PES UNIVERSITY

(*Established under Karnataka Act No. 16 of 2023*)
100 ft. Ring Road, Bengaluru – 560085, Karnataka, India

Programming with Python

Unit I – Python Basics and Data Structures

Prepared by,

Niteesh K R,
Assistant Professor,
Department of Computer Applications,
PES University,
Bengaluru

Major features of Python

- Created by Guido van Rossum
- High-level, general-purpose language
- Dynamically-typed
  - No variable is **bound** to a datatype
  - Datatype is decided based on the value held in the variable
- Free and open source
  - For commercial and personal use
  - Anyone can contribute
  - Latest Version: 3.12
- Portable and cross-platform support
  - Same program works on different platforms (UNIX, Windows, Linux, macOS etc.)
  - Some modifications (Like file path) maybe necessary
- Extensible and Embeddable with C/C++
- Interpreted, not compiled
  - One line is translated at a time
- Large standard libraries
  - Enables easier programming
  - Reuse existing library functions and classes
  - Avoid rewriting and reinventing the wheel
  - New modules can also be added to the library (Python is open source)
- Object-oriented Language
  - Everything is treated as an object
    - Eg., Integers belong to **class int**
  - Easier to represent and model real-world problems

Working with Python

- **Immediate Mode**
  - Each line is interpreted. Eg., IDLE, Command Line Python
- **Script Mode**
  - Creating a file with .py as the extension and executing the same
- **Integrated Development Environment (IDE)**
  - Various IDEs support Python

Basics of Writing Python Programs

- Each line is a **Statement**
- No Semicolons to separate lines. Every new line is treated as a new statement
- Sequential Execution
- Case-sensitive Language
- Comments
  - Single line – # Comment
  - Multiline – ' ' ' Comment ' ' ' or " " " Comment " " "
  - Comments are ignored while interpreting
  - In most IDEs, a line can be commented using Cmd + / or Ctrl + /
  - Good practice to use comments to explain what the code does
- Set of statements is called **Suite** or a **Compound Statement**
  - Each suite begins with a **':'** and every statement under the suite must be indented the same level
  - No curly braces to begin and end a compound statement
- Literals

  - String literals
  - Numeric literals
  - Boolean literals
  - Special literal (None)

- Strings
  - Strings are written within a pair of single or double quotes
  - Strings belong to **class str**
  - No concept of char in Python; Even a single character is a string
  - Indexed (Each character can be identified using index position) but, immutable. Index value starts from 0.

- Numeric Literals
  - Integers
    * Belong to **class int**
    * Integer literals are immutable
    * No maximum restriction on the memory allocated
  - Floating-point numbers
    * Belong to **class float**
    * Float literals are immutable
    * No maximum restriction on the memory allocated
    * 15 places after decimal point
  - Complex numbers
    * Belong to **class complex**
    * Represents numbers of form

$$a + jb \; (Where, j = \sqrt{-1}) \tag{1}$$

  - Type conversion
    * Lower to higher – Possible
    * Higher to lower – Not possible
    * int can be converted to float or complex

        - float can be converted to complex, not int
        - complex can not be converted to int or float

- Boolean Literals
    - True
    - False
- Special Literal (None)
    - **None** is a special literal
    - None is a built-in constant
    - Belongs to **class NoneType**
    - A variable with no value
- Keywords
    - Reserved words with a predefined meaning
    - Can not be used as variable names or function names
    - Python v3.12 has **33 keywords**
- Data Collections
    - Lists
    - Tuples
    - Sets
    - Dictionaries
    - Strings

Operators

- **Arithmetic Operators**

  +, -, *, /, %

- **Assignment Operators**

  =, All Shorthand Operators

- **Logical Operators**

  and, or, not

- **Relational or Comparison Operators**

  Compares two values (Less than, less than or equal, greater than, greater than or equal, equal, not equal)

- **Membership Operators**

  in and not in

  - Used to check if an item is present in an iterable

- **Identity Operators**

  is and is not

  - Used to check if two variables are having same value and are pointing to the same address in the memory
  - Strict equality: Checks for both value and type

- **Bitwise Operators**

  Bitwise and, Bitwise or, Bitwise not, Bitwise XOR, Bitwise left shift, Bitwise right shift

Control Structures

- Simple if
- if-else
- if-elif-else (if ladder)
- Nested if
- switch

Loops

- while loop
- while-else loop
- for-in loop (Often used with iterables)
- for-in-else loop

Note:
break: Breaks the loop it is present in. If break is present in an inner loop, it will break the inner loop

continue: Skips the remainder of the current loop and continues from the next iteration

Python Datatypes and Collections

- Mutable / Modifiable types
  - List (Collection)
  - Dictionary (Collection)
  - Set (Collection)
- Immutable / Non-modifiable types
  - Numeric (int, float, complex)
  - String (Collection)
  - Tuple (Collection)

Special Note: All Collections have multiple elements as items in them. All Collections are iterable, meaning, they can be traversed using a for loop. All iterables are interconvertible, meaning, can be converted from one type to another.

Lists

- Mutable
- Indexed
- Ordered
- Heterogeneous
- Created using list() or []
- One of the most flexible data collections in Python
- Closest possible collection to an array
- Eg., L = [1, 2, 'abc']

Tuples

- Immutable
- Indexed
- Ordered
- Heterogeneous
- Created using tuple() or ()
- **Note: An *n*-tuple is a tuple with *n* items in it**
- Eg., T = (1, 2, 'abc')
- Eg., T = 1, 2, 'abc'

Sets

- Mutable
- Not indexed
- Unordered

- Heterogeneous
- No redundancy allowed in items
- Created using set() or {} with items
- Use set() to create an empty set
- Used widely to perform mathematical set operations
- Eg., S = {1, 2, 'abc'}

Dictionaries

- Stores data in the form of multiple key: value pairs
- Can be used as a non-relational data store
- Mutable
- Indexed (Using keys)
- Ordered (Using keys)
- Heterogeneous
- No redundancy of keys. Modification of the value in an existing key will not create a new key, instead, overwrites the value of the existing key
- Values can be repeated
- Iterated through a for-in loop over the keys
- Only immutable types can be used as keys
- Created using dict() or {}
- Eg., D = {1:1, 2:4, 3:9}

Lists vs. Tuples vs. Sets vs. Dictionaries

Comparison is tabulated in Table 1.

	List	**Tuple**	**Set**	**Dictionary**
Mutable	Yes	No	Yes	Yes
Redundancy	Allowed	Allowed	Not Allowed	Allowed in values, not in keys
Ordered	Yes	Yes	No	Yes
Indexed	Yes	Yes	No	Yes (Through keys)
Operator	[]	()	No operator	{}
Function	list()	tuple()	set()	dict()
Empty object	[]	()	set()	{}
Example	L = [1, 2]	T = (1, 2)	S = {1, 2}	D = {1:1, 2:4, 3:9}

Table 1: Comparison of Data Collections

range object

- An iterable with all the numbers between a given range
- range(lower, upper, [step])
  - lower: Lower limit (int)
  - upper: upper limit (Excluded) (int)
  - step: Optional, the number of items to be skipped (int)
- Eg., range(0, 11) will have integers between 0 and 10, including 10
- Eg., range(0, 11, 2) will have even integers between 0 and 10, including 10

List Functions

Major list functions are listed in Table 2.

Function	**Parameters (Type)**	**Return type**	**Purpose**
append(item)	item (Any type)	None	Inserts the given item at the end
pop()		Any type	Deletes and returns the last (Right-most) element from the list. IndexError if the list is empty
pop(pos)	pos (int)	Any type	Deletes and returns the element present at the given position. IndexError if list index out of range
insert(pos, item)	pos (int), item (Any)	None	Inserts the given item at the mentioned position
index(item)	item (Any)	int	Returns the index of the first occurance of the given item in the list. ValueError if item is absent
remove(item)	item (Any)	None	Deletes the first occurance of the given item.
extend(iter)	iter (Iterable)	None	Adds all the items of the iter to the end of the given list
count(item)	item (Any)	int	Returns the number of times the given item appears in the list
sort(reverse = False)	reverse (True or False) (Not mandatory)	None	Sorts the given list in ascending order. Alphabetical order will be used for string items. If reverse = True, the list will be sorted in descending order

Table 2: List Functions

Tuple Functions

Tuple functions are tabulated in Table 3.

Function	**Parameters (Type)**	**Return type**	**Purpose**
index(item)	item (Any)	int	Returns the index of the first occurance of the item. ValueError if item is absent
count(item)	item (Any)	int	Returns the number of times the item occurs

Table 3: Tuple Functions

Set Functions

Function	Parameters (Type)	Return type	Purpose
add(item)	item (Any)	None	Adds the given item to the set. If already present, nothing is added
remove(item)	item (Any)	None	Removes the given item from the set. KeyError if item is absent
union(sets)	sets (Any number of sets separated by comma)	set	Returns the union of all the given sets. Equivalent to set1 | set2
intersection(sets)	sets (Any number of sets separated by comma)	set	Returns the intersection of all the given sets. Equivalent to set1 & set2
difference(set2)	set2 (set)	set	Returns a set with the difference of the sets. Equivalent to set1 - set2

Table 4: Set Functions

Dictionary Functions

Dictionary Functions are tabulated in Table 5.

Function	**Parameters (Type)**	**Return type**	**Purpose**
get(key)	key (Hashable type)	Any	Returns the value of the given key if the key exists
keys()		Iterable called dict_keys	Returns an iterable with all the keys
values()		Iterable called dict_values	Returns an iterable with all the values
items()		Iterable called dict_items	Returns an iterable with (key, value) as an item
pop(key)	key (Hashable)	Value (Any)	Removes the key-value pair and returns the value. KeyError if the key is absent
popitem()		tuple (2-tuple with (key, value))	Removes the right-most key-value pair and returns the same as a 2-tuple of (key, value)

Table 5: Dictionary Functions

String Functions

String functions are tabulated in Table 6.

Function	**Parameters (Type)**	**Return type**	**Purpose**
capitalize()		str	Returns the string with the first alphabet capitalized
upper()		str	Returns a string with all the alphabets in the uppercase
lower()		str	Returns a string with all the alphabets in the lowercase
count(substring)	substring (str)	int	Returns the number of times the substring occurs in the string
find(substring)	substring (str)	int	Returns the position of the substring in the string. Returns -1 if the substring is not found
isalpha()		boolean	Returns True if the string has only alphabets
isalnum()		boolean	Returns True if the string is alphanumeric
isdigit()		boolean	Returns True if the string has digits only
isupper()		boolean	Returns True if the string is in uppercase
islower()		boolean	Returns True if the string is in lowercase
partition(substring)	substring (str)	tuple	Returns a 3-tuple of (string before the substring, substring, string after the substring)
split(sep = ',')	sep (str)	List	Splits the string at the specified sep.

Table 6: String Functions

Random Package and Functions

- Python has a built-in module to generate random numbers

- Module **random** has to be imported
- Works on a seed value
  - Used to control randomisation
  - Helps in the reproduction of results
  - Eg., If random seed value is set to 1, using a random function will generate the same results even on different systems with the same version of Python

Functions of the random package are tabulated in Table 7.

Function	**Parameters (Type)**	**Return type**	**Purpose**
seed(val = SystemTime)	val (int)	None	Sets the random number generator with val
randrange(l, u)	l (int), u (int)	int	Returns a random integer between l and u, excluding u. Optional step count can be used
randint(l, u)	l (int), u (int)	int	Returns a random integer between l and u, including u
random()		float	Returns a float between 0 and 1
choice(iter)	iter (Iterable)	item (Any)	Returns an item from iter at random
choices(iter, k = 1)	iter (Iterable), k (int)	list	Returns list of 'k' items from iter at random
shuffle(iter)	iter (Mutable Iterable)	None	Shuffles the given mutable iter
sample(iter, k)	iter (Iterable), k (int)	List	Returns a list with 'k' items from the iter

Table 7: Random Module Functions

Functions

- Functions are submodules to perform a specific set of operations
- Organised and reusable
- Calling function – The function which calls another function
- Called function – The function which is being called by another function
- Provides modularity and reusability of code
- Defined using keyword **def**

  Syntax:

  def functionName (ParamList):

  > **Body of the function**
  > **return [value/s]**

- Functions in Python can return any number of values of any type
- If no return statement is found, Python functions return NoneType
- Arguments or Parameters – Inputs given to the function
- It is not mandatory to have parameters or return values
- Functions need to be called, just defining is not enough
- Parameters in Python are **passed by reference**, not by value
- No specified type to the parameters, type decided during runtime

Types of parameters

- Functions sometimes, need parameters to work
- Parameters are of **4** types in Python
  - **Required or Mandatory Parameters**
    - Required parameters are mandatory to be passed if defined
    - Position of the required parameters matters
    - Number of required parameters in the function must match with the number of required parameters defined in the function definition, throws TypeError if not matched

      Eg.,

      ```
      def foo(a, b):
          return a + b
      foo(1, 2) # Works fine
      foo(1) # TypeError
      ```
  - **Variable length Parameters**
    - A function defined with variable length parameters can accept any number of parameters
    - All the parameters will be passed as a **single tuple** to the called function
    - To use variable length parameters, the function must have ***parameter** as one of the parameters

      Eg.,

      ```
      def foo(*a): # Accepts any number of arguments as a tuple
          print('Number of parameters: '+str(len(a)))
          print('Type: '+str(type(a)))
          return sum(a)

      foo() # Prints 0, class tuple, returns 0
      foo(1, 2, 3) # Prints 3, class tuple, returns 6
      ```

- **Keyword Arguments**
  - Keyword arguments are defined using ****keywordarguments**
  - Accepts any number of parameters
  - When keyword arguments are used in a function call, the caller identifies the arguments by the parameter name
  - Keyword arguments are passed as a single dictionary with variable names as the keys (str) and the values as the values (Any type)

    Eg.,
    ```
    def foo(**kw): # Accepts variable length keyword arguments
        print('Number of parameters: '+str(len(a)))
        print('Type: '+str(type(a)))
        for key in kw: print(kw[key])

    foo() # Prints 0, class dict
    foo(a=1, b=2) # Prints 2, class dict, 1, 2
    ```

- **Default Parameters**
  - Default parameters are optional parameters
  - Functions use predefined default values for the parameters if not values are passed during the function call
  - If values are passed, the passed values will be considered

    Eg.,
    ```
    def foo(a, b = 0): # a is required while b is default parameter
    # If 1 parameter is passed, that will be a
    # the second parameter will be 0
    # Function accepts 1 or 2 parameters only
        return a + b

    foo(10) # Returns 10, default value for second parameter will
    # be used
    foo(10, 20) # Returns 30
    foo() # TypeError: Requires one positional parameter
    ```

- Order of the parameters
  - Positional parameters (required arguments)
  - Default parameters
  - Variable-length parameters
  - Keyword parameters

    Eg.,
    ```
    def foo(positional1, positional2, default1='defaultvalue', *args, **kwargs):
        #Body
    ```

  - Valid function calls
    - With positional parameters only:

      foo(arg1, arg2)

    - With positional parameters and custom value for default parameter:

      foo(arg1, arg2, default1='customdefault')

    - With positional parameters, custom value for default parameter, and variable length parameters:

      foo(arg1, arg2, default1='customdefault', extraarg1, extraarg2)

    - With positional parameters, custom value for default parameter, variable length parameters, and keyword parameters:

      foo(arg1, arg2, default1='customdefault', extraarg1, extraarg2, kw1=a, kw2=b)

Scope and Lifetime of Variables

- Scope – Location where a variable can be accessed
  - Local (Narrowest) – Local variables are accessible only inside the current function or the current class
  - Enclosing – Variables accessible inside nested functions. Defined in the outer function and is accessible in both inner and outer functions
  - Global – Variables available anywhere in the current program
  - Built-in (Broadest) – Variables in the **builtins** module
- Lifetime – Time for which a variable is accessible
  - For local variables in functions, the lifetime is the time between the function call and termination (return statement execution)
  - For enclosing variables, the lifetime is the time between the outer function call and the termination of the outer function
  - For global variables, until the current programming session / Kernel is terminated
  - Built-in variables are always alive
- **global** keyword – Used to define a variable as global in a function

  Eg.,
  Without global keyword

```
a = 100
def foo():
    # Modifies global variable a
    a = 10
    # a is local, not global
    print(a)
print(a) # Prints global a, 100
foo() # Prints local a in foo(), 10
```

With global keyword

```
a = 100
def foo():
      global a
      a = 10
      # a is local, not global
      print(a)
print(a) # Prints global a, 100
foo() # Prints local a in foo(), 10
print(a) # Prints global a, 10, since the global variable was modified
```

- Note: Multiple variables can be declared as global in separate lines.

  Eg.,
  global a
  global b
  global c

  But, all global variables must be declared in the **first lines of the function**

Packing and Unpacking

- **Packing**
  - Putting together multiple items into a **tuple**
  - When a function is called with variable arguments, all arguments will be **packed** and sent as a tuple
- **Unpacking**
  - Picking individual items from a tuple and storing the values in multiple non-iterable variables
  - Unpacking happens when a function returns multiple values
  - Multiple values are returned as a tuple, which is later **unpacked** into individual variables
  - If a function returns n values, n variables are to be used to unpack
  - If a tuple is to be unpacked and has n items, n variables are to be used to unpack

Recursion

- A function calling itself
- Can be seen as an equivalent to looping
- **Base Case or Termination Condition** – Necessary to stop the recursion, **must not** call the same function
- **Recursive Call** – Function calling itself
- **RecursionError** is thrown in Python if the stack depth is exceeded, i.e., when no base case is mentioned in the recursive function

  This limit prevents infinite recursion from causing an overflow of the stack and crashing Python

  Example:

  Finding factorial of a number using recursion

```
def fact(n):
    # Base Case
    if n == 0 or n == 1: return 1
    # Recursive Call
    else: return n * fact(n-1)

fact(5) # Returns 120
```

- Table 8 has tracing of the program.

Count	***n***	**Return**	**Stack**
1	5	5 * fact(4)	
2	4	4 * fact(3)	5
3	3	3 * fact(2)	5 * 4 = 20
4	2	2 * fact(1)	5 * 4 * 3 = 60
5	1	1	5 * 4 * 3 * 2 = 120

Table 8: Tracing of fact(n)

Lambda Functions

- Also known as **Anonymous Functions**
- Defined using **lambda** keyword
- Syntax:

  lambda parameters: expression

  The function accepts **parameters**, evaluates an expression, and returns the value of the **expression**
- Example:

  a = lambda x, y: x + y
  # This function returns x + y when called with 2 parameters
  a(10, 20) # Returns 30
- Used mostly to perform some simple operations
- Often used as a parameter to a higher-order function (A function that takes in other functions as arguments)

map

- Syntax

  map(function, iter)

- map() accepts 2 parameters, a function with **exactly one parameter** and an **iterable** sequence
- The function passed as the parameter can be a regular function or a lambda function
- map() returns an iterable object called map object
- map object can be type-casted into any other iterable
- map (func, iter) applies the given function, **func** on **each** item of the iterable sequence, **iter** and returns an iterable object, **map**

  Eg.,
  Using map(), find the squares of the items of a list

  ```
  L = [1, 2, 3, 4]
  L1 = list(map(lambda x: x * x, L))
  print(L1)
  # Prints [1, 4, 9, 16]
  ```

  Eg.,
  Using map(), accept and display a user-input list

  ```
  s = input()
  L1 = s.split() # Splits the input string and creates a list of strings
  L = list(map(int, L1)) # Applies int() on every item of L1 and returns a
  map object, later converted to a list
  print(L)
  # Prints [1, 2, 3, 4] if the input is '1 2 3 4'
  ```

Single-line solution:

```
L = list(map(int, input().split()))
print(L)
```

- list(map(func, iter)) is equivalent to
  - Creating an empty list and a list with items
  - Creating a function with one parameter
  - Run a for-in loop on items of the non-empty list
  - Apply the function for the item
  - Append the evaluation result into the empty list

filter

- Syntax

  filter(function, iter)

- filter() accepts 2 parameters, a function with **exactly one parameter** and an **iterable** sequence
- The function passed as the parameter can be a lambda function or a regular function and should evaluate a condition
- filter() returns an iterable object, **filter**
- For every item in the iterable sequence, the function will be applied. If the function expression returns True, the item will be added to the filter object, otherwise discarded
- Always filter first and then map to avoid performing operations on items that could eventually get discarded
  Eg.,
  Using filter(), create a list of even integers

  ```
  L = [1, 2, 3, 4]
  L1 = list(filter(lambda x: x % 2 == 1, L))
  print(L1)
  # Prints [1, 3]
  ```

  Eg.,
  Using filter(), create 2 lists, one with the even integers and another with odd integers.
  Using map(), multiply the items of the odd-integer list by 5 and even-integer list by 10

  ```
  L = [1, 2, 3, 4]
  L1 = list(filter(lambda x: x % 2 == 0, L))
  L2 = list(filter(lambda x: x % 2 != 0, L))
  print(L1) # Prints [2, 4]
  ```

```
print(L2) # Prints [1, 3]
L3 = list(map(lambda x: x * 5, L2))
L4 = list(map(lambda x: x * 10, L1))
print(L3) % Prints [5, 15]
print(L4) % Prints [20, 24]
```

reduce

- **reduce** is to be imported from **functools** module

  ```
  from functools import reduce
  ```

- reduce(func, iter) accepts a function with 2 parameters and an iterable
- reduce() applies the function, **func** repeatedly and reduce the output to a single element
- reduce() returns a single value, non-iterable output

  Eg.,
  Using reduce(), find the sum of the items of a list

  ```
  from functools import reduce
  L = [1, 2, 3, 4]
  print(reduce(lambda x, y: x + y, L))
  # Prints 10
  ```

  Eg.,
  Using reduce, find the sum of first *n* natural numbers. Accept *n* from the user

  ```
  from functools import reduce
  print(reduce(lambda x, y: x + y, range(1, int(input()) + 1)))
  # Prints 21 if input (n) is 6
  ```

zip

- zip(iterables) accepts any number of iterable sequences and returns an iterable object called zip object
- The resulting **zip** object will have multiple tuples
- If *n* iterables are passed as parameters and they all have same number of items (*k*) in them, the resulting zip object will have *k* tuples each with *n* items (AKA, an *n*-tuple)
- The first tuple will have all of the first items from the iterables as items, the second tuple will have all of the second items from the iterables as items and so on
- If the iterables passed do not have the same number of elements in them, the number of tuples in the zip object will be as same as the length of the smallest iterable passed as the parameter

Eg.,
list(zip([1, 2], [3, 4], [5, 6])) — [(1, 3, 5), (2, 4, 6)]
Eg.,
list(zip([1, 2, 3], [4, 5])) — [(1, 4), (2, 5)]

Eg.,
Create a list of numbers from 1 to 5 and another list of numbers from 100 to 500.
Create a list of tuples such that each tuple will have (itemOfFirstList, itemOfSecondList)

```
L1, L2 = list(range(1, 6)), list(range(100, 501, 100))
L3 = list(zip(L1, L2))
print(L3)
# Prints [(1, 100), (2, 200), (3, 300), (4, 400), (5, 500)]
```

map vs. filter vs. reduce vs. zip

map(), filter(), reduce(), and zip() are compared in Table 9.

Benchmark	**map**	**filter**	**reduce**	**zip**
Parameters	(function, iter)	(function, iter)	(function, iter)	(*iter)
Nature of function argument	One parameter, evaluates an expression	One parameter, tests a test expression	Two parameters, evaluates an expression	
Purpose	Applies function on all items of iter	Filters items from iter based on the test expression evaluated in the function parameter	Repeatedly applies the function parameter on items of the iter	Creates an iterable with k n-tuples if k iterables of n items is passed
Return type	map (Iterable)	filter (Iterable)	Single value (Non-iterable)	zip (Iterable)
Use Case	Transforming data, applying a function to each element	Filtering data based on certain criteria	Performing computations like sum, product, etc., on an iterable	Combining multiple iterables element-wise

Table 9: map vs. filter vs. reduce vs. zip

Comprehensions – Lists

- Shortcut to for-in loop on Lists
- Using rules to create Lists
- Simpler and shorter syntax than for-in loop
- General Form of List Comprehension

  L = [**expr** for **item** in **iterable**]

  # For every item in the iterable, an expr will be evaluated and will be appended to L

  This is equivalent to:
  L = [] # Creation of an empty list
  for **item** in **iterable**: # Run a for-in loop on an iterable
      L.append(**expr**) # Evaluate an expression and append to the empty list

- List comprehension is equivalent to
  - Creating an empty list
  - Iterating over an iterable
  - Evaluating an expression for every item in the iterable
  - Appending the expression result to the list
- List Comprehension needs another iterable and some expression will be evaluated for all the items of the iterable and appended to the new list

  Eg.,
  Create a list with 'Hello' as an item based on the number of times as given by the user

  n = int(input('Number:'))
  L = ['Hello' for i in range(n)]
  # L will be ['Hello', 'Hello', 'Hello'] if n = 3

  Eg.,

Create an integer list of variable length by accepting the integers input by the user separated by a whitespace

```
s = input('Integers separated by space:')
L1 = s.split() # Creates a list of strings separated by space
L = [int(i) for i in L1]
# Every integer string in L1 will be type-casted and appended to L
```

Single line:

```
L = [int(i) for i in input('Integers separated by space: ').split()]
```

Eg.,
Using comprehension, find the square of all the items of numbers between 1 and 5, including 5.

```
L = [i * i for i in range(1, 6)]
```

- Comprehension, map(function, iterable), and for-in loop can be used interchangeably, i.e., every for loop can be written as a map() or as a comprehension or vice-versa

  Eg.,
  Accept a user-input list of integers

  Using for-in loop:

  ```
  L = []
  for i in input('Elements: ').split():
      L.append(int(i))
  ```

  Using list comprehension:

  ```
  L = [int(i) for i in input('Elements: ').split()]
  ```

Using map():

L = list(map(int, input('Element: ').split()))

- Conditional comprehension:

  new_list = [expression for variable in iterable if condition == True]

- Conditional comprehension can be used as an alternative to filter()
- For set comprehensions, use curly braces instead of square brackets

Comprehensions – Dictionaries

- Structure of Dictionary Comprehension:

  D = {key:expression for key in iterable}

- For every key in iterable, an expression will be evaluated and added to the dictionary

- Dictionary comprehension is equivalent to

  - Creating an empty dictionary
  - Iterate over an iterable and each item of the iterable will be a key
  - Evaluate an expression
  - Add the key-value pair to the dictionary

- Example

  Given a list [1, 2, 3, 4, 5], create a dictionary with **number : square** of the number

  Using for-in loop:

  ```
  d = {}
  for i in [1, 2, 3, 4, 5]:
      d[i] = i * i
  ```

  Using Dictionary comprehension:

  ```
  d = {i : i * i for i in [1, 2, 3, 4, 5]}
  ```

Iterators

- Iterable is a sequence of items
- All iterables are treated in the same way in Python, i.e., the same for-in loop works for different iterables like List, Tuple, Set, Dictionary, String etc. despite them having different behaviour
- This is possible as Python considers all of them as **Iterables**
- An iterable is a container of items
- To iterate / traverse / walk-through an iterable, an item is needed
- Iterators are objects that can be iterated upon
- An iterable object will return one object at a time using iterator
- Iterators have 2 responsibilities
  - Returning the data from a stream or container one item at a time
  - Keeping track of the current and visited items
- **Iterator Protocol**
  - **__iter__()** – Returns an iterator from the iterable
  - **__next__()** – Manually moves to the next item of the iterable, keeps track of visited items
  - **StopIteration** – Raised when the iterable is fully traversed
- Example:

```
L = [1, 2, 3, 4]
I = iter(L) # Creates an Iterator object of L
print(next(I))
# Prints 1
print(next(I))
# Prints 2
print(next(I))
# Prints 3
print(next(I))
```

```
# Prints 4
print(next(I))
# Raises StopIteration
```

- Iterators are implemented but, hidden, in for-in loops and comprehensions
- Iterators can be used with any sequences

Generators

- Simple functions that return an iterable set of items one at a time in a special way
- Generator Function – Any function with **yield** statement
- **yield** – Sends a value to the calling function but, does not terminate the function call unlike with **return**. yield **pauses** the function call when yield statement is encountered. The next function call will **resume** from the paused statement.
- Example:

```
def gen():
    print('A')
    yield 1 # Pauses here
    print('B')
    yield 2 # Pauses here again
    return None


g = gen() # Create a generator object
print(next(g)) # Prints 'A' and 1
print(next(g)) # Prints 'B' and 2
print(next(g)) # StopIteration
```