly anybody uses Form 2.

# --------------------------------------------------------------------------------
```

```
# 2. __init__ method:

    # The above class is not useful as it only represents one particular flight.
    # We need to make flight's number method more general. To do that we need to
    # learn the initialization method i.e. __init__().

    # If provided an initialization method, it is called uring during the process
    # of creating an object. Like all other instance methods, first argument of
    # initializer method must be self. Initializer method should not return
    # anything. It should just modify the object using 'self'.

    # Note: __init__() is an initializer, not a constructor. Constructor is
    #       provided by the Python runtime system and the constructor is called
    #       during creation of objects. When that constructor is called, one
    #       thing constructor function does is calling the initializer function.
    #       And 'self' is similar to 'this' in C++ or Java.

    # During initalization we create identifiers used inside the class by
    # assignment operation. We follow the naming convention that the identifiers
    # used inside the class start with an underscore.

    class Flight:
        def __init__(self, number):
            self._number = number

        def number(self):
            return self._number

    # On REPL, execute the following:

    from airtravel import flight
    f = Flight('SN060')
    f.number()      # Output: SN060

    # Note: There are no 'public', 'private' or 'protected' keywords in Python
    #       as everything is public. There is a way of creating private
    #       variables in Python - with a way called 'Name Mangling', but we will
    #       see it later. But there is no strict way to define private or
    #       protected variables. This does not affect the security as program's
    #       security is maintained by system methods and is enough to provide
    #       sufficient security, even in huge programs.

    # To check if a flight is valid or not, we need to do some checking and
    # should raise errors if in case of exceptions. (We will talk about exceptions
    # in a lot more detail, in the next topic.)
```

```
    class Flight:
        def __init__(self, number):
            if not number[:2].isalpha():
                raise ValueError("No airline code in {}".format(number))
            if not number[:2].isupper():
                raise ValueError("Invalid airline code {}".format(number))
            if not (number[2:].isdigit() and int(number[2:]<=9999)):
                raise ValueError("Invalid route number {}".format(number))
            self._number = number

        def number(self):
            return self._number

        def airline(self):
            return self._number[:2]

# -------------------------------------------------------------------------------
```

```
# 3. WORKING WITH MULTIPLE CLASSES

    # One of the features which we will wish from our aircraft model is -
    # that it should accept seat bookings. For that we should first build the
    # seat layour and for that we need to know the aircraft. So lets make
    # another class to model different kinds of aircrafts:

    class Aircraft :
        def __init__(self, registration, model, num_rows, num_seats_per_row):
            self._registration = registration
            self._model = model
            self._num_rows = num_rows
            self._num_seats_per_row = num_seats_per_row

        def registration(self):
            return self._registration

        def model(self):
            return self._model

    # Rows in aircraft are numbered from 1 and seats in each row are designated
    # with letter which omits 'I' to avoid confusion with 1.

        def seating_plan(self):
            return (range(1, self._num_rows + 1),
"ABCDEFGHJK"[:self._num_seats_per_row])

    # range() from the iterable sequence of row numbers from 1 upto row numbers
    # in the plane. String is sliced here to return one character per row. These
    # two objects are bundled up into a tuple. With that in mind - lets
    # construct a plane: (On REPL)
```

```python
from airtravel import *
a = Aircraft('GEUPT', 'Airbus A319', num_rows = 22, num_seats_per_row = 6)

# Now, a user should get details of seating arrangement from flight class
# only. It is not good implementation to expose all classes to user.
# Instead we pass objects of other classes to a single class and let user
# interact with the object of that class.

# So we pass objects of aircraft to Flight. So now our airtravel.py looks
# like this:

class Flight:
    def __init__(self, number, aircraft):
        if not number[:2].isalpha():
            raise ValueError("No airline code in {}".format(number))
        if not number[:2].isupper():
            raise ValueError("Invalid airline code {}".format(number))
        if not (number[2:].isdigit() and int(number[2:]<=9999)):
            raise ValueError("Invalid route number {}".format(number))
        self._number = number
        self._aircraft = aircraft

    def number(self):
        return self._number

    def airline(self):
        return self._number[:2]

    def aircraft_model(self)
        return self._aircraft.model()

class Aircraft :
    def __init__(self, registration, model, num_rows, num_seats_per_row):
        self._registration = registration
        self._model = model
        self._num_rows = num_rows
        self._num_seats_per_row = num_seats_per_row

    def registration(self):
        return self._registration

    def model(self):
        return self._model

    def seating_plan(self):
        return (range(1, self._num_rows + 1),
"ABCDEFGHJK"[:self._num_seats_per_row])

# Now we can proceed to implement our simple booking system.
```

```
# This is how we plan our seating arrangement:

# There will be a list of dictionaries, where each dictionary will map
# seat alphabet to passenger name in string format. We initialize the
# seating plan in the 'flight' using the following fragment:

rows, seat = self._aircraft.seating_plan()
self._seating = [None] + [{letter:None for letter in seats} for _ in rows]

# The second line creates a list of dictionaries for seat allocation.
# Rather than continuously dealing with the fact that the seating index
# is 1 based and python lists are 0 based, we choose to waste 1 entry by
# setting 1st entry to 'None'. The list is constructed using list
# comprehension which iterates over the rows object which is the range of
# row numbers retrived from rows of the aircraft in the previous line.
# We are not interested in row numbers, hence we discard it by using the
# dummy underscore variable.
# The main part of the list comprehension is the dictionary comprehension.
# This iterates for each letter of the row and includes the mapping from
# the single character string to None to indicate an empty seat.

# We use list comprehension rather than * operator for creating rows
# since we need distinct rows and remember - repetition is shallow.

# Let's test our seating arrangement now on REPL:

from airtravel import *
f = flight('BA758', Aircraft('GEUPT', 'Airbus A319', num_rows=22,
num_seats_per_row = 6))
f._seating      # This gives accurate results but not particularly beautiful

# We use pretty print for better print format.

from pprint import pprint as pp
pp(f._seating)      # This gives a very good format

# Now let's add feature to allocate seats to passengers. Let's keep this
# simple by assuming passenger is simply a string name. Most of the code
# is here about - seat validation.


# This is in flight class:
```

```python
def allocate_seat (self, seat, passenger):
    rows, seat_letters = self._aircraft.seating_plan()

    letter = seat[-1]
    if letter not in seat_letters:
        raise ValueError('Invalid seat letter {}'.format(letter))

    row_text = seat[:1]
    try:
        row = int(row_text)
    except ValueError:
        raise ValueError('Invalid seat row {}'.format(row_text))

    if row not in rows:
        raise ValueError('Invalid row number {}'.format(row))

    if seat._seating[row][letter] is not None:
        raise ValueError('Seaat {} is already occupied.'.format(seat))

    self._seating[row][letter] = passenger

# We now refactor a code a little, because we are doing seat validation
# in allocate_seat() method. We define another function _parse_seat() which
# will contain parsing and validatin logic in its own.

def _parse_seat(self, seat):
    row_numbers, seat_letters = self._aircraft.seating_plan()

    letter = seat[-1]
    if letter not in seat_letters:
        raise ValueError('Invalid seat letter {}'.format(letter))

    row_text = seat[:1]
    try:
        row = int(row_text)
    except ValueError:
        raise ValueError('Invalid seat row {}'.format(row_text))

    if row not in rows:
        raise ValueError('Invalid row number {}'.format(row))

    return row, letter

 def allocate_seat(self, seat, passenger):
    row, letter = self._parse_seat(seat)
    if seat._seating[row][letter] is not None:
        raise ValueError('Seaat {} is already occupied.'.format(seat))

    self._seating[row][letter] = passenger

# We now add a function to relocate the passenger's seat.
```

```
    def relocate_passenger (self, from_seat, to_seat):
        from_row, from_letter = self._parse_seat(from_seat)
        if self._seating[from_row][from_letter] is None:
            raise ValueError("No passenger on seat {}".format(from_seat))

        to_row, to_letter = self._parse_seat(to_seat)
        if self._seating[to_row][to_letter] is not None:
            raise ValueError("Seat {} is not vacant.".format(to_seat))

        self._seating[to_row][to_letter] = self._seating[from_row][from_letter]
        self.seating[from_row][from_letter] = None

    # It's important during booking to know, how many seats are available.

    def num_available_seats(self):
        return sum(sum(1 for s in row.values() if s is None) for row in
self._seating if row is not None)

    # Also we have to print 'Boarding cards' for the passengers. You can see -
    # there is no 'self' in the argument list of following function. Because
    # this function doesn't belong to any class. It is module level function.

    def console_card_printer(passenger, seat, flight_number, aircraft):
        output = "| Name: {0} "\
                 " Flight: {1}"\
                 " Seat: {2}"\
                 " Aircraft: {3}"\
                 " |".format(p, flight_number, seat, aircraft)
        banner = '+' + '-'*(len(output)-2) + '+'
        border = '|' + '-'*(len(output)-2) + '|'
        lines = [banner, border, output, border, banner]
        card = '\n'.join(lines)
        print(card)
        print()

    # The following functions go into flight class:

    def make_boarding_cards (self, card_printer):
        for passenger, seat in sorted(self._passenger_seats()):
            console_card_printer(passenger, seat, self.number(),
self.aircraft_model())

    def _passenger_seats(self):
        row_numbers, seat_letters = self._aircraft.seating_plan()
        for row in row_numbers:
            for letter in seat_letters:
                passenger = self._seating[row][letter]
                if passenger is not None:
                    yield(passenger, "{}{}".format(row, letter))

# ------------------------------------------------------------------------------
```

```
# 4. FINAL CODE

    # This is how - at the end 'airtravel.py' file should look like:

    """Model for Aircraft flights."""

    class Flight:
        """A flight with a particular passenger aircraft."""

        def __init__(self, number, aircraft):
            if not number[:2].isalpha():
                raise ValueError("No airline code in '{}'".format(number))

            if not number[:2].isupper():
                raise ValueError("Invalid airline code '{}'".format(number))

            if not (number[2:].isdigit() and int(number[2:]) <= 9999):
                raise ValueError("Invalid route number '{}'".format(number))

            self._number = number
            self._aircraft = aircraft

            rows, seats = self._aircraft.seating_plan()
            self._seating = [None] + [ {letter:None for letter in seats} for _ in
rows ]


        def number(self):
            return self._number


        def airline(self):
            return self._number[:2]


        def aircraft_model(self):
            return self._aircraft.model()


        def _parse_seat(self, seat):
            """Parse a seat designator into a valid row and letter.

            Args:
                seat: A seat designator such as 12F

            Returns:
                A tuple containing an integer and a string for row and seat.
            """
            row_numbers, seat_letters = self._aircraft.seating_plan()

            letter = seat[-1]
```

```
        if letter not in seat_letters:
            raise ValueError("Invalid seat letter {}".format(letter))

        row_text = seat[:-1]
        try:
            row = int(row_text)
        except ValueError:
            raise ValueError("Invalid seat row {}".format(row_text))

        if row not in row_numbers:
            raise ValueError("Invalid row number {}".format(row))

        return row, letter


    def allocate_seat(self, seat, passenger):
        """Allocate a seat to a passenger.

        Args:
            seat: A seat designator such as '12C' or '21F'.
            passenger: The passenger name.

        Raises:
            ValueError: If the seat is unavailable.
        """
        row, letter = self._parse_seat(seat)

        if self._seating[row][letter] is not None:
            raise ValueError("Seat {} already occupied".format(seat))

        self._seating[row][letter] = passenger


    def relocate_passenger(self, from_seat, to_seat):
        """Relocate a passenger to a different seat.

        Args:
            from_seat: The existing seat designator for the
                    passenger to be moved.

            to_seat: The new seat designator.
        """

        from_row, from_letter = self._parse_seat(from_seat)
        if self._seating[from_row][from_letter] is None:
            raise ValueError("No passenger to relocate in seat {}"
.format(from_seat))

        to_row, to_letter = self._parse_seat(to_seat)
        if self._seating[to_row][to_letter] is not None:
            raise ValueError("Seat {} already occupied".format(to_seat))
```

```
            self._seating[to_row][to_letter] = self._seating[from_row][from_letter]
            self._seating[from_row][from_letter] = None


        def num_available_seats(self):
            return sum( sum(1 for s in row.values() if s is None)
                        for row in self._seating
                        if row is not None)


        def make_boarding_cards(self, card_printer):
            for passenger, seat in sorted(self._passenger_seats()):
                card_printer(passenger, seat, self.number(), self.aircraft_model())

        def _passenger_seats(self):
            """An iterable series of passenger seating allocations."""
            row_numbers, seat_letters = self._aircraft.seating_plan()
            for row in row_numbers:
                for letter in seat_letters:
                    passenger = self._seating[row][letter]
                    if passenger is not None:
                        yield (passenger, "{}{}".format(row, letter))

    class Aircraft:
        def __init__(self, registration, model, num_rows, num_seats_per_row):
            self._registration = registration
            self._model = model
            self._num_rows = num_rows
            self._num_seats_per_row = num_seats_per_row


        def registration(self):
            return self._registration


        def model(self):
            return self._model


        def seating_plan(self):
            return (range(1, self._num_rows + 1),
"ABCDEFGHJK"[:self._num_seats_per_row])


    def make_flights():
        f = flight('BA758', Aircraft('GEUPT', 'Airbus A319', num_rows=22,
num_seats_per_row = 6))
        f.allocate_seat('12A', 'Guido van Rossum')
        f.allocate_seat('15F', 'Bjarne Stroustrup')
        f.allocate_seat('15E', 'Anders Hejlsberg')
        f.allocate_seat('1C', 'John McCarthy')
```

```
        f.allocate_seat('1D', 'Richard Hickey')
    return f, g


def console_card_printer(passenger, seat, flight_number, aircraft):
    output = "| Name: {0}"     \
             "  Flight: {1}"   \
             "  Seat: {2}"     \
             "  Aircraft: {3}" \
             " |".format(passenger, flight_number, seat, aircraft)
    banner = '+' + '-' * (len(output) - 2) + '+'
    border = '|' + ' ' * (len(output) - 2) + '|'
    lines = [banner, border, output, border, banner]
    card = '\n'.join(lines)
    print(card)
    print()
# ----------------------------------------------------------------------------
```

```python
######################### TOPIC 9 - EXCEPTION HANDLING ########################

# 1. SYNTAX ERROR

    # There are (at least) two distinguishable kinds of errors:
    # syntax errors and exceptions.

    # Syntax errors, also known as parsing errors, are perhaps the most common
    # kind of complaint you get while you are still learning Python:
    while True print('Hello world')
    # File "<stdin>", line 1
    #     while True print('Hello world')
    #                  ^
    # SyntaxError: invalid syntax

    # The parser repeats the offending line and displays a little 'arrow'
    # pointing at the earliest point in the line where the error was detected.
    # In the example, the error is detected at the function print(), since a
    # colon (':') is missing before it. File name and line number are printed
    # so you know where to look in case the input came from a script.

# ------------------------------------------------------------------------------

# 2. EXCEPTIONS

# try...except block:

    # Even if a statement or expression is syntactically correct, it may cause
    # an error when an attempt is made to execute it. Errors detected during
    # execution are called exceptions and are not unconditionally fatal:
    # you will soon learn how to handle them in Python programs.

    10 * (1/0)
    # Traceback (most recent call last):
    # File "<stdin>", line 1, in <module>
    # ZeroDivisionError: division by zero

    4 + spam*3
    # Traceback (most recent call last):
    # File "<stdin>", line 1, in <module>
    # NameError: name 'spam' is not defined

    '2' + 2
    # Traceback (most recent call last):
    # File "<stdin>", line 1, in <module>
    # TypeError: Can't convert 'int' object to str implicitly

    # It is possible to write programs that handle selected exceptions.
    # Look at the following example, which asks the user for input until a
    # valid integer has been entered, but allows the user to interrupt the
    # program (using Control-C or whatever the operating system supports);
```

```python
# note that a user-generated interruption is signalled by raising the
# KeyboardInterrupt exception.

while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")

# The try statement works as follows:

# - First, the try clause (the statement(s) between the try and except
#   keywords) is executed.

# - If no exception occurs, the except clause is skipped and execution of
#   the try statement is finished.

# - If an exception occurs during execution of the try clause, the rest of
#   the clause is skipped. Then if its type matches the exception named
#   after the except keyword, the except clause is executed, and then
#   execution continues after the try statement.

# - If an exception occurs which does not match the exception named in the
#   except clause, it is passed on to outer try statements; if no handler is
#   found, it is an unhandled exception and execution stops with a message
#   as shown above.

# - A try statement may have more than one except clause, to specify
#   handlers for different exceptions. At most one handler will be executed.

# - An except clause may name multiple exceptions as a parenthesized tuple,
#   for example:
except (RuntimeError, TypeError, NameError):

# - The raise statement allows the programmer to force a specified
#   exception to occur. For example:
raise NameError('HiThere')
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# NameError: HiThere

# - The last except clause may omit the exception name(s), to serve as a
#   wildcard. Use this with extreme caution, since it is easy to mask a
#   real programming error in this way!
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
```

```python
    except OSError as err:
        print("OS error: {0}".format(err))
    except ValueError:
        print("Could not convert data to an integer.")
    except:
        print("Unexpected error:", sys.exc_info()[0])
        raise
    # Without any argument 'raise' raises the exception, currently being
    # handled.

# A Sqrt() Example:

    # Following function computes square root using 'Heron's Method'.

    def sqrt(x):
        guess = x
        i = 0
        while guess*guess != x and i < 20:
            guess = (guess + x/guess)/2.0
            i += 1
        return guess

    def main():
        print(sqrt(9))
        print(sqrt(2))
        print(sqrt(-1))

    if __name__ == "__main__":
        main()

    # Now when we execute the above script, the error which we get is:

    # ZeroDivisionError: float division by zero

    # Now we know that - User of sqrt() function won't generally expect
    # ZeroDivisionError. Thus, we use exception handing to print errors
    # that users will anticipate. Let's modify the main() as:

    def main():
        print(sqrt(9))
        print(sqrt(2))
        try:
            print(sqrt(-1))
        except ZeroDivisionError:
            print("Cannot compute square root of negative number.")

    # Instead of modifying main() method, we should have incorporated exception
    # handling in sqrt() function itself. Also if we are sure that the sqrt()
    # method will fail for -ve numbers instead of trying sqrt(-ve number), we
    # can simply put an if statement and raise the error.
```

```
    def sqrt(x):
        if x < 0:
            raise ValueError("Cannot compute square root of negative number"
                             "{}.".format(x))
        guess = x
        i = 0:
        while guess*guess != x and i < 20:
            guess = (guess + x/guess)/2.0
            i += 1
        return guess

    def main():
        print(sqrt(9))
        print(sqrt(2))
        try:
            print(sqrt(-1))
        except ZeroDivisionError:
            print("Cannot compute square root of negative number.")

    # If we run the code now, we still get the unexpected program 'Traceback'
    # call, because we forgot to modify our exception handler to catch
    # 'ValueError' rather than 'ZeroDivisionError'.

    import sys

    def sqrt(x):
        if x < 0:
            raise ValueError("Cannot compute square root of negative number"
                             "{}.".format(x))
        guess = x
        i = 0:
        while guess*guess != x and i < 20:
            guess = (guess + x/guess)/2.0
            i += 1
        return guess

    def main():
        try:
            print(sqrt(9))
            print(sqrt(2))
            print((sqrt(-1)))
            print("This statement won't get executed because the control has "
                  "shifted to except block in the previous statement.\n")
        except ValueError as e:
            print(e, file=sys.stderr)

# Error Handling Philosophies:

    # There are 2 philosophies of handling failures -
    # a) Look Before You Leap (LBYL)
    # b) Easier to Ask Forgiveness than Permission (EAFP)
```

```python
# a) LBYL Approach:

# Consider the following code for file processing. We are not interested in
# what is the process() being happening. We can assume, the process() func
# does required processing for us.

import os
p = '/path/to/datafile.dat'
if os.path.exists(p):
    # 'Location X' - read following explanation
    process_file(p)
else:
    print('No such file as {}'.format(p))

# There are several problems with this approach:

# 1) This only checks condition for file existence.
#    What if the file exists, but contains garbage?
#    What if the path refers to a directory and not to a file?
# According to LBYL approach, we should add further tests for handling
# these failures too.

# 2) A more subtle problem is at Location X. It is possible for file to get
#    deleted by some other process between the existence check and
#    process_file(p) call. This is classic 'atomicity' issue. There is no
#    good way here to deal with this.

# If we started solving these errors, the main logic of code is set behind
# and we have got involved more in writing conditions.

# b) EAFP Approach:

# This method says - Handle the exceptions not by using conditions, but by
# exception handling. Python is in favour of this appraoch. This is because
# Python believes that if a any error can occur, let it occur. This will
# safeguard the program from improper usage.

# "Errors are like bells, and if we make them silent, they are of no use."

p = '/path/to/datafile.dat'
try:
    process_file(p)
except OSError as e:
    print('Could not process file because {}'.format(str(e))
```

```python
# finally block:

    # finally statement always executes whether the exception has occured or not.
    # This statement is majorly used for resource clean up. For example:

    # Consider the following function which uses various facilities provided by
    # 'os' module and - changes the current working directory, make a directory
    # at new location and come back to current working directory.

    import os

    def make_at(path, dir_name):
        original_path = os.getcwd()
        os.chdir(path)
        os.mkdir(dir_name)
        os.chdir(original_path)

    # At first sight, the function seems reasonable. But if os.mkdir() function
    # fails for some reason and is unable to make a directory at mentioned path,
    # the current working directory won't be restored to its original value.
    # And make_at() function may lead to unintended side effects. We fix this
    # by using 'finally' block.

    import os
    def make_at(path, dir_name):
        original_path = os.getcwd()
        try:
            os.chdir(path)
            os.mkdir(dir_name)
        finally:
            os.chdir(original_path)

    # The above function could be made even better by handling OSError as -

    import os, sys
    def make_at(path, dir_name):
        original_path = os.getcwd()
        try:
            os.chdir(path)
            os.mkdir(dir_name)
        except OSError as e:
            print(e, file = sys.stderr)
        finally:
            os.chdir(original_path)

# ------------------------------------------------------------------------------
```

```
############################## TOPIC 10 - FILE I/O ##############################

# 1. BASIC FUNCTIONS

    # To open a file in python, we use inbuilt function - open(). This function
    # takes following arguments:
    # - path: path of the file
    # - mode: read/write/append, binary/text
    # - encoding: text encoding (by default it's 'utf-8' in most of the systems)

    f = open('filepath/filename.txt', mode='wt', encoding='utf-8')
    # In this case, 'w' means write and 't' means text.

    # Here is the list of some open() modes:
    # 'r' - open for reading file (default)
    # 'w' - open for writing in file by overwriting the already existing data
    # 'a' - open for wrtiting in file by appending the data to the end of file
    # 't' - text mode (default)
    # 'b' - binary mode
    # '+' - open a disk file for updating (reading and writing at same time)

    # Let's write some text in the opened file. This is done by using write()
    # function on the file object. write() returns the number of characters
    # written in the file.

    f.write('What are the roots that clutch, ')     # 32
    f.write('what branches grow\n')                 # 19
    f.write('Out of this stony rubbish? ')          # 27
    f.close()                                       # Closing file is important

    # Now let's read what we have written in the file.

    g = open('filepath/filename.txt', mode='rt', encoding='utf-8')

    # In text mode, read method accepts the number of characters to be read.
    g.read(32)  # Output: 'What are the roots that clutch, '

    # If no argument is given to read(), it will read out the remaining
    # characters till the end of the file.
    g.read()    # Output: 'what branches grow\nOut of this stony rubbish? '

    # Further calls on read would return an empty string.
    g.read()    # Output: ''

    # Usually we close the file after reading it, but here we will read this
    # file again. To do so, we must set the iterator to the beginning of the
    # file. For this, we use seek() method. seek(x) sets the iterator
    # to the x'th character.
    g.seek(0)
```

```
    # Usually, read() is not a good method to read the file because it reads
    # the file all at once. There is another method, which reads the file
    # line by line.

    g.readline()
    # Output: 'What are the roots that clutch, what branches grow\n'
    g.readline()
    # Output: 'Out of this stony rubbish? '

    # When we know that the file is not big enough, and we can store every
    # line in a list, we use readlines() method.
    g.seek(0)
    g.readlines()
    # Output: ['What are the roots that clutch, what branches grow\n', 'Out of
    #            this stony rubbish? ']
    g.close()

    # Now let's append few lines in our existing file.
    h = open('filepath/filename.txt', mode='at', encoding='utf-8')
    h.writelines(
        ['Son of man, \n',
        'You cannot say, or guess, ',
        'for you know only, \n',
        'A heap of broken images, ',
        'where the sun beats\n']
    )
    h.close()

# ---------------------------------------------------------------------------

# 2. FILES AS ITERATORS

    # Let's write a function in a python script, which does the job of printing
    # each line of the file. Since file objects are iterators as well, we can
    # use them in for loops like any other iterator. Following is the script
    # named as - files.py

    import sys

    def main(filename):
        f = open(filename, mode='rt', encoding='utf-8')
        for line in f:
            print(line)
        f.close()

    if __name__ == '__main__':
        main(sys.argv[1])

    # Now we run this script by using following command:
    python3 files.py filename.txt
```

```python
    # Output:
    # What are the roots that clutch, what branches grow
    #
    # Out of this stony rubbish? Son of man,
    #
    # You cannot say, or guess, for you know only,
    #
    # A heap of broken images, where the sun beats
    #

    # The lines are separated by newlines because each line of the file
    # terminates with a newline and print() function prints a newline at the
    # end of the line by it's own. This issue could be solved by two ways:
    # Either use strip() method to remove the newline character at the end
    # of the line and let the print add its own newline, or instead of print()
    # use sys.stdout.write() method.

    def main(filename):
    f = open(filename, mode='rt', encoding='utf-8')
    for line in f:
        sys.stdout.write(line)
    f.close()

# ------------------------------------------------------------------------------

# 3. WITH statement

    # Till now, we followed this pattern - Open a file, work with the file and
    # close the file. close() is important because if you dont close the data,
    # it is possible to lose the data and makes system less secure and reliable.
    # Due to some reasons - it is possible that close() function doesn't get
    # executed. For example, if you are opening too many files, your system may
    # run out of resources and close() for all the files is not executed.
    # Since we always want to pair - every open() with a close(), we can either
    # use try...finally block structure or use 'with' blocks.

    # 1) without using with statement
    file = open('file_path', 'w')
    try:
        file.write('Hello world!')
    finally:
        file.close()

    # 2) using with statement
    with open('file_path', 'w') as file:
        file.write('Hello world!')

    # Notice that unlike the first implementation, there is no need to call
    # file.close() when using with statement. The with statement itself ensures
    # proper acquisition and release of resources.
# ------------------------------------------------------------------------------
```