

List is a data type of **Python** used to store multiple values of different types of data at a time. List are represented with `[]`.

A list can be created by putting comma separated values between square brackets `[]`.

The following program shows creation of two lists namely **list1** and **list2** :

```
list1 = [1, 2, "one", "hi"]  
list2 = [4, 5, "hello"]
```

We can access the elements of a list by using **index** similar to accessing the individual characters of a string.

Unlike strings, lists are **mutable** i.e. we can modify the list (change the items, add items and reassign an item).

The following operations can be performed on the list :

1. Modify or reassign an item present at a particular index.
2. Add an item to the list at a particular index.
3. Remove an item from the list.

Add an item to the list

The `append()` method is used to add items to the list.

```
chars = ['a', 'b', 'c', 'd']  
chars.append("abcd")  
print(chars) # will print output as follows
```

Output :
['a', 'b', 'c', 'd', 'abcd']

Adding items of a list as individual items of another list

The `extend()` method is used to add items of one list to be added as individual items of another list.

```
chars = ['a', 'b', 'c', 'd']  
list2 = [1, 2, 3]  
chars.extend(list2)  
print(chars) # will print output as follows
```

Output :
['a', 'b', 'c', 'd', 1, 2, 3]

Note : Closely observe that the items of `list2` are added as individual items to the `chars` list.

A **set** is a **mutable** data type that contains an **unordered** collection of items.

Every element in the set should be **unique** (no duplicates) and must be **immutable** (which cannot be changed). But the set itself is **mutable**. We can **add** or **remove** items / elements from it.

Note : Mutable data types like list, set and dictionary cannot become elements of a set.

The **set** itself is mutable i.e. we can add or remove elements from the set.

The main uses of sets are:

- Membership testing
- Removing duplicates from a sequence
- Performing mathematical operations such as intersection, union, difference, and symmetric difference

A **set** is represented with { }.

Note : Empty curly braces {} does not make an empty set in Python, it makes an empty dictionary instead. Dictionary data type is introduced in the upcoming lessons.

```
test = { }  
print(type(test))
```

Output :
<class 'dict'>

Empty set

An empty set can be created using built-in **set()** function.

```
myset = set()  
print(myset) # will print empty set.  
print(type(myset)) # will print type of myset.
```

Output :
set()
<class 'set'>

Another way to create a set is to put all elements inside curly braces separated by commas.

```
myset1 = {1, 2, 3}  
print(myset1) # will print the elements of a set.  
print(type(myset1)) # will print type of myset1.
```

Output :
{1, 2, 3}
<class 'set'>

A **tuple** is a data type similar to **list**.

The major differences between the two are: Lists are enclosed in square brackets `[]` and their elements and size can be changed (**mutable**), while tuples are enclosed in parentheses `()` and their elements cannot be changed (**immutable**).

Tuples can be thought of as **read-only** lists.

Note: Since a tuple is immutable, iterating through tuple is faster than with list. This gives a slight performance improvement.

Once a **tuple** is defined, **we cannot add elements in it or remove elements from it**.

A tuple can be converted into a list so that elements can be modified and converted back to a tuple. Conversion of a tuple into a list and a list into a tuple is discussed in the later sessions.

A tuple is represented using parenthesis `()`.

Creation of a Tuple:

Tuples can be created using built-in function called `tuple()`.

An empty tuple can be created using `tuple()` function as follows :

```
tuple1 = tuple() # Creating an empty tuple using tuple() function
print(tuple1) # Printing tuple1
```

Output :
`()`

A **tuple** can be created by placing all the items (elements) inside a parentheses `()`, separated by comma. The parentheses are optional but it is a good practice to write them.

```
mytuple = (1, 2, 3, "Data types") # mytuple = 1, 2, 3, "Data types" will also work.
print(mytuple)
print(type(mytuple))
```

Output :
`(1, 2, 3, "Data types")`
`<class 'tuple'>`

Converting a tuple into a list

A tuple can be converted into a list using the built-in `list()` function.

```
tuple1 = ('hi', 'hello', 55, 66) # Creating a tuple, tuple1
print(tuple1) # Printing tuple1, output contains values enclosed in parentheses indicating a tuple
print(type(tuple1)) # Printing the type of tuple1

list1 = list(tuple1) # Converting the tuple1 into a list using list() function.
print(list1) # Printing list1, output contains values enclosed in square brackets indicating a list
print(type(list1))
```

Dictionary is an **unordered** collection of **key** and **value** pairs.

Note : While other compound data types (like lists, tuples and sets) have only value as an element, a dictionary has a **key** : **value** pair

General usage of dictionaries is to store key-value pairs like :

- Employees and their wages
- Countries and their capitals
- Commodities and their prices

In a dictionary, the **keys should be unique**, but the values can change. For example, the price of a commodity may change over time, but its name will not change.

Immutable data types like number, string, tuple etc. are used for the **key** and any data type is used for the **value**.

A dictionary with elements can be created as follows :

```
mydict = dict(Hyderabad = 20, Delhi = 30) # A dictio
print(mydict) # Prints the dictionary
```

Output :

```
{'Hyderabad': 20, 'Delhi': 30}
```

2. Assigning elements directly.

A dictionary is created using direct assignment as follows :

```
mydict = {1:"one", 2:"two", 3:"three"} # Create a dicti
print(mydict) # Printing the dictionary
print(type(mydict)) # will print output as follows
```

Output :

```
{1:"one", 2:"two", 3:"three"}
<class 'dict'>
```

The elements of the dictionary can be **retrieved/accessed** in 2 ways.

1. Using the keys of the dictionary.
2. Using the `get()` method.

1. Using the keys of the dictionary.

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} #  
print(capitals["India"]) # Printing the value with the key "India" i.e. "New Delhi".  
print(capitals["Nepal"]) # Printing the value with the key "Nepal" i.e. "Kathmandu".
```

Output :
New Delhi
Kathmandu

2. Using the `get()` method.

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} #  
print(capitals.get("India")) # Printing the value with the key "India" i.e. "New Delhi".  
print(capitals.get("Nepal")) # Printing the value with the key "Nepal" i.e. "Kathmandu".
```

Output :
New Delhi
Kathmandu

TYPE conversion

1. **int(x, base):** This function converts **x** to an integer of specified base. If base is not specified, it defaults to **10**.

The syntax of `int()` function is:

```
int(x=0, base=10)
```

- x - Number or string to be converted to integer. Default argument is zero.
- base - Base of the number in x. Can be 0 (code literal) or 2-36.

Consider the following program to understand the working of `int()` function.

```
s = "0011" # A binary string.  
print(int(s, 2)) # Converts string type to int type using int() with base 2  
print(int(s)) # base not specified, it defaults to 10
```

Output:
3
11

2. **float(x)**: This function is used to convert any data type to a floating point number. The float() function returns a floating point number from a number or a string.

The syntax of `float()` function is:

3. **ord()**: The `ord()` method returns an integer representing Unicode code point for the given Unicode character.

The syntax of `ord()` function is:

```
ord(c)
```

- c - character string of length 1 whose Unicode code point is to be found

```
print(ord('A')) # convert Unicode 'A' character to respective integer value.  
print(ord('Z')) # convert Unicode 'Z' character to respective integer value.  
print(ord('a')) # convert Unicode 'a' character to respective integer value.  
print(ord('z')) # convert Unicode 'z' character to respective integer value.
```

Output:

```
65  
90  
97
```

4. **hex(x)**: The `hex()` function converts an integer to its corresponding hexadecimal string.

The syntax of `hex()` function is:

```
hex(x)
```

- x - is an integer that is to be converted to hexadecimal string

Note: The returned hexadecimal string starts with prefix "0x" indicating that it is in hexadecimal form.

```
print(hex(45)) # Takes an int value to convert it into hexadecimal value.
```

Output:

```
'0x2d' # respective hexadecimal value for 45. 0x prefix indicates this is a hexadecimal number
```

5. **oct(x)**: The `oct()` method takes an integer and returns its octal representation.

The syntax of `oct()` function is:

```
oct(x)
```

- x - is an integer that is to be converted to an octal string

Note: The returned octal string starts with prefix "0o" indicating that it is in octal form.

```
print(oct(45)) # Takes an int value to convert it into octal value.
```

Output:

```
'0o55' # respective octal value for 45. 0o prefix indicates this is an octal number
```

6. **complex(real, imag)**: The `complex()` method returns a complex number when the real and imaginary parts are provided to a complex number.

The syntax of `complex()` function is:

```
complex(real, imag)
```

- real - real part. If real is omitted, it defaults to 0.
- imag - imaginary part. If imag is omitted, it defaults to 0.

```
print(complex(10, 3)) # Creates a complex number with real part 10 and imaginary part 3
```

Output:

```
(10 + 3j)
```

7. **str(x)**: The `str()` function is used to convert x to a string representation.

The syntax of `str()` function is:

```
str(object='')
```

```
str(object=b'', encoding='utf-8', errors='strict')
```

Return a string version of object. If object is not provided, returns the empty string. The encoding depends on whether encoding or errors is given.

- object - object whose informal representation is to be returned
- encoding - Defaults to UTF-8. Encoding of the given object
- errors - response when decoding fails.

```
print(str(100)) # convert a int value into str value
```

Output:

```
100
```

8. **eval(str)**: The `eval()` method parses the expression passed to this method and runs python expression (code) within the program.

The syntax of `eval()` function is:

```
eval(expression, globals=None, locals=None)
```

- expression: this string is parsed and evaluated as a Python expression.
- globals (optional): a dictionary to specify the available global methods and variables.
- locals (optional): another dictionary to specify the available local methods and variables.

```
a = 100
b = 3
print(eval("a + b")) # evaluate an expression using eval() function.
103
```

9. **chr()**: The `chr()` method returns a character (a string) from an integer (that represents unicode code point of the character). This is the inverse of `ord()` function.

The syntax of `chr()` function is :

```
chr(i)
```

i - integer value, The valid range of `i` is from 0 through 1,114,111(0x110000).

For example: **chr(97)** returns the string 'a', while **chr(8364)** returns the string '€'.

```
print(chr(97))
print(chr(65))
print(chr(1200))
```

Output:

```
a
A
¥
```


Python Bitwise Operators take one or two operands, and operate on them bit by bit, instead of whole. Following are the bitwise operators in Python

1. `<< (Left shift)` - Multiply by 2^{**} number of bits

Example: $x = 12$ and $x << 2$ will return **48** i.e. $(12 * (2^{**} 2))$ This is similar to multiplication and more efficient than the regular method

2. `>> (Right shift)` - divide by 2^{**} number of bits

Example: $x = 48$ and $x >> 3$ will return **6** i.e. $(48 / (2^{**} 3))$ This is similar to division by powers of 2 (2, 4, 8, 16 etc.)

3. `& (AND)` - If both bits in the compared position are 1, the bit in the resulting binary representation is 1 ($1 \times 1 = 1$) i.e. **True**; otherwise, the result is 0 ($1 \times 0 = 0$ and $0 \times 0 = 0$) i.e. **False**.

Example : $x \& y$ Does a "bitwise and". If the bit in x and the corresponding bit in y are **1**, then the bit in the result will be **1**. otherwise it will be **zero**.

$2 \& 5$ will result in **zero** because $(010 \& 101 = 000)$. $3 \& 5$ will result in **1** because $(011 \& 101 = 001)$

4. `| (OR)` - If the result in each position is 0(**False**) if both bits are 0, while otherwise the result is 1(**True**).

Example : $x | y$ Does a "bitwise or". If the bit in both operands is **zero**, then the resulting bit is **zero**. otherwise it is **1**.

$2 | 5$ will result in **7** because $(010 | 101 = 111)$ and $3 | 5$ will also result in **7** because $(011 | 101 = 111)$

5. `~ (NOT)` - The bitwise NOT, or complement, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Bits that are 0 become 1, and those that are 1 become 0.

Example : \sim Returns the complement of x . If the bit is **1** it will become **zero** and if the bit is **zero** it will become **1**.

~ 5 will result in **2** because $(\sim 101 = 010)$ and ~ 2 will become **5** because $(\sim 010 = 101)$. This is true only for unsigned integers.

6. `^ (Bitwise XOR)` - If the comparison of two bits, being 1 if the two bits are different, and 0 if they are the same.

Example: Does a "bitwise exclusive or". If the bit in either operands is **1**, but not in both, then the resultant bit will be **1**. Otherwise it will be **0**.

$5 \wedge 3$ will result in **6** because $(101 \wedge 011 = 110)$

The operators `in` and `not in` test for membership. $x \text{ in } s$ evaluates to True if x is a member of s , and False otherwise.

The Right Hand Side (RHS) can be a String, List, Tuple, Set or a Dictionary.

For strings, the Left Hand Side (LHS) can be any string. If this string exists in the RHS string, then True is returned. Otherwise False is returned.

For all other data types (Lists, Tuples, Sets, Dictionaries, etc.) the LHS should be a single element.

Let's consider an example for strings:

```
'am' in 'I am working'
```

will return `True`

```
'am' not in 'I am working'
```

will return `False`

Identity Operators : Every object has an identity, a type and a value. An object's identity never changes once it is created, you may think of it as the object's address in memory. The '`is`' operator compares the identity of two objects, the '`id()`' function returns an integer representing its identity (currently implemented as its address).

1. `is` : If two objects are in the same memory, then it means they are the same objects. So the result is **True**.

For Example :

```
a = 10
b = 10
c = a is b
```

Here `c` is `True`.

Note: Small integers (-5 to 256 both included) are `interned` in Python since they are used so often.

It's an optimization. `x = 5, y = 5, x is y => True` because `id(x) == id(y)`.

It's the same integer object which is reused. Works in Python since integers are immutable.

If you do `x = 1.0, y = 1.0` or `x = 9999, y = 9999`, it won't be the same identity, because floats and larger ints aren't interned.

2. `is not` : If two objects are in the different memory location , then it means they are the different objects and so not equal. So the result is **False**.

For Example :

```
c = a is not b
```

Here `c` is `False`.

Python provides special constructs to control the execution of one or more statements depending on a condition. Such constructs are called as `control statements` or `control-flow statements`.

The control-flow statements are of three types:

Selection Statement - is a statement whose execution results in a choice being made as to which of two or more paths should be followed.

- if construct
- if-else construct
- if-elif-else construct
- Nested if-elif-else construct

Iterative Statement - is a statement which executes a set of statements repeatedly depending on a condition.

- while loop
- for loop
- else clause on loop statements

Control flow Statement - is a statement which transfers control-flow to some other section of the program based on a condition.

- break statement
- continue statement
- pass statement

Write a program to take `capital` and `state` as input from the user, and create a Dictionary with these inputs. The program should terminate once a user enters `end`.

Q6



once

StateCap.py

```
1 st2cap = dict()
2 state = input("state or 'end' to quit: ")
3
4 # write your logic using while loop
5 while(state!='end'):
6     cap=str(input("capital: "))
7     st2cap[state]=cap
8     state=str(input("state: "))
9     # take inputs capital and state from the user and store
10
11 print(sorted(st2cap.items()))
12
```

Write a program to print the **Floyd's Triangle**. Print the result to the console as shown in the example.

Sample Input and Output:

```
n: 5
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

Q6



Floyds.py

```
1 n=int(input("n: "))
2 k=1
3 for i in range(1,n+1):
4     for j in range(1,i+1):
5         print(k,end=" ")
6         k+=1
7     print("")
```

Write a program to print the **Pascal's triangle**. Print the result to the console as shown in the example.

Sample Input and Output:

```
rows: 5
  1
 1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Q7

Pascal.py

```
1 n=int(input("rows: "))
2
3 for i in range(1,n+1):
4     for j in range(1,n-i+2):
5         print(" ",end="")
6         val=1
7     for k in range(1,i+1):
8         print(val,"",end="")
9         val=val*(i-k)//k
10    print("")
```

List is a data type of **Python** and is used to store sequence of items.
The items of a list need not be of the same data type.
Lists are **ordered sequence of items**.

Let us consider an example:

```
L1 = [56, 78.94, "India"]
```

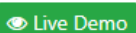
In the above example, **L1** is a list, which contains **3** elements. The first element is **56** (integer), the second is **78.94** (float), third is **"India"** (string).

Lists are ordered, we can retrieve the elements of a List using "**index**".

A List in general is a collection of objects. A List in **Python** is an ordered group of items or elements.
List can be arbitrary mixture of types like numbers, strings and other lists as well.

The main properties of Lists are :

1. Elements in a list are ordered
2. They can contain objects of different types
3. Elements of a list can be accessed using an **index** similar to accessing an element in an array
4. A List can contain other lists as their elements
5. They are of variable size i.e they can **grow** or **shrink** as required
6. They are **mutable** which means the elements in the list can be changed/modified

Click on  button to know List in **Python**.

• copy() method

Cloning using Slicing

```
a = [1, 2, 3, 4, 5]
b = a[:]
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
```

Cloning using List() function

```
a = [1, 2, 3, 4, 5]
b = list(a)
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
a[0] = 100
print(a)
[100, 2, 3, 4, 5]
print(b)
[1, 2, 3, 4, 5]
```

Cloning using copy() method

```
a = [1, 2, 3, 4, 5]
b = a.copy()
print(b)
[1, 2, 3, 4, 5]
print(a is b)
False
```

append(x)

Add a single item to the end of the list

Equivalent to **a[len(a):] = [x]**

```
x = ['a', 'b', 'c', 'd']
x.append('e')
print(x)
['a', 'b', 'c', 'd', 'e']
```

extend(iterable)

Extend the list with the items of another list or iterable

Equivalent to **a[len(a):] = iterable**

```
x = ['a', 'b', 'c', 'd']
y = [1, 2, 3, 4]
x.extend(y)
print(x)
['a', 'b', 'c', 'd', 1, 2, 3, 4]
```

insert(index, item)

Insert an item at a position before the element given by the index.

a.insert(0, x) inserts at the front of the list.

a.insert(len(a), x) is equivalent to **a.append(x)** which inserts an element at the end of list.

```
x = ['a', 'b', 'c', 'd']
x.insert(0, 1)
print(x)
[1, 'a', 'b', 'c', 'd']
x.insert(len(x), 'e')
print(x)
[1, 'a', 'b', 'c', 'd', 'e']
```

remove(element)

Remove the first item in the list whose value is element

Error if the item doesn't exist with the value element in the list

```
L1 = [1, 'a', 'b', 'c', 'd', 'e']
L1.remove(1)
print(L1)
['a', 'b', 'c', 'd', 'e']
L1.remove('f')
Traceback (most recent call last):
```

pop(), pop(index)

Removes an item at the given position specified by index

Removes and returns the last element if the index is not specified

If an invalid index is specified, then an IndexError is thrown

```
x = ['a', 'b', 'c', 'd', 'e']
x.pop(2)
'c'
print(x)
['a', 'b', 'd', 'e']
x.pop()
'e'
print(x)
['a', 'b', 'd']
```

count(x)

Return the number of times item x appears in the list

```
x = ['a', 'b', 'a', 'c', 'd', 'e', 'a']
x.count('a')
3
```

sort(key = None, reverse = False)

Sort the items of the list in place in ascending order

If the parameter reverse = True, then the list is sorted in place in descending order.

```
x = ['z', 'f', 'e', 'a', 'b', 'g', 't']
x.sort()
print(x)
['a', 'b', 'e', 'f', 'g', 't', 'z']
x.sort(key = None, reverse = True)
print(x)
['z', 't', 'g', 'f', 'e', 'b', 'a']
```

reverse()

Reverse the order of elements of the list in place

```
x = ['z', 'f', 'e', 'a', 'b', 'g', 't']
x.reverse()
print(x)
['t', 'g', 'b', 'a', 'e', 'f', 'z']
```

copy()

Return the shallow copy of the list

Equivalent to **a[:]**

```
x = ['z', 'f', 'e', 'a', 'b', 'g', 't']
y = x.copy()
print(y)
['z', 'f', 'e', 'a', 'b', 'g', 't']
print(x is y)
False
```

A `tuple` is a sequence which is similar to a list, except that these set of elements are enclosed in parenthesis `()`

Once a tuple has been created, addition or deletion of elements to a tuple is not possible(immutable).

A tuple can be converted into a list and a list can be converted into a tuple.

Functions	Example	Description
<code>list()</code>	<pre>a = (1, 2, 3, 4, 5) a = list(a) print(a) [1, 2, 3, 4, 5]</pre>	Convert a given tuple into list using <code>list()</code> function.
<code>tuple()</code>	<pre>a = [1, 2, 3, 4, 5] a = tuple(a) print(a) (1, 2, 3, 4, 5)</pre>	Convert a given list into tuple using <code>tuple()</code> function

Benefits of Tuple:

- Tuples are faster than lists.
- Since a Tuple is immutable, it is preferred over a list to have the data write-protected.
- Tuples can be used as keys in dictionaries unlike lists.

A tuple can be created by placing all elements inside parentheses `()` where each element is separated by a comma `,`.

The elements of a tuple can be of different types such as - integer, float, list, tuple, string, etc.,

```
mytup = (1, "a", 3.5 , [1, 2, 3], (4, 5, 6), "Python")
```

To create a tuple with a single element, the element should be followed with a comma `,` as shown below.

```
lonetup = (1,)
```


A set is a disordered collection with unique elements.

The elements in a Set cannot be changed (immutable).

Hence, a set cannot have a mutable element, like a list, set or dictionary as its element.

The set is mutable which means elements can be added, removed or deleted from it.

The main operations that can be performed on a set are:

- Membership test
- Eliminating duplicate entries.
- Mathematical set operations like union, intersection, difference and symmetric difference.

A Set can be created by placing all the items or elements inside curly braces `{ }` each separated by a comma `(,)`.

The `set()` built-in function can also be used for this purpose.

The Set can contain any number and different types of items - **integer, float, tuple, string** etc.

```
numset = {1, 2, 3, 4, 5, 3, 2}
print(numset)
{1, 2, 3, 4, 5}
```

An element in a set can be removed using the **discard()** and **remove()** methods. The main difference between the two methods is

when an element doesn't exist then,

- **discard()** does nothing.
- **remove()** will raise an error in such a condition.

Dictionary is an unordered collection of `key` and `value` pairs.

We use Dictionaries to store **key and value pairs** such as countries and capitals, cities and population, goods and prices etc.

The **keys should be unique**, but the values can change (The price of a commodity may change over time, but the name of the commodity will not change).

That is why we use immutable data types (Number, string, tuple etc.) for the key and any type for the value.

Dictionary is an disordered collection of elements. Dictionary contains elements in the form of (key and value) pairs.

- Unlike sequences which are indexed by a range of numbers, dictionaries are indexed by keys.
- The element in a dictionary is of type `key:value` pair.
- Hence, a dictionary is a set of disordered collection of comma separated key:value pairs which are enclosed within {} braces.
- The requirement for the Key is it should be immutable and unique and can be of types - strings, numbers, tuples.
- Tuples can also be Keys if they contain only strings, numbers or tuples.
- If any tuple contains a mutable object (such as a list), it cannot be used as a Key.
- Usually a pair of braces `{ }` represents an empty dictionary.
- Elements can be added, changed or removed by using the key.

The generalized form of declaring a dictionary is,

```
d = {key1: value1, key2: value2, ... , keyn:valuen}
```

Syntax for List Comprehensions:

```
[ expression for an item in iterate if condition ]
```

Let's consider the below example to print **1** to **10** numbers in List using **for clause**.

```
list1 = []
for i in range(1, 11): # loop repeats from 1 to 10
    list1.append(i) # every time value in 'i' will be appended to empty list.
print(list1) # finally print list with no. of elements.
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here we consider an empty List and iterate the for loop for **10** times and every element in **range(1, 11)** will be appended to empty list. and then print list.

Let's try the above example using List Comprehension:

```
[i for i in range(1, 11)] # will print the result as
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Let us discuss random functions:

1. `choice(seq)` - This function returns a random element from the non-empty sequence. If `seq` is empty, interpreter raises an `IndexError`.

```
import random
seq = "abcdefghijklmnopqrstuvw"
print(random.choice(seq)) # will print result as follows
d
```

2. `shuffle(list)` - This function returns shuffled list.

```
L1 = [10, 20, 2, 3, 1]
random.shuffle(L1)
print (L1) # will print result as follows
[1, 10, 3, 20, 2]
```

3. `randint(a, b)` - This function returns a random integer between `a` and `b` inclusive

```
print(random.randint(1, 5)) # will print result as follows
2
print(random.randint(1, 5)) # will print result as follows
5
```

4. `seed()` - This function always returns the same random value.

```
for i in range(5):
    print(random.randint(1, 100))
# will print result as follows
74
5
55
62
74
```

Here `randint()` function returns random numbers every time from given sequence.

In the above example for loop repeats for 5 times that is why we get **5** different random values.

If we want to get the same previous value every time then we use `seed()` function to **get same value** through out the loop.

```
for i in range(5):
    random.seed(10)
    print(random.randint(1, 100))
# will print result as follows
74
74
74
74
74
```

5. `random()` -using `random()` get the next random number in the range (0.0, 1.0) means a random float value **f**, such that **0** is less than or equal to **f** and **f** is less than **1**.

```
print(random.random()) # will print result as follows
0.3611586246233909
```

6. `randrange(start, stop, step)` - This function returns random values in the given sequence based on step.

```
print(random.randrange(2, 10)) # will print result as follows
5
```

Let us consider example with step 2:

```
print(random.randrange(2, 10, 2)) # will print result as follows
8
```

In **Python**, string is a contiguous sequence of Unicode Characters.

In the computer, everything is stored in binary format, i.e. `0(zero)'s` and `1(one)'s`. The elementary storage unit is called a bit, which can store either **zero** or **one**.

To represent various letters, symbols and numbers in different languages, we need **16 bits**.

Each different sequence of the 16 bits represents one symbol like A or B or C etc.

This system of representing the various existing letters, numbers and symbols (+ _ - \$ % @ ! etc) is called **UNICODE**.

Strings are represented with prefixing and suffixing the characters with quotation marks (either single quotes (`'`) or double quotes (`"`)).

An individual character within a string is accessed using an **index**. The index should always be an integer (positive or negative).

The index starts from **0 to n-1**, where `n` is the number of characters in the string.

The contents of the string **cannot** be changed after it is created.

The return value of the **Python input()** statement is, by default, a string.

A string of length `1` can be treated as a character.

[Hint: Some of the words which appear in violet are links. Click to know more about them.]

Take string as input from the console using `input()` function, print the given input string as shown in the example.

Sample Input and Output:

```
str: Strings in Python are immutable
Strings in Python are immutable
```

Python provides the following built-in **string methods** (operations that can be performed with string objects)

The syntax to execute these methods is:

```
stringobject.methodname()
```

1) **capitalize()** function is used to capitalize first letter of string.

```
a = "welcome to Python"
print(a.capitalize()) # will print result as follows
Welcome to python
```

2) **upper()** function is used to change entire string to upper case.

```
a = "welcome to Python"
print(a.upper()) # will print result as follows
'WELCOME TO PYTHON'
```

3) **lower()** function is used to change entire string to lower case.

```
a = "welcome to Python"
print(a.lower()) # will print result as follows
'welcome to python'
```

4) **title()** is used to change string to title case i.e. first characters of all the words of string are capitalized.

```
a = "this is my title test"
print(a.title()) # will print result as follows
'This Is My Title Test'
```

5) **swapcase()** is used to change lowercase characters into uppercase and vice versa.

```
a = "this is my title test"
print(a.swapcase()) # will print result as follows
'THIS IS MY TITLE TEST'
Here swapcase() function converts the entire string containing lower case letters into upper case.
```

6) **split()** is used to return a list of words separated by space.

```
a = "this is my title test"
print(a.split()) # will print result as follows
['this', 'is', 'my', 'title', 'test']
```

7) **center(width, "fillchar")** function is used to pad the string with the specified **fillchar** till the length is equal to **width**.

Let us discuss with the example:

```
a = "hello"
print(a.center(10, '&')) # will print result as follows
'&&hello&&&'
Here the width is 10 and the string length is 5, so now we need to fill the remaining width(10 - 5 = 5) with '&' special character.
```

Let us take another example on **center()** for better understanding.

```
b = "Python"
print(b.center(10, '*')) # will print result as follows
'***Python***'
```

8) **count()** returns the number of occurrences of substring in particular string. If the substring does not exist, it returns **zero**.

```
a = "happy married life happy birthday birthday baby"
print(a.count('happy')) # happy word occurred two times in a string.
2
```

```
print(a.count('birthday')) # will print result as follows
2
```

```
print(a.count('life')) # will print result as follows
1
```

9) **replace(old, new)**, this method replaces all old substrings with new substrings. If the old substring does not exist, no modifications are done.

```
a = "java is simple"
print(a.replace('java', 'Python')) # will print result as follows
'Python is simple'
```

Here **java** word is replaced with **Python**.

10) **join()** returns a string concatenated with the elements of an iterable. (Here "L1" is iterable)

```
b = '.'
L1 = ["www", "codetantra", "com"]
print(b.join(L1)) # will print result as follows
'www.codetantra.com'
```

Here **'.'** symbol is concatenated or joined with every item of the list.

11) **isupper()** function is used to check whether all characters (letters) of a string are uppercase or not.

If all characters are uppercase then returns **True**, otherwise **False**.

```
a = 'Python is simple'
print(a.isupper()) # will print result as follows
False
```

```
a = "ABC"
print(a.isupper()) # will print result as follows
True
```

12) **islower()** is used to check whether all characters (letters) of a string are lowercase or not.

If all the letters of the string are lower case then it returns **True** otherwise **False**.

```
a = "python"
print(a.islower()) # will print result as follows
True
```

13) **isalpha()** is used to check whether a string consists of alphabetic characters only or not.

If the string contains only alphabets then it returns **True** otherwise **False**.

```
a = "hello programmer"
print(a.isalpha()) # will print result as follows
False
b = "helloprogrammer"
print(a.isalpha()) # will print result as follows
True
```

If the string contains only alphabets then it returns **True** otherwise **False**

```
a = "hello23good"
print(a.isalpha()) # will print result as follows
False
```


14) **isalnum()** is used to check whether a string consists of alphanumeric characters.

Return **True** if all characters in the string are alphanumeric and there is at least one character, **False** otherwise.

Note:

- Characters those are not alphanumeric are: (space) ! # % & ? etc.
- Numerals 0 through 9 as well as the alphabets A - Z (uppercase and lowercase) came into the category of alphanumeric characters.

```
a = "alpha789"
print(a.isalnum()) # will print result as follows
True
a = "alpha"
print(a.isalnum()) # will print result as follows
True
a = "789"
print(a.isalnum()) # will print result as follows
True
a = "789 @!"
print(a.isalnum()) # will print result as follows
False
```

15) **isdigit()** is used to check whether a string consists of digits only or not. If the string contains only digits then it returns **True**, otherwise **False**.

```
a = "123"
print(a.isdigit()) # will print result as follows
True
Here the string 'a' contains only digits so it returns True.
```

```
a = "123hello"
print(a.isdigit()) # will print result as follows
False
```

16) **isspace()** checks whether a string consists of spaces only or not. If it contains only white spaces then it returns **True** otherwise **False**.

```
a = "      "
print(a.isspace()) # will print the result as follows
True
Here the string 'a' contains only white spaces so it returns True.
```

```
a = " h "
print(a.isspace()) # will print the result as follows
False
```

Here the string 'a' contains white spaces with one character so it returns **False**.

```
a = "hello python"
print(a.isspace()) # will print the result as follows
False
```

17) **istitle()** checks whether every word of a string starts with upper case letter or not.

```
a = "Hello Python"
print(a.istitle()) # will print the result as follows
True
```

`\n` it is used for providing **'new line'**

Let us consider an example:

```
print("hello\npython") # will print result as follows  
hello  
python
```

`\\` it is used to represent **backslash**:

```
print("hello\\how are you") # will print result as follows  
hello\how are you
```

It returns one single backslash.

`\'` it is used to print a Single Quote(').

```
print("It\'s very powerful") # will print result as follows  
It's very powerful
```

`\"` it is used to represent double Quote(" ")

```
print("We can represent Strings using \" ") # will print result as follows  
We can represent Strings using "
```

`\t` it is used to provide a **tab space**

```
print("Hello\tPython") # will print result as follows  
Hello   Python
```

Some times we want the escape sequence to be printed as it is without being interpreted as special.

When using with **repr()** or `r` or `R`, the interpreter treats the escape sequences as normal strings so interpreter prints these escape sequences as string in output. We use **repr()**, the built-in method to print escape characters without being interpreted as special .

We can also use both `'r'` and `'R'`. Internally they called `repr()` method.

Let us take a simple example using **repr()**

```
str = "Hello\tPython\nPython is very interesting"
print(str) # will print result as follows
Hello      Python
Python is very interesting
print(repr(str))# will print result as follows
'Hello\tPython\nPython is very interesting'
```

Let us consider another example using `'r'` and `'R'`.

```
print(r"Hello\tPython\nPython is very interesting") # will print result as follows
Hello\tPython\nPython is very interesting
print(R"Hello\tPython\nPython is very interesting") # will print result as follows
Hello\tPython\nPython is very interesting
```

18) **startswith(substring)** checks whether the main string starts with given sub string. If yes it returns **True**, otherwise **False**.

```
a = "hello python"
print(a.startswith('h')) # will print result as follows
True
```

19) **endswith(substring)** checks whether the string ends with the substring or not.

```
a = "hello python"
print(a.endswith('n')) # will print result as follows
True
```

20) **find(substring)** returns the index of the first occurrence of the substring, if it is found, otherwise it returns **-1**

```
a = "hello python"
print(a.find('py')) # will print result as follows
6
```

```
a = "hello python"
print(a.find('java')) # will print result as follows
-1
```

21) **len(a)** returns the length of the string.

```
a = 'hello python'
print(len(a)) # will print result as follows
12
```

22) **min(a)** returns the minimum character in the string

```
a = "Hello Python"
print(min(a)) # will print result as follows# min(a) prints space because space is minimum value in Hello Python.
```

Here it returns **white space** because in the given string white space is the minimum of all characters.

23) **max(a)** returns the maximum character in the string.

```
print(max(a)) # will print result as follows
y
```

A **module** is a file containing **Python definitions, functions** and **statements**.

Standard library of **Python** is extended as **modules**.

To use these modules in a program, the programmer needs to **import the modules**.

Once we import a module, we can refer to or use any of its functions or variables defined in the module of our code.

There are a large number of standard modules available in **Python**.

Standard modules can be imported in the same way as we import our user-defined modules. We will learn about modules in detail later

Syntax:

```
import module_name
```

Here we use all functions of the string module.

```
import string
print(string.punctuation) # will print result as follows
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

L40/ Basics of Functions/ What is a function

A **program** is written to solve a simple/complex problem.

Generally, a program solving simple problem is very easy to understand and also to identify mistakes, if any, in them.

The steps involved in the programs that solve large complex problems are huge and have thousands of lines of code and become difficult to understand.

So, large programs are subdivided into a number of smaller programs called **subprograms**.

Some times, we need to write a particular **block of code** that needs to be executed more than once in our program.

An error in one such block may introduce many errors in the program. This block of code can be grouped as a subprogram.

Such subprograms specify one or more actions to be performed for the larger program. These subprograms are called **functions**.

Most programming languages provide this feature called **functions**, where we need to declare and define a group of statements once and that can be called and used whenever required. This saves both **time and effort**.

So, we can define a **Function** as a self-contained block of statements that specifies one or more actions to be performed.

The main reasons for using functions are:

- To improve the readability of code.
- Improves the re-usability of the code, same function can be used in any program rather than writing the same code.
- debugging of the code would be easier if you use functions (errors are easy to be traced).
- reduces the size of the code, duplicate set of statements are replaced by function calls.

Syntax of a function:

```
def function_name(parameters):  
    """A Comment on what this functions does"""  
    statement(s)
```

- The keyword **def** always occurs as the start of function header.
- The **function_name** comes next to the **def** keyword. It follows the same rules as any identifiers of **Python** and is unique.
- The **parameters/arguments** that are passed to the function are **optional**.
- A **colon (:)** starts the function block.
- A good practice is to have a few lines of what the function does, as a **comment** in the start of the function block. This is called the **DocString**.
- There can be **one or more valid Python statements** in the function body .
- Statements must have **same indentation level**.
- A **return statement** which is **optional** can be used to return a value from the function.

As we have learnt during the syntax of the function, that a **comment** that occurs in the first line of the function body after the colon(:) is known as **Docstring**.

Let's understand the features of docstrings:

- It is called the **Docstring**.
- It is a string written between **triple quotes** `""" """`.
- It can have **multiple** lines.
- This string is available in the program as a `__doc__` attribute.
- This is a very **good practice** when you write code and add a **docstring** to the function.

ARGUMENT

- An **argument** is an expression which is passed to a function by its caller, in order for the function to perform its task.
- It is an expression in the comma-separated list bound by the parentheses in a function call expression. Arguments are local to the particular function.
- These variables are placed in the **function declaration** and **function call**.
- These arguments are defined in the **calling function**.

PARAMETER

- Parameters are variables defined in the function to receive the arguments.
- Parameters are those defined in the **function definition**.
- **Parameters** are available only within the specified function and parameters belong to the **called function**.
- **Parameters** are the local variables to the function.
- The Parameters occupy memory only when the function execution starts and are destroyed when the function execution completes.

Write a program to print a pascal triangle, and print the result as shown in the example.

Sample Input and Output:

```
num: 4  
[1]  
[1, 1]  
[1, 2, 1]  
[1, 3, 3, 1]
```

Pascal.py

```
1 def pt(n):
2     tr=[1]
3     y=[0]
4     for x in range(max(n,0)):
5         print(tr)
6         tr=[1+r for l,r in zip(tr+y,y+tr)]
7     return n>=1
8
9 n=int(input("num: "))
10 pt(n)
11
```

Notice the sequence in which the arguments are passed.

They are not in the same sequence as they are in the function definition.

This passing of arguments in any order by passing the names and assigning them to the right parameters is called **keyword arguments**.

Till now, the programs we have written can be called with **positional arguments** also.

But if we want the arguments to be only keyword arguments then we define the function with `*` as the first parameter.

Let's consider an example:

```
def nameage(*, name, age):
    print(name, age)
nameage(age = 62, name = "Ramana")
nameage(name = "Ramanamurthy", age = 62)
nameage("Ramanam", 87)
```

The first two calls will produce the correct result. The third call will give an error

```
Traceback (most recent call last):
  File "C:/Users/rsari/AppData/Local/Programs/Python/Python37/temp10.py", line 7, in
    nameage("Ramanam",87)
TypeError: nameage() takes 0 positional arguments but 2 were given
```

Regular functions are defined using the `def` keyword. Similarly, anonymous functions are defined by using the `lambda` keyword.

So, **anonymous functions** can also be referred to as **lambda functions**.

The syntax of a lambda function in Python:

```
lambda arguments: expression
```

Features of **lambda** function:

- They can be used **anywhere** we need regular function objects.
- They can have **many arguments** but can have only **one expression**.
- The **expression is evaluated first** and a value is returned.

Select the correct statements given below.

Func_Ex11.py

```
1 tax = lambda salary:(salary*20)/100
2
3 salary = int(input("Please enter your salary: "))
4 print("Tax to be paid is", tax(salary))
```

The `map(f, sequence)` function applies a function `f` to each element of a `sequence` and returns a transformed list.

If a `sequence = [a1, a2, a3, a4, ..., an]` and a function `f` is given, then

`map(f, sequence) => iterator over [f(a1), f(a2), f(a3), ..., f(an)]`

Consider the below example for your understanding:

```
list1 = [1, 2, 3, 4, 5]
def squares(x):
    return x ** 2
```

1. A **map** function can be written using,

```
print(list(map(squares, list1)))
```

2. A **lambda** function can be written using,

```
print(list(map(lambda x: x ** 2, list1)))
```

3. A **list comprehension** can be written using ,

```
print([x ** 2 for x in list1])
```

Sample Input and Output:

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```


The `filter(f, sequence)` function applies the function `f` to each element in the `sequence`.

This function returns an iterator to a sequence of elements where each element satisfies the condition in the function `f`.

Let's consider an example:

consider two lists

`a = [1, 2, 3, 5, 7, 9]`

`b = [2, 3, 6, 7, 9, 8]`

The filter function can be applied by using lambda function.

```
print(list(filter(lambda x : x in a, b)))
```

A **list comprehension** can be written using ,

```
print([x for x in a if x in b])
```

The program contains two lists. Write the missing code in the below program to print the elements that are common in both the lists using the **filter function** and **list comprehension**

Sample Input and Output:

```
[2, 3, 7, 9]
[2, 3, 7, 9]
```

Fruitful functions:

A **fruitful function** is a function which returns **value** using a `return` statement.

If we want a **function** to return a value to the statement that called the function, then the return statement is used.

To reiterate what we learnt before, a return statement consists of the **return** keyword followed by an **expression**. This expression is **evaluated** and returned to the **function caller**.

Python evaluates a fruitful function and how the function **executes a return statement**.

The below are the steps involved :

1. Inside the function body, a return statement with the **expression** is evaluated to produce a value.
2. The **value** is returned back to the caller.
3. The **control** from the function that it is executing stops and immediately returns to the location of the caller function.
4. When the control returns after a **return** statement and any statement that exists in the executing function after **return statement is ignored**.

Most of the built-in functions are fruitful functions. Every function is **fruitful** even if the body does not contain a return statement.

Referring to the previous lessons one can conclude that the function will return a special value called `None`, which has type **NoneType** if the function does not have a return statement.