DATA SCIENCE PYTHON BASICS

Module - 1

Overview of Module-1

- Why Python
- Python IDE
- Hello World Program
- Variables & Names
- String Basics
- List
- Tuple

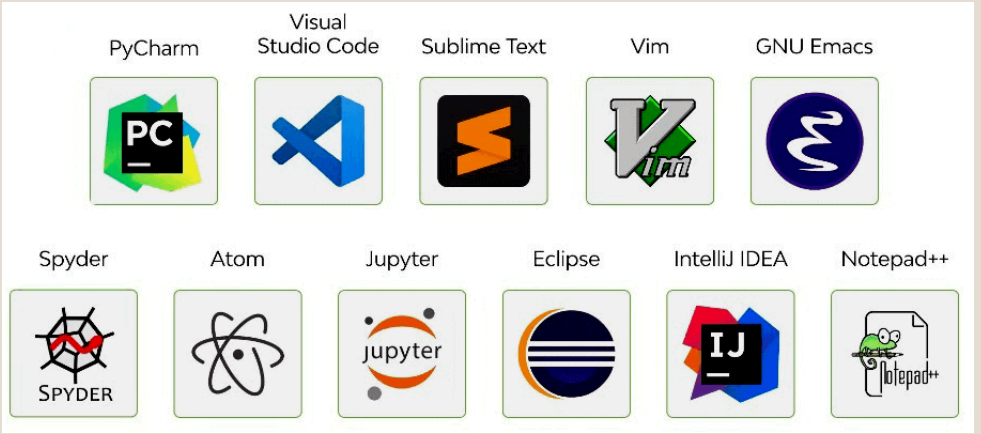
- Dictionaries
- Conditional Statements
- For and While Loop
- Functions
- Numbers and Math Functions
- Common Errors in Python

What is Python? Why?

- What kind of language Python is?
  - interpreted,
  - object-oriented,
  - high-level programming
- Provides Rapid Application Development
- simple, easy to learn syntax emphasizes readability reduces the cost of program maintenance.
- supports modules and packages, which encourages program modularity and code reuse.
- Open Source

Getting Python

- Install python
  - https://www.python.org/
- Install an Integrated Development Environment
  - VS Code
- Do not want install anything?
  - Google Colab



Are you ready to write your first program?

- Do you have python on your computer?
  - Command prompt
  - Python default IDLE
  - IDE
- Want to run online?
  - Run it in colab

Variables and Names

- Variable
  - is a named location used to store data in the memory
  - how to create a variable? print it? assign and update values?
  - can i assign multiple values to multiple variables?
  - Are there any rules to name these variables?
- Keywords:
  - reserved words in python.
  - can you name few keywords now?
- Identifier
  - is a name given to classes, functions, variables.
- Comment
  - To improve readability
  - Python Interpreter ignores comments.
  - Start with a #

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

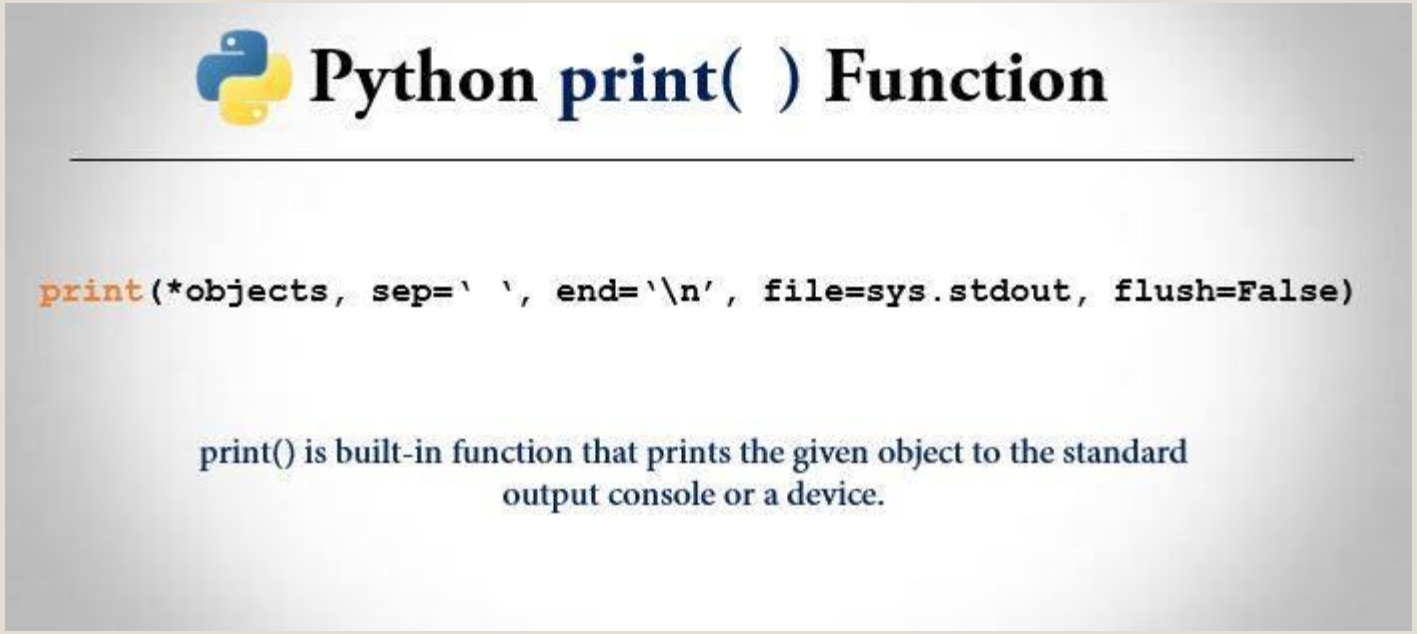
Understanding the data… and using it?

- Every value in Python has a datatype.
- How to know the data type of a value?
- Python numbers
  - integer
  - float
  - complex
- can you perform some mathematical operations on numbers?
  - +, -, *, /, %, //, **
- conversion between data types..
  - int()
  - float()
  - str()

- few more datatypes
  - Tuple
  - List
  - String
  - Set
  - Dictionary

Extend your first python program….

- Python output
  - print
- Python Input
  - input
- import



Input Statement

- Python's built-in input() function reads a string from the standard input.
- The function blocks until such input becomes available and the user hits ENTER.
- You can add an optional prompt string as an argument to print a custom string to the standard output to tell the user that your program expects their input.

*Blocks until user types in a string and hits **ENTER**.*

```
>>> s = input()
42
>>> s
'42'
>>> x = input('your input: ')
your input: 42
>>> x
'42'
```

Handy functions with input()

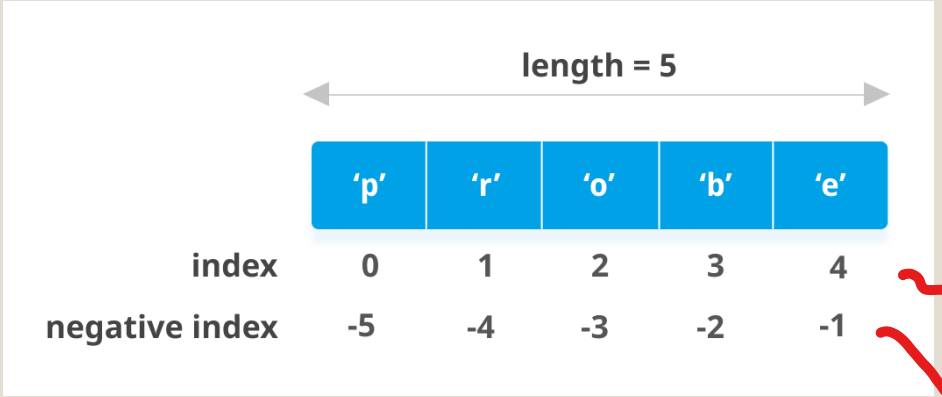
- int()
  - A built-in function that returns an integer object constructed from a number or string passed as argument or returns o if no arguments are given.
  - Usage : **int(number, base)**
- float()
  - A built-in function that returns a floating point number constructed from a number or a string.
  - Usage : **float(arg)**

```
>>> int(8)
8
>>> int(8.2)
8
>>> int(9.9e5)
990000
>>> int('0111',2)
7
>>> int('0111')
111
>>> int('0111',8)
73
>>> int('1111',16)
4369
>>> int('F',16)
15
>>> int(0b1010)
10
```

```
>>> float(1)
1.0
>>> float('1')
1.0
>>> float('infinity')
inf
>>> float(1e-2)
0.01
```

Strings

- A String is a sequence of characters.
- How to create a string in Python?
  - created by enclosing characters inside a single quote or double-quotes.
  - triple quotes can be used in Python but generally used to represent multiline strings and docstrings.
- How to access characters in a string?
  - indexing
  - usage of negative indexing
- What kind of errors we get while using Strings?
  - IndexError
  - TypeError
- Python strings are immutable
- How to delete a string from memory?

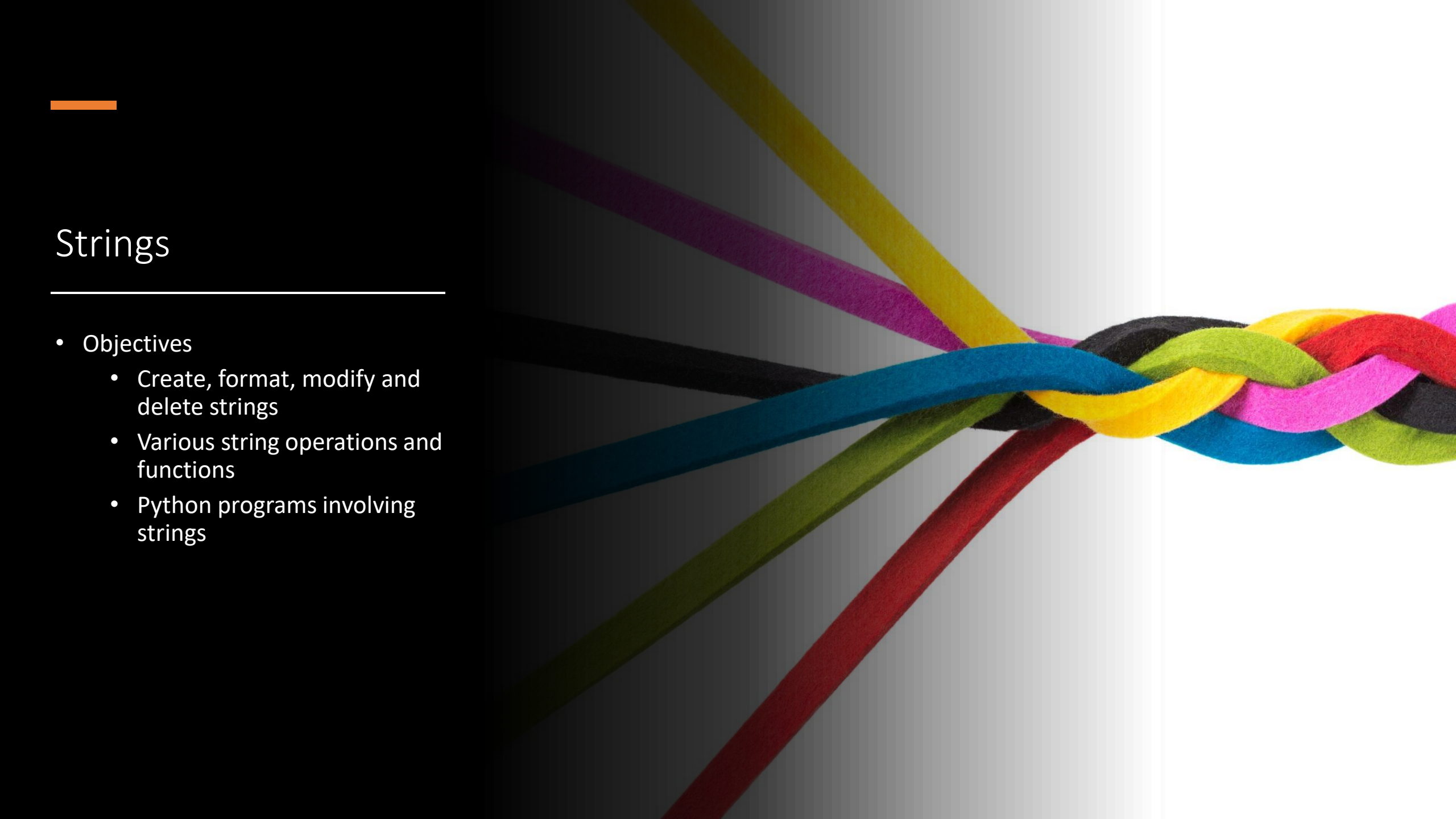


Strings contd..

- String operations
  - Concatenation (+) - Joining of two or more strings into a single one is called concatenation.
  - Repition (*) - used to repeat the string for a given number of times.
  - String membership test
- Built-in functions
  - len()
- string methods
  - lower()
  - upper()
  - replace()
  - split()
  - join()
  - find()

Strings

- Objectives
  - Create, format, modify and delete strings
  - Various string operations and functions
  - Python programs involving strings



What is a String?

- A string is a sequence of characters.
- A character is simply a symbol.
- Computers do not deal with characters; they deal with numbers (binary). The characters are internally stored and manipulated as a combination of 0s and 1s.
- This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.
- In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Creation of Strings

- Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
>>> s1 = "Bhimavaram"
>>> print(s1)
Bhimavaram
>>> s2 = 'CSE at VIT'
>>> print(s2)
CSE at VIT
>>> s3 = '''I love coding'''
>>> print(s3)
I love coding
>>> s4 = """Python is easy to learn"""
>>> print(s4)
Python is easy to learn
>>> s5 = """Computer Science and Engineering,
Vishnu Institute of Technology,
Bhimavaram"""
>>> print(s5)
Computer Science and Engineering,
Vishnu Institute of Technology,
Bhimavaram
>>> type(s1)
<class 'str'>
```

Accessing characters in a string

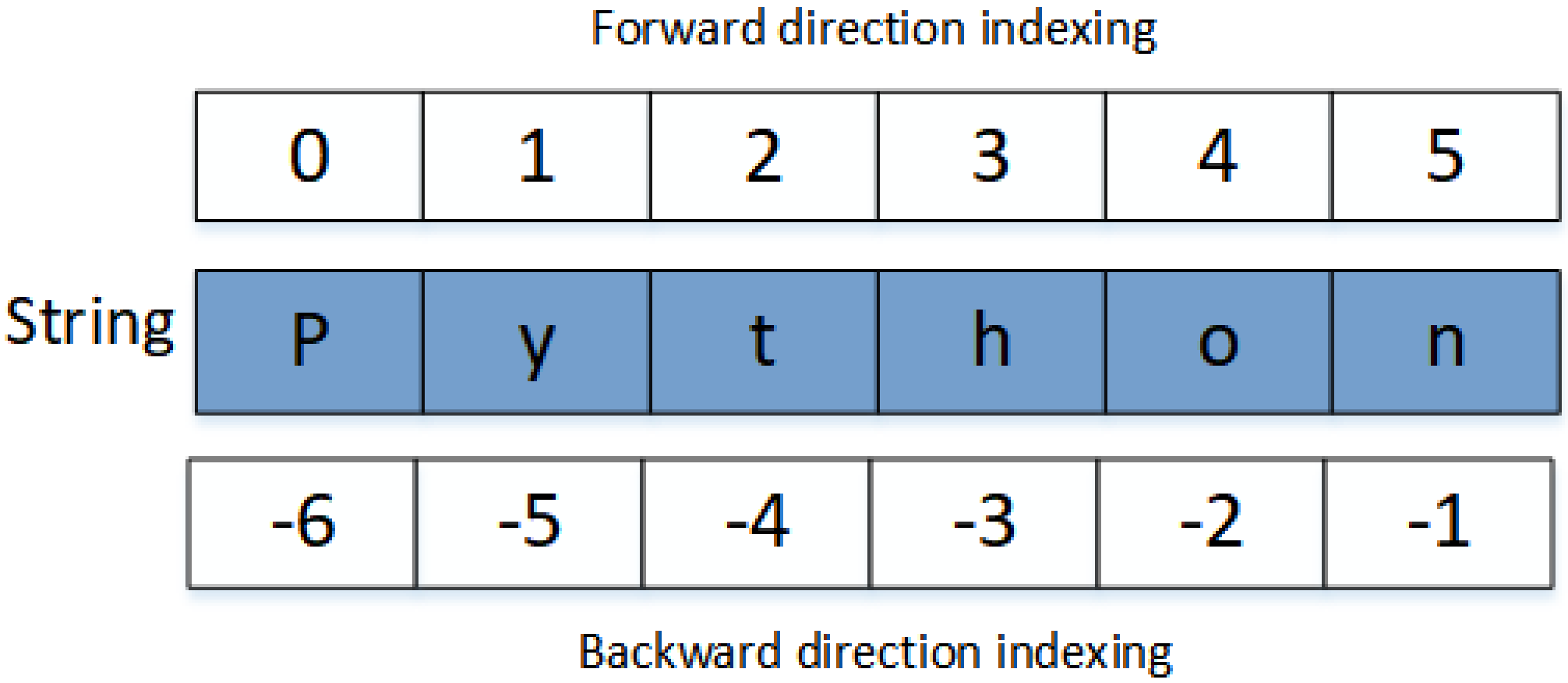
```
>>> s1 = "Technology"
>>> s1[0]
'T'
>>> s1[-1]
'y'
>>> s1[:5]
'Techn'
>>> s1[6:]
'logy'
>>> s1[2:7]
'chnol'
>>> s1[1:5]
'echn'
>>> s1[5:-2]
'olo'
```

```
>>> s1[19]
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    s1[19]
IndexError: string index out of range
>>> s1[9.4]
Traceback (most recent call last):
  File "<pyshell#29>", line 1, in <module>
    s1[9.4]
TypeError: string indices must be integers
```

- Individual characters are accessed using indexing and a range of characters using slicing - colon(:) is the slicing operator.
- Index in Strings starts from 0.
- The index must be an integer. Using float as index results in TypeError.
- Trying to access a character out of index range will raise an IndexError.
- Python allows negative indexing for its sequences.
- The index of -1 refers to the last item, -2 to the second last item and so on.

String Slicing (contd..)

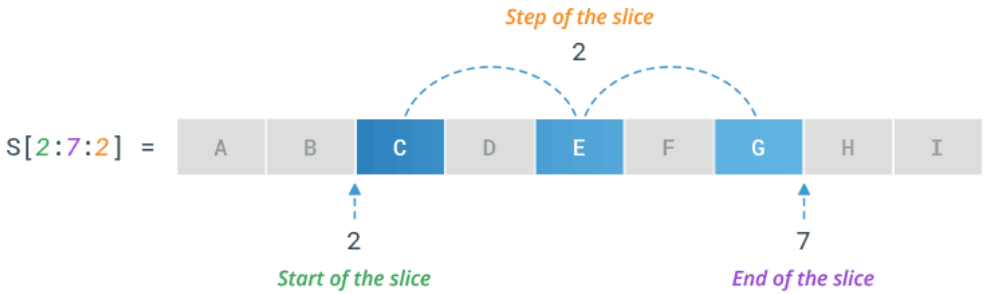
If we want to access a range, we need the index that will slice the portion from the string.



	0	1	2	3	4	5	6
word	a	m	a	z	i	n	g
	–7	–6	–5	–4	–3	–2	–1

Then,

`word[ 0 : 7 ]`	will give	`'amazing'`	(the letters starting from index 0 going up till 7 – 1 *i.e.*, 6 : from indices 0 to 6, both inclusive)
`word[ 0 : 3 ]`	will give	`'ama'`	(letters from index 0 to 3 – 1 *i.e.*, 0 to 2)
`word[ 2 : 5 ]`	will give	`'azi'`	(letters from index 2 to 4 (*i.e.*, 5 – 1))
`word[-7 : -3]`	will give	`'amaz'`	(letters from indices –7, –6, –5, –4 excluding index –3)
`word[-5 : -1]`	will give	`'azin'`	(letters from indices –5, –4, –3, –2 excluding –1)



How to change or delete a String?

```
>>> s1 = "Python"
>>> s1[0] = "C"
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    s1[0] = "C"
TypeError: 'str' object does not support item assignment
>>> del s1
>>> print(s1)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    print(s1)
NameError: name 's1' is not defined
>>> s1 = "Python"
>>> del s1[0]
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    del s1[0]
TypeError: 'str' object doesn't support item deletion
```

- Strings are immutable. This means that elements of a string cannot be changed once they have been assigned.
- We can simply reassign different strings to the same name.
- We cannot delete or remove characters from a string.
- But deleting the string entirely is possible using the del keyword.

Operations with Strings

- Strings are one of the most used data types in Python as we can perform many operations o it.
- **Cancatenation** - Joining two or more strings into a single one.
  - + operator is used to join two strings
  - * operator is used to repeat the string for a given number of times.
- **Iterating** - Strings can be iterated with the help of loops.
- **Membership test** - in keyword can be used to test whether a substring exists withing a string or not.
- **Built-in functions -** The built-in functions that work with sequences work with strings as well.
  - **len()** - returns the length of the string
  - **enumerate()** - returns an enumerate object. It contains the index and value of all the items in the string as pairs. This is useful for iterations.

```python
#program to find the length of the string
s1 = "Programming"
length = 0
for x in s1:
    length += 1
print("Length of ", s1, "is", length)
```

```
>>> s1 = "CSE"
>>> s2 = "Rocks.."
>>> s1 + s2
'CSERocks..'
>>> s1 + "  " + s2
'CSE  Rocks..'
>>> s1 * 3
'CSECSECSE'
>>> s1+" " *3
'CSE   '
>>> (s1 + " ") * 3
'CSE CSE CSE '
```

```python
#program to count the number of vowels in the string
s1 = "I am learning Python"
vowelcount = 0
for x in s1:
    if x in 'aeiou':
        vowelcount += 1
print("No.of vowels in ", s1, "is", vowelcount)
```

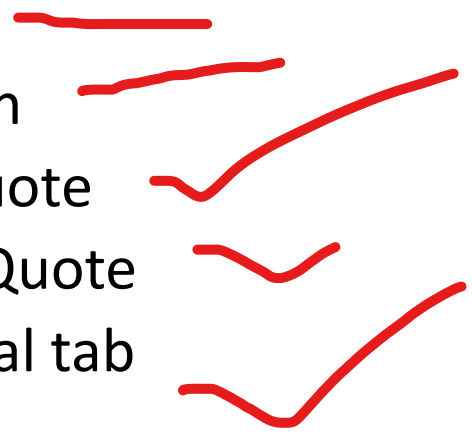
```
>>> s1 = "Python"
>>> for x in s1:
        print(x)

P
y
t
h
o
n
```

```
>>> 'a' in 'Python'
False
>>> 'a' not in 'Python'
True
```

Formatting the strings

- to print quote as part of string use either triple quotes or use a backslash
- Escape sequences supported by python
  - \n - Newline
  - \\ - Backslash
  - \' - Single Quote
  - \" - Double Quote
  - \t - Horizontal tab



String methods

- lower() - converts all uppercase characters in a string into lowercase
- upper() -
- title() - Convert String to title case
- capitalize() - Converts the first character of the string to a capital letter
- count() - Returns the number of occurrences of a substring in the string.
- endswith() - Returns "True" if a string ends with the given suffix
- startswith() -
- join() - Returns a concatenated string
- split() - Splits the string by specified character
- find() - Returns the lowest index of the substring if it is found
- replace()
- index() - returns the position of the first occurrence of a substring in a string.
- isalpha() - Returns True if all characters in the string are alphabets
- isdigit() -
- islower() -
- lstrip() - returns the string with leading characters removed.
- rstrip()
- strip()

```
>>> 'a' in 'Python'
False
>>> 'a' not in 'Python'
True
>>> s1 = "Ramayanam"
>>> len(s1)
9
>>> len("Maha Bharatam")
13
>>> s2 = "God"
>>> enumerate(s2)
<enumerate object at 0x0000025296769A40>
>>> list(enumerate(s2))
[(0, 'G'), (1, 'o'), (2, 'd')]
```

```
>>> s3 = "SITA"
>>> s1.upper()
'RAMAYANAM'
>>> s3.lower()
'sita'
>>> s4 = "i am learning python these days"
>>> s4.title()
'I Am Learning Python These Days'
>>> s4.split()
['i', 'am', 'learning', 'python', 'these', 'days']
```

```
>>> s1.isalpha()
True
>>> s1.isdigit()
False
>>> "123".isdigit()
True
>>> 'Hai'.islower()
False
>>> 'hai'.islower()
True
```

```
>>> s4.replace('e','EEE')
'i am lEEEarning python thEEEsEEE days'
>>> s4.count('e')
3
>>> "playing".endswith("ing")
True
>>> s1.startswith("Ram")
True
>>> s1.endswith("ing")
False
```

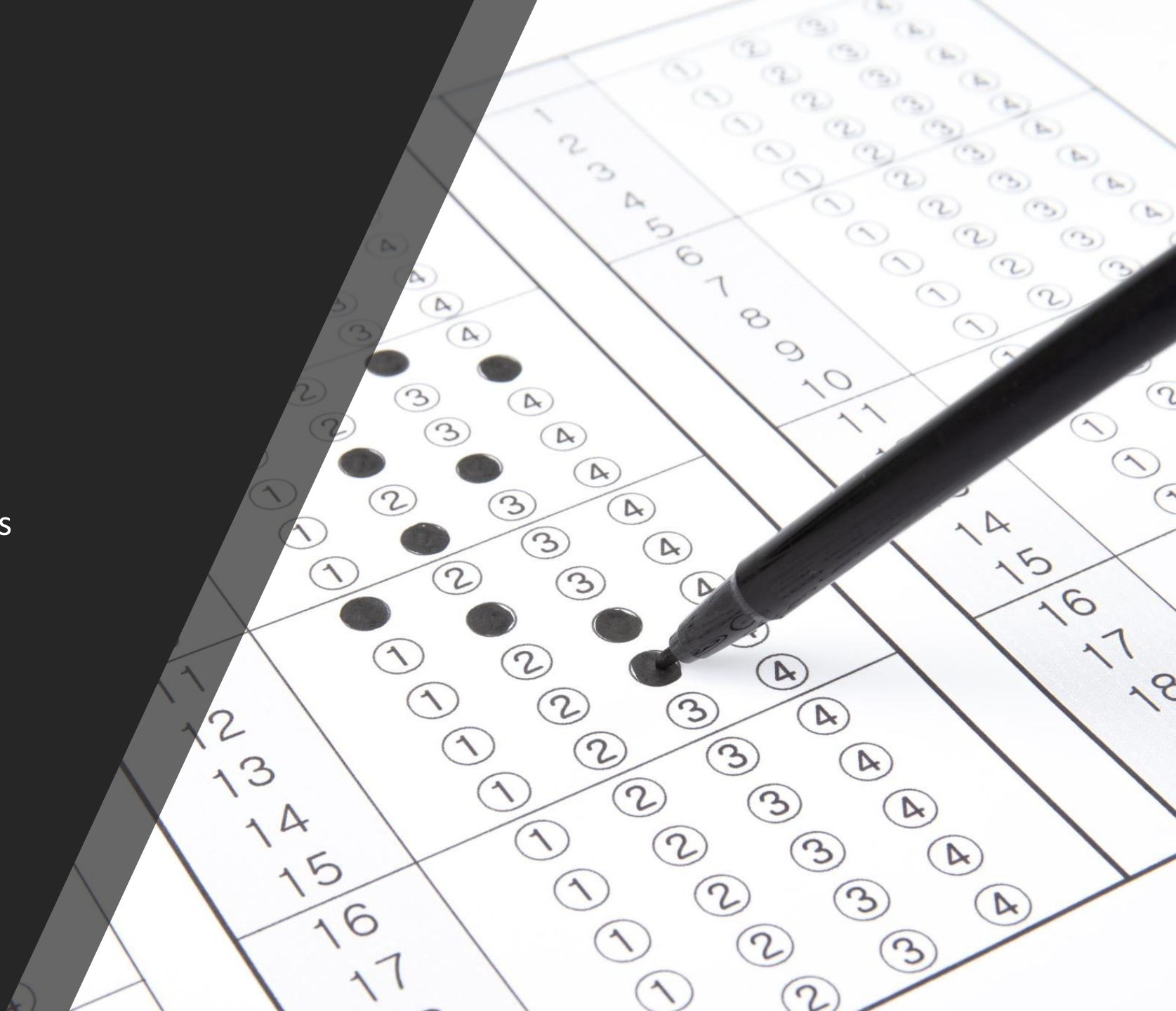
```
>>> s4.find('python')
14
>>> s4.find('java')
-1
>>> s4.find('e')
6
>>> s4.index('python')
14
>>> s4.index('java')
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <mod
    s4.index('java')
ValueError: substring not found
```

```
>>> names = ['sai', 'ram', 'lalitha']
>>> "".join(names)
'sairamlalitha'
>>> " ".join(names)
'sai ram lalitha'
>>> "-".join(names)
'sai-ram-lalitha'
>>> "\n".join(names)
'sai\nram\nlalitha'
>>> print("\n".join(names))
sai
ram
lalitha
```

```
>>> s6 = "      sairam      "
>>> len(s6)
15
>>> print(s6)
      sairam
>>> s6 = s6.strip()
>>> len(s6)
6
>>> print(s6)
sairam
```

Lists

- Objectives
  - Creation of lists
  - Changing the lists
  - Removing elements
  - List operations & methods



Lists

- Python lists are ordered sequences of items.
- For instance, a sequence of *n* numbers might be called *S*:
  $S = s_0, s_1, s_2, s_3, \ldots, s_{n-1}$
- Specific values in the sequence can be referenced using subscripts.
- Almost all computer languages have a sequence structure like this, sometimes called an *array*.
- In **other** programming languages,
  - arrays are generally a fixed size, meaning that when you create the array, you have to specify how many items it can hold.
  - Arrays are generally also *homogeneous*, meaning they can hold only one data type.

Python Lists

Python lists are dynamic. They can grow and shrink on demand.

Python lists are also *heterogeneous*, a single list can hold arbitrary data types.

Python lists are mutable sequences of arbitrary objects.

Basic list principles

- A list is a sequence of items stored as a single object.
- Items in a list can be accessed by indexing, and sublists can be accessed by slicing.
- Lists are mutable; individual items or entire slices can be replaced through assignment statements.
- Lists support a number of convenient and frequently used methods.
- Lists will grow and shrink as needed.

```
>>> L = [123, 'JAVA', 5.6]
>>> #to find number of elements in a list
>>> len(L)
3
>>> #concatenation
>>> L + [4,5,6]
[123, 'JAVA', 5.6, 4, 5, 6]
>>> #L is unchanged
>>> L
[123, 'JAVA', 5.6]
>>> #Repetition
>>> L*3
[123, 'JAVA', 5.6, 123, 'JAVA', 5.6, 123, 'JAVA', 5.6]
>>> #add an element at the end
>>> L.append('Python')
>>> L
[123, 'JAVA', 5.6, 'Python']
>>> #Removing an element
>>> L.pop()
'Python'
>>> #Removing a particular item
>>> L.pop(1)
'JAVA'
>>> L
[123, 5.6]
```

- Python’s for and in constructs are extremely useful when working with lists.
- They provide an easy way to access a list.

- for var in List:

    _____

```
>>> L = L + ['Python', 'Lists','Strings']
>>> for x in L:
        print(x)


123
5.6
Python
Lists
Strings
>>> L.sort()
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    L.sort()
TypeError: '<' not supported between instances of 'str' and 'float'
>>> L1=[2,7,5]
>>> L1.sort()
>>> L1
[2, 5, 7]
>>> L.reverse()
>>> L
['Strings', 'Lists', 'Python', 5.6, 123]
>>>
```

List Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

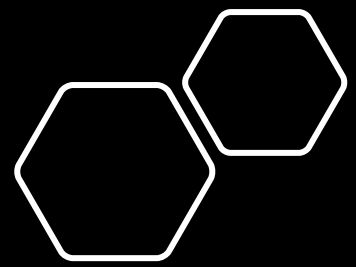
List methods

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

```
>>> #Nesting
>>> M = [ [1,2,3], [4,5,6], [7,8,9] ]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
>>> for row in M:
        print(row)


[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
>>> for row in M:
        print(row[0])


1
4
7
```



Check your understanding….

- Is list mutable? True/False

- sampleList = [10, 20, 30, 40, 50]
  - print(sampleList[-2])
  - print(sampleList[-4:-1])

- aList = [“Ramayana", [4, 8, 12, 16]]
  - print(aList[0][1])
  - print(aList[1][3])

Built-in Functions

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

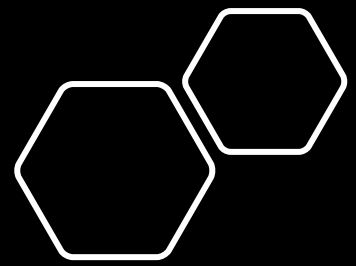
S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()



```
>>> L1 = [89, 34, 2, -1]
>>> L2 = ['Sai', 'Ram', 'Lakshmi', 1, 2, 3.4]
>>> len(L2)
6
>>> sum(L1)
124
>>> max(L1)
89
>>> min(L1)
-1
>>> sorted(L1)
[-1, 2, 34, 89]
>>> list("Bharat")
['B', 'h', 'a', 'r', 'a', 't']
```

```
>>> list(enumerate(L2))
[(0, 'Sai'), (1, 'Ram'), (2, 'Lakshmi'), (3, 1), (4, 2), (5, 3.4)]
>>> zip([1,2],['Ram','Laxman'])
<zip object at 0x000001B4945CD900>
>>> for item in zip([1,2],['Ram','Laxman']):
        print(item)


(1, 'Ram')
(2, 'Laxman')
```

Tuple

- Objectives
  - What are tuples?
  - how to create them?
  - when to use them?
  - methods of tuple

Tuple

- Another data structure supported by Python
- Similar to Lists, but it is a sequence of immutable objects
- Uses parentheses
- Creating a tuple
  - Put the comma separated values within a parenthesis
- Any set of multiple, comma separated values without an identifying symbol like [], () are treated as tuples by default.

```
>>> t1 = ()
>>> t1
()
>>> type(t1)
<class 'tuple'>
>>> t2 = (1,2,3,4,5)
>>> t3 = (1, 1.2, 'abc')
>>> t3
(1, 1.2, 'abc')
>>> len(t2)
5
```

```
>>> a = 10,20
>>> a
(10, 20)
>>> type(a)
<class 'tuple'>
>>> print(a, 'sairam',1000)
(10, 20) sairam 1000
```

More about tuples

- Can we create an empty tuple?
- How to create a tuple with single element?
- Can I create a tuple without parenthesis?
  - Tuple packing
- How to access (indexing and slicing) tuple elements?
- Is negative indexing allowed in tuple?
- Is tuple immutable?
  - Tuples are immutable. This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like a list, its nested items can be changed.

Accessing / Removing

- Index starts from 0 like lists.
- Indexing, slicing, concatenation, operations are similar to lists.

```
>>> t1 = (1,2,3,4,5)
>>> t2 = ('one', 'two', 'three')
>>> t3 = t1 + t2
>>> t3
(1, 2, 3, 4, 5, 'one', 'two', 'three')
>>> t3[:]
(1, 2, 3, 4, 5, 'one', 'two', 'three')
>>> t3[:4]
(1, 2, 3, 4)
>>> t3[3:6]
(4, 5, 'one')
>>> t3[4:]
(5, 'one', 'two', 'three')
>>> del t3[2]
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    del t3[2]
TypeError: 'tuple' object doesn't support item deletion
>>> del t3
```

Basic tuple Operations

Operation	Expression	Output
Length	len((1,2,3,4))	
Cancatenation	(1,2,3)+(4,5,6)	
Repetition	('CSE',)*3	
Membership	3 in (1,2,3)	
Iteration	for i in (1,2,3,4,5): print(i, end=' ')	
Comparison (< > ==)	(1,2,3)==(1,2,3) (1,2,3)<(3,2,1)	
Maximum	max(1,2,3)	
Minimum	min(1,2,3)	
Conversion	tuple('Hello') tuple([1,2,3])	

Features of tuple

- Tuple assignment

```
>>> t = (v1, v2, v3) = (2+4, 5/3, 9%6)
>>> t
(6, 1.6666666666666667, 3)
```

- Returning multiple values

```
>>> def max_min(vals):
        x = max(vals)
        y = min(vals)
        return(x,y)

>>> max_mark, min_mark = max_min((1,2,3,4))
>>> max_mark
4
>>> min_mark
1
>>> marks = max_min((1,2,3,4))
>>> marks
(4, 1)
```

Do you know a built-in function which returns a tuple?

```
>>> divmod(100,3)
(33, 1)
```

Methods on tuple

- Tuples did not support remove(), pop(), append(), sort(), reverse() and insert() methods like lists.
- index( ) : to obtain the index of an element in the tuple.
- count() : to return the number of elements with a specific value in a tuple.

```
>>> t1 = tuple('abcdefabcdefabgg')
>>> t1.index('f')
5
>>> t1.count('f')
2
```

zip() function

- Built-in function that takes two or more sequences and zips them into a list of tuples.
- The formed tuple has one element from each sequence

```
>>> l1 = [1,2,3]
>>> l2 = ['one', 'two', 'three']
>>> z1 = list(zip(l1,l2))
>>> z1
[(1, 'one'), (2, 'two'), (3, 'three')]
>>> for i, w in z1:
        print(i,w)


1 one
2 two
3 three
```

enumerate() function

- Gives the index along with the value
- used to traverse the elements of a sequence and also print their indices

```
>>> t = ('a', 'b', 'c')
>>> for index, ele in enumerate(t):
        print(index, ele)


0 a
1 b
2 c
```

List vs. Tuple

	List	Tuple
Syntax	List is represented with square brackets []	Tuple is represented with parentheses ()
Mutable/Immutable	List is mutable object	Tuple is immutable object
Built-in methods	List supports many built-in methods such as insert, pop, remove, etc.	Tuple doesn't support as many built-in methods as list
Storage/memory Efficiency	List consumes a lot of memory/storage	Tuple consumes a lot less storage/memory
Time Efficiency	Creation of list and accessing the list elements is slower	Creation of tuple and accessing the tuple elements is faster

SETS

- Another data structure supported by Python
- Is a mutable and an unordered collection of immutable items
- same as lists, but with no duplicate entries
- Empty set creation
  - s1 = set()
- To initialize a set with values, you can pass in a list to set()
  - pltest = set(['C', 'CPP', 'R', 'Python', 'Java'])
  - They are stored as - {'R', 'Java', 'Python', 'CPP', 'C’}
- Set containing values can also be initialized by using curly braces.
  - s2 = {'a', 'b', 'c', 'a'}
  - They will be stored as - {'c', 'a', 'b'}
- curly braces can only be used to initialize a set containing values.

```
#initialize empty set
emptySet = set()
```

Sets are a mutable collection of distinct (unique) immutable values that are unordered.

```
>>> s3 = {'a', [1,2,3], 4}
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    s3 = {'a', [1,2,3], 4}
TypeError: unhashable type: 'list'

>>> s3 = {'a', (1,2), 4}
>>> s3
{(1, 2), 'a', 4}
```

Add and Remove values from Sets

- add() - adds values to a set
  - you can only add a value that is immutable
- to remove a value from the set
  - remove()
    - the drawback of this mehtod is that if you try to remove a value that is not in your set, you will get a KeyError.
  - discard()
  - pop() - to remove an arbitrary element
  - clear() - removes all the elements

```
>>> pl
{'R', 'Java', 'Python', 'CPP', 'C'}
>>> pl.add('PHP')
>>> pl
{'R', 'Java', 'PHP', 'Python', 'CPP', 'C'}
>>> pl.remove('CPP')
>>> pl
{'R', 'Java', 'PHP', 'Python', 'C'}
>>> pl.remove('SQL')
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    pl.remove('SQL')
KeyError: 'SQL'
```

```
>>> pl
{'R', 'Java', 'PHP', 'Python', 'C'}
>>> pl.discard('PHP')
>>> pl
{'R', 'Java', 'Python', 'C'}
>>> pl.pop()
'R'
```

- Iterate through a set
  - skillset = {‘R’, ‘C’, ‘Python’}
  - for ss in skillset:

    print(ss)
- Transform the set into ordered values
  - sorted(skillset)
  - sorted(skillset, reverse=True)

Remove Duplicates from a list

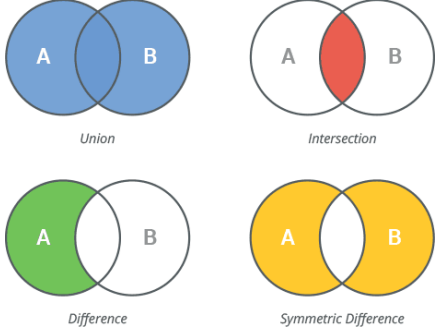
- Approach 1 : Use a set to remove Duplicates from the list

```
>>> L = [1,2,3,1,7,3]
>>> set(L)
{1, 2, 3, 7}
>>> list(set(L))
[1, 2, 3, 7]
>>> sorted(set(L))
[1, 2, 3, 7]
```

- Use a list comprehension

```
>>> unique=[]
>>> [unique.append(n) for n in L if n not in unique]
```

Set Operations



- A common use of sets in Python is computing standard math operations such as union, intersection, difference, and symmetric difference.
  - union() - finds out all the unique values in two sets
  - intersection() - returns the values which are common to both the sets
  - isdisjoint() - when the intersection is empty then the sets are referred as disjoint. This method is used to find whether they are disjoint sets or not.
  - difference() - returns the set difference
  - symmetric_difference() - returns the set of all values that are values of exactly one of two sets, but not both.
  - issubset() – to determine whether a set is subset of another set

Operation	Equivalent	Result
`len(s)`		number of elements in set *s* (cardinality)
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

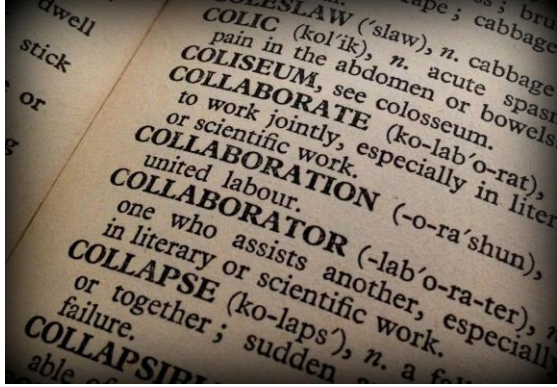
```
>>> fyear = {'C', 'CPP', 'Math', 'English' }
>>> syear = {'Python', 'CPP', 'DS', 'English', 'R', 'Java' }
>>> fyear.union(syear)
{'DS', 'R', 'Java', 'Math', 'Python', 'CPP', 'C', 'English'}
>>> fyear | syear
{'DS', 'R', 'Java', 'Math', 'Python', 'CPP', 'C', 'English'}
>>> fyear.intersection(syear)
{'English', 'CPP'}
>>> fyear & syear
{'English', 'CPP'}
>>> fyear.isdisjoint(syear)
False
>>> fyear - syear
{'Math', 'C'}
>>> syear.difference(fyear)
{'DS', 'Python', 'R', 'Java'}
>>> fyear.symmetric_difference(syear)
{'DS', 'Math', 'Python', 'R', 'Java', 'C'}
>>> fyear ^ syear
{'DS', 'Math', 'Python', 'R', 'Java', 'C'}
```

Set Comprehension

- Similar to List and Dictionary comprehensions
- The idea behind using set comprehensions is to let you write and reason in code the same way you would do mathematics by hand.

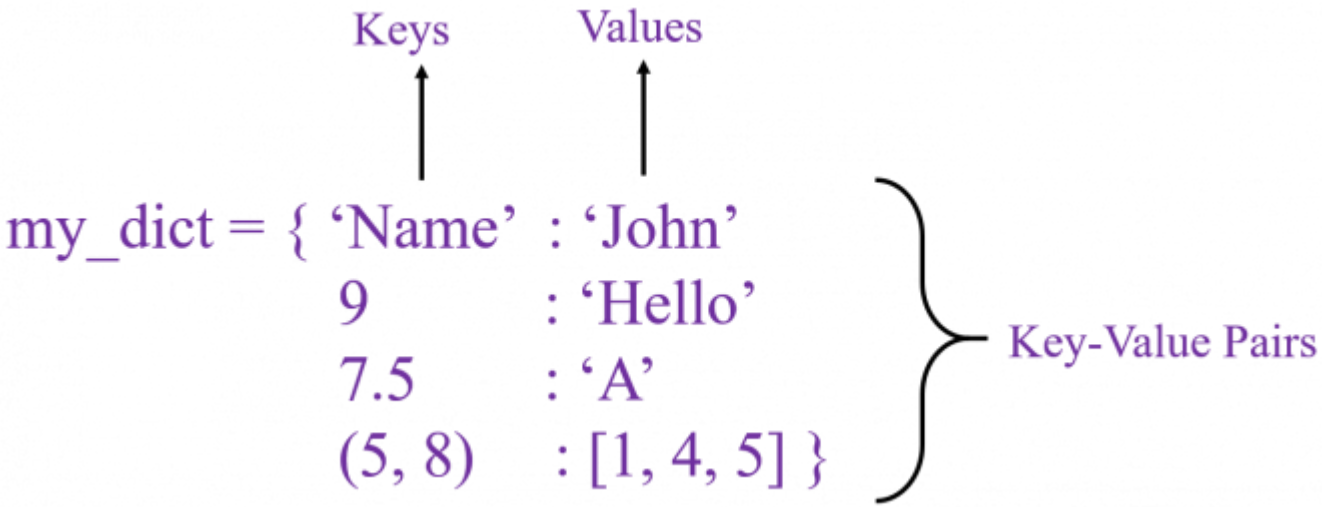
  - {skill for skill in ['SQL', 'SQL', 'PYTHON', 'PYTHON']}
  - {skill for skill in ['GIT', 'PYTHON', 'SQL'] if skill not in {'GIT', 'PYTHON', 'JAVA'}}

Dictionaries



- Dictionaries are mappings, they are not sequences
- Mappings are collections of object that store objects by key instead of relative position.

```
{'key': 'value'}
```



Dictionaries

- Data structures in which we store values as a pair of key and value.
  - Key and value are separated by colon.
  - Consecutive items are separated by comma.
  - All items are enclosed in a pair of curly brackets
- Syntax :

dictionary_name = { key1 : value1, key 2 : value2, ...
Keyn : value n }

- Keys in the dictionary must be unique and be of any immutable data type (like strings, numbers or tuples)
- There is no rule on uniqueness and type of values i.e., value of a key can be of any type

Creating a dictionary

- Using key-value pairs

```
>>> D = {'fruit':'Apple', 'qty':5, 'color':'Red'}
>>> D['fruit']
'Apple'
>>> D
{'fruit': 'Apple', 'qty': 5, 'color': 'Red'}
```

- Using dict() function
  - Creates a dictionary from a sequence of key value pairs

```
>>> D = dict([('regdno', '16PA1A05C6'), ('Name', 'Pramod'),('Marks',12)])
>>> D
{'regdno': '16PA1A05C6', 'Name': 'Pramod', 'Marks': 12}
```

- Using Dictionary comprehensions

```
>>> SD = { x : 2*x for x in range(1,5) }
>>> SD
{1: 2, 2: 4, 3: 6, 4: 8}
```

- To access, square brackets are used along with the key to obtain its value
- To add a new entry, the syntax is
  - Dictionary_variable[key] = value
- To modify just overwrite the existing value

```
>>> D = {'fruit':'Apple', 'qty':5, 'color':'Red'}
>>> D['fruit']
'Apple'
>>> D['price'] = 30
>>> D
{'fruit': 'Apple', 'qty': 5, 'color': 'Red', 'price': 30}
>>> D['qty'] = 6

>>> D

{'fruit': 'Apple', 'qty': 6, 'color': 'Red', 'price': 30}
```

Deleting items

- del can be used to remove a key-value pair or the whole dictionary
- clear() method deletes all entries
- pop() method is also used to remove an item from the dictionary.

```
>>> D = {'fruit':'Apple', 'qty':5, 'color':'Red'}
>>> dir()
['D', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
_package__', '__spec__']
>>> del D['color']
>>> D
{'fruit': 'Apple', 'qty': 5}
>>> D.clear()
>>> D
{}
>>> del D
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__pa
age__', '__spec__']
```

Nesting

```
>>> rec = {'name' : {'first' : 'Sai', 'second' : 'Vamsika'}, 'jobs' : ['dev','mgr'],
'age':45}
>>> rec['name']
{'first': 'Sai', 'second': 'Vamsika'}
>>> rec['name']['first']
'Sai'
```

Membership

```
>>> D
{'fruit': 'Apple', 'qty': 5, 'color': 'Red'}
>>> 'fruit' in D
True
>>> 'size' in D
False
>>> if 'fruit' in D:
        print(D['fruit'])


Apple
```

Sorted() and keys()

- keys() method returns the keys used in the dictionary.
- sorted() is used to sort the keys.

```
>>> D.keys()
dict_keys(['fruit', 'qty', 'color'])
>>> sorted(D.keys())
['color', 'fruit', 'qty']
```

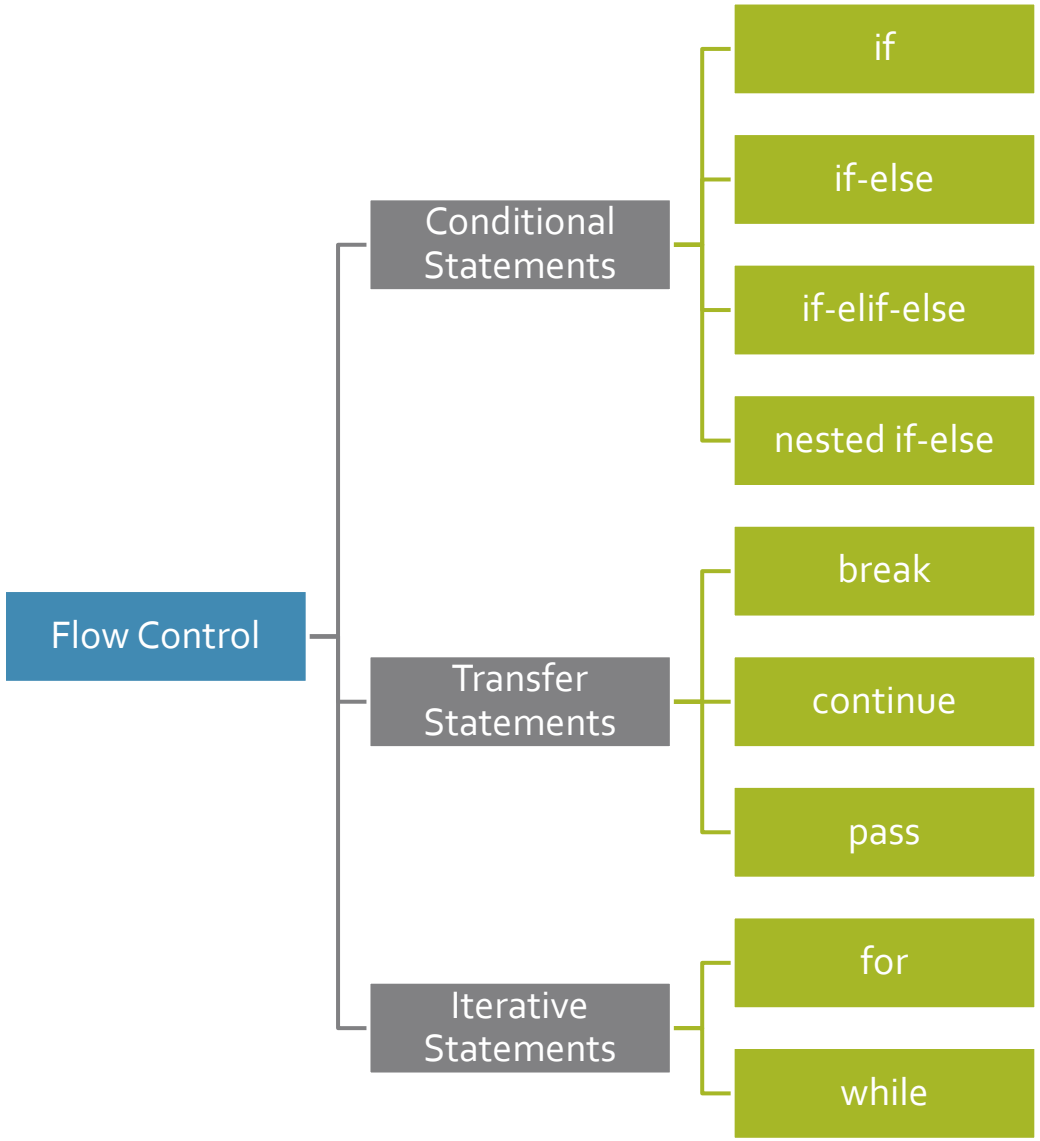
Lists from Dictionaries

- It's possible to create lists from dictionaries by using the methods items(), keys() and values().
- keys() creates a list, which consists solely of the keys of the dictionary.
- values() produces a list consisting of the values.
- items() can be used to create a list consisting of 2-tuples of (key,value)-pairs:

```
>>> D.keys()
dict_keys(['fruit', 'qty', 'color'])
>>> D.values()
dict_values(['Apple', 5, 'Red'])
>>> D.items()
dict_items([('fruit', 'Apple'), ('qty', 5), ('color', 'Red')])
```

Built-in Dictionary methods and functions

Operation	Description
len(dict)	Returns the number of items in the dictionary.
str(dict)	Returns the string representation of the dictionary
Dict.clear()	Removes all entries in the dictionary
Dict.get_key()	Returns the value for the key passed as argument
Dict.has_key()	Returns true if the key is present in the dictionary otherwise false.
Dict.keys()	
Dict.values()	
Dict.items()	


Flow Control
Conditional Statements
if
if-else
if-elif-else
nested if-else
Transfer Statements
break
continue
pass
Iterative Statements
for
while

Conditional Statements

- conditional statements act depending on whether a given condition is true or false.
- Different blocks of code can be executed depending on the outcome of a condition. Condition statements always evaluate to either True or False.
- There are four types of conditional statements.
  1. if statement
  2. if-else
  3. if-elif-else
  4. nested if-else

if statement

- In control statements, The if statement is the simplest form
- It takes a condition and evaluates to either True or False.
- If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of coc is skipped, and the control moves to the next line

Syntax of the `if` statement

```
if condition:
    statement 1
    statement 2
    statement n
```

Example

```
number = 6
if number > 5:
    # Calculate square
    print(number * number)
print('Next lines of code')
```

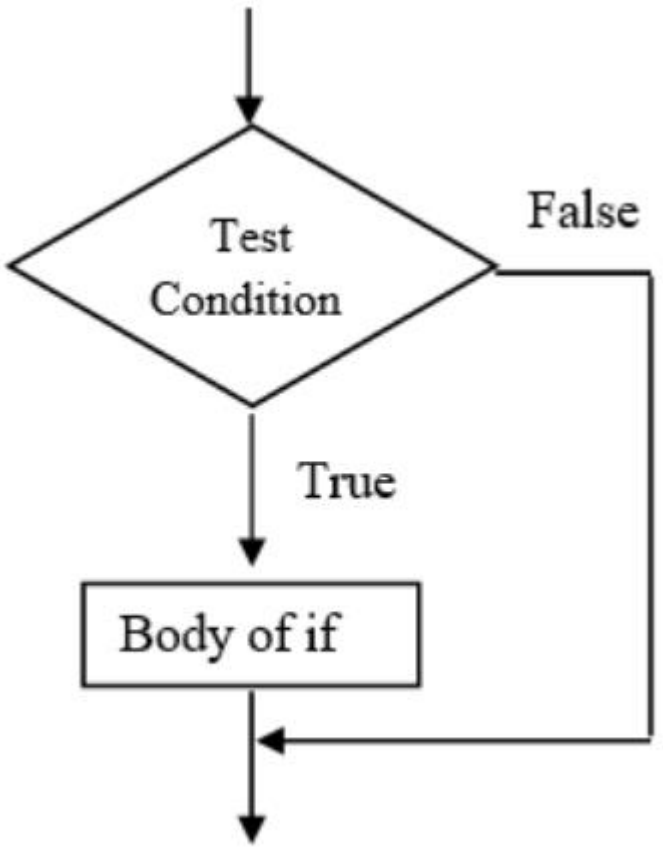
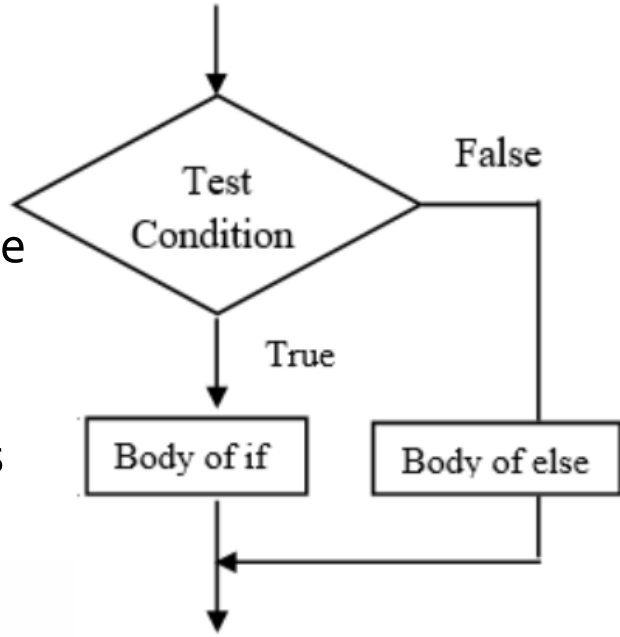


Fig. Flowchart of if statement

if – else statement



- The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.
- If the condition is True, then statement 1 will be executed If the condition is False, statement 2 will be executed. See the following flowchart for more detail.

Syntax of the `if-else` statement

```
if condition:
    statement 1
else:
    statement 2
```

Example

```
password = input('Enter password ')

if password == "PYnative@#29":
    print("Correct password")
else:
    print("Incorrect Password")
```

if-else ladder

- the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.
- The elif statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the `if-elif-else`

```
if condition-1:
    statement 1
elif condition-2:
    stetement 2
elif condition-3:
    stetement 3
    ...
else:
    statement
```

```
if choice == 1:
    print("Admin")
elif choice == 2:
    print("Editor")
elif choice == 3:
    print("Guest")
else:
    print("Wrong entry")
```

Nested if-else statement

- the nested if-else statement is an if statement inside another if-else statement.
- It is allowed in Python to put any number of if statements in another if statement.
- Indentation is the only way to differentiate the level of nesting.
- The nested if-else is useful when we want to make a series of decisions.

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers

```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))

if num1 >= num2:
    if num1 == num2:
        print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
else:
    print(num1, 'is smaller than', num2)
```

My job is over….
It is your turn now……

```
#check whether atleast one number ends with zero in the given two numbers
a = int(input())
b = int(input())
if                          :
    print('YES')
else:
    print('NO')
```

Leap year

Statement

Given the year number. You need to check if this year is a leap year. If it is, print `LEAP`, otherwise print `COMMON`.

The rules in Gregorian calendar are as follows:

- a year is a leap year if its number is exactly divisible by 4 and is not exactly divisible by 100
- a year is always a leap year if its number is exactly divisible by 400

Warning. The words `LEAP` and `COMMON` should be printed all caps.

- Given an integer, print "YES" if it's positive and print "NO" otherwise.
- Check whether the given number is EVEN or ODD
- Check whether the given number ends with seven or not.
- Given two numbers, print the smaller value
- Given three numbers, print the largest value
- Given three numbers, print them in sorted order
- Given two integers, print "YES" if they're both odd and print "NO" otherwise.
- Given two integers, print "YES" if at least one of them is odd and print "NO" otherwise.
- For the given integer X print 1 if it's positive, -1 if it's negative, or 0 if it's equal to zero. Try to use the cascade if-elif-else for it.

Given an integer, n, perform the following conditional actions:

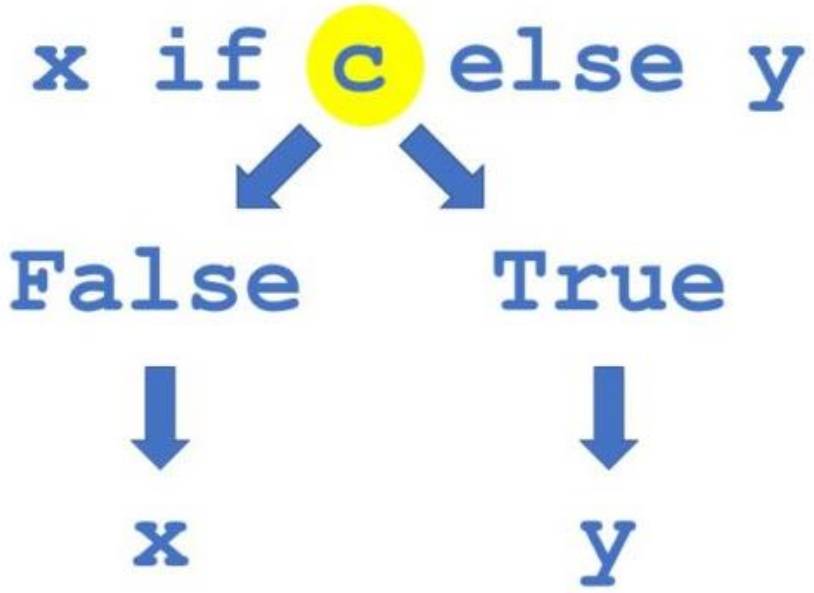
- If n is odd, print `Weird`
- If n is even and in the inclusive range of 2 to 5, print `Not Weird`
- If n is even and in the inclusive range of 6 to 20, print `Weird`
- If n is even and greater than 20, print `Not Weird`

Writing if-else as single statement

- Whenever we write a block of code with multiple if statements, indentation plays an important role. But sometimes, there is a situation where the block contains only a single line statement.
- Instead of writing a block after the colon, we can write a statement immediately after the colon.

```
number = 56
if number > 0: print("positive")
else: print("negative")
```

If-else with ternary operator



Rewrite this with ternary operator

```
if (a>b):
    big = a
else:
    big = b
```

```
>>> age = 15
>>> # Conditions are evaluated from left to right
>>> print('kid' if age < 18 else 'adult')
```

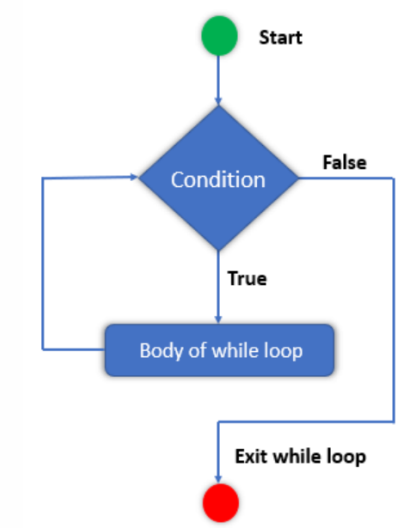
Iterative Statements

- iterative statements allow us to execute a block of code repeatedly as long as the condition is True.
- These are also called as loop or repetitive statements
- Python provides us the following two loop statement to perform some actions repeatedly
  1. for loop
  2. while loop

while loop

Syntax of while Loop

```
while test_expression:
    Body of while
```



- is used to iterate over a block of code as long as the test expression (condition) is true.
- We generally use this loop when we don't know the number of times to iterate beforehand.
- In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.
- In Python, the body of the while loop is determined through indentation.
- The body starts with indentation and the first unindented line marks the end.
- Python interprets any non-zero value as True. None and 0 are interpreted as False

```python
n = 5
i = 0
while i<5:
    print("CSE - B")
    i = i + 1
print("Done")
```

```
CSE - B
CSE - B
CSE - B
CSE - B
CSE - B
Done
```

```python
n = 5
i = 1
while i<=5:
    print(i)
    i = i + 1
print("Done")
```

```
1
2
3
4
5
Done
```

Few more

1. Write a program to find out sum of first N numbers
2. Write a program to find out sum of odd numbers in the first N numbers

   $1 + 3 + 5 + \ldots$

3. Write a program to find out sum of squares of even numbers in the first N numbers

   $2^2 + 4^2 + 6^2 \ldots$

4. Write a program to find out factorial of a number
5. Write a program to find the number of digits in a given number

   input : 867

   output : 3

6. Write a program to find the sum of digits in a given number

   input : 1234

   output : 10

Nesting of while loop

- A loop inside another loop is called as nested loop.
- In the nested while loop, the number of iterations will be equal to the number of iterations in the outer loop multiplied by the iterations in the inner loop. In each iteration of the outer loop inner loop execute all its iteration.

```
Outer loop :  1
        Inner loop :  1
        Inner loop :  2
Outer loop :  2
        Inner loop :  1
        Inner loop :  2
Outer loop :  3
        Inner loop :  1
        Inner loop :  2
** Done **
```

```
x = 3
y = 2
i = 1
while i<=x:
    print("Outer loop : ", i)
    j = 1
    while(j <= y):
        print("\tInner loop : ", j)
        j = j + 1
    i = i + 1
print("** Done **")
```

```
while expression:
    while expression:
        statemen(s) of inner loop
    statemen(s) of outer loop
```

Few more programs

Write programs to print the following patterns

```
*
* *
* * *
* * * *
* * * * *
```

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

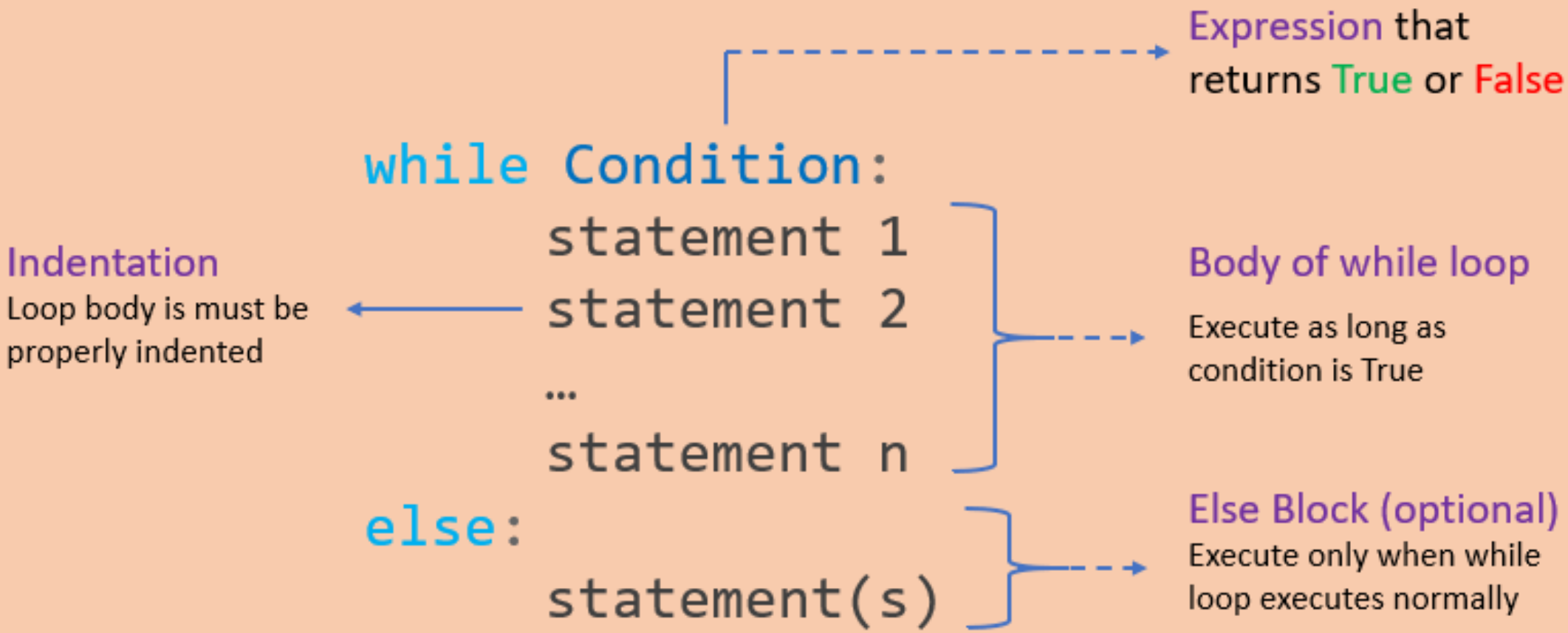
```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
A
B B
C C C
D D D D
E E E E E
```

```
A
B C
D E F
G H I J
K L M N O
```

Python While loop

While loops **repeat the same code as long as a certain condition is true**



while loop with else

- we can use else block in the while loop, which will executed when the loop terminates normally.
- else block in while loop is optional.
- The else block will not execute in the following conditions:
  - while loop terminates abruptly
  - The break statement is used to break the loop

```
i=1
while i<=5:
    print(i)
    i = i + 1
else:
    print("Loop exited normally.")
print("Done")
```

```
1
2
3
4
5
Loop exited normally.
Done
```

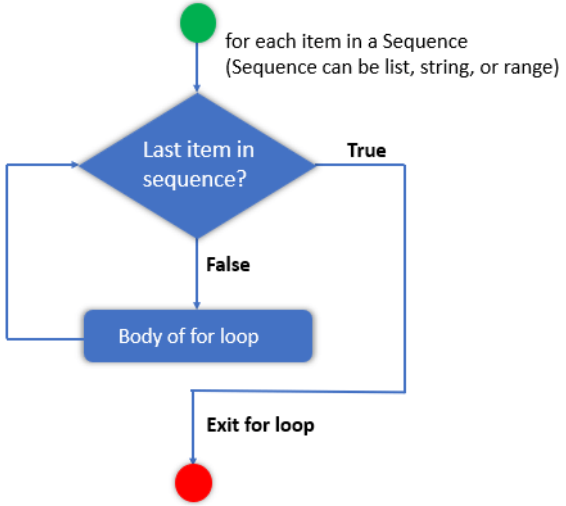
```
i=1
while i<=5:
    print(i)
    if i == 3:
        break
    i = i + 1
else:
    print("Loop exited normally.")
print("Done")
```

```
1
2
3
Done
```

for loop

Syntax of for Loop

```
for val in sequence:
    loop body
```



- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects.
- Iterating over a sequence is called traversal.
- In syntax val is the variable that takes the value of the item inside the sequence on each iteration.
- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Example program

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

range() function

- Used to generate a sequence of numbers
- Range(10) will generate numbers of **0 to 9**
- We can also define the start, stop and step size as **range(start, stop,step_size)**
- This function does not store all the values in memory; It remembers the start, stop, step size and generates the next number on the go.
- To output all the items, we can use list()
- We can use the range() function in for loops to iterate through a sequence of numbers.
- It can be combined with the **len()** function to iterate through a sequence using indexing.

```
>>> range(10)
range(0, 10)
>>> print(range(10))
range(0, 10)
>>> L = list(range(10))
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(3, 8))
[3, 4, 5, 6, 7]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
>>> list(range(10,2))
[]
>>> list(range(10,2,-1))
[10, 9, 8, 7, 6, 5, 4, 3]
>>> list(range(5, 101, 5))
```

Hey! What is the output of this?

```python
L = ['Akhilesh', 'Suneel', 'Sujan']
for x in L:
    print(x)

for i in range(len(L)):
    print(L[i])
```

```python
for i in range(10):
    print(i)
```

for loop with else

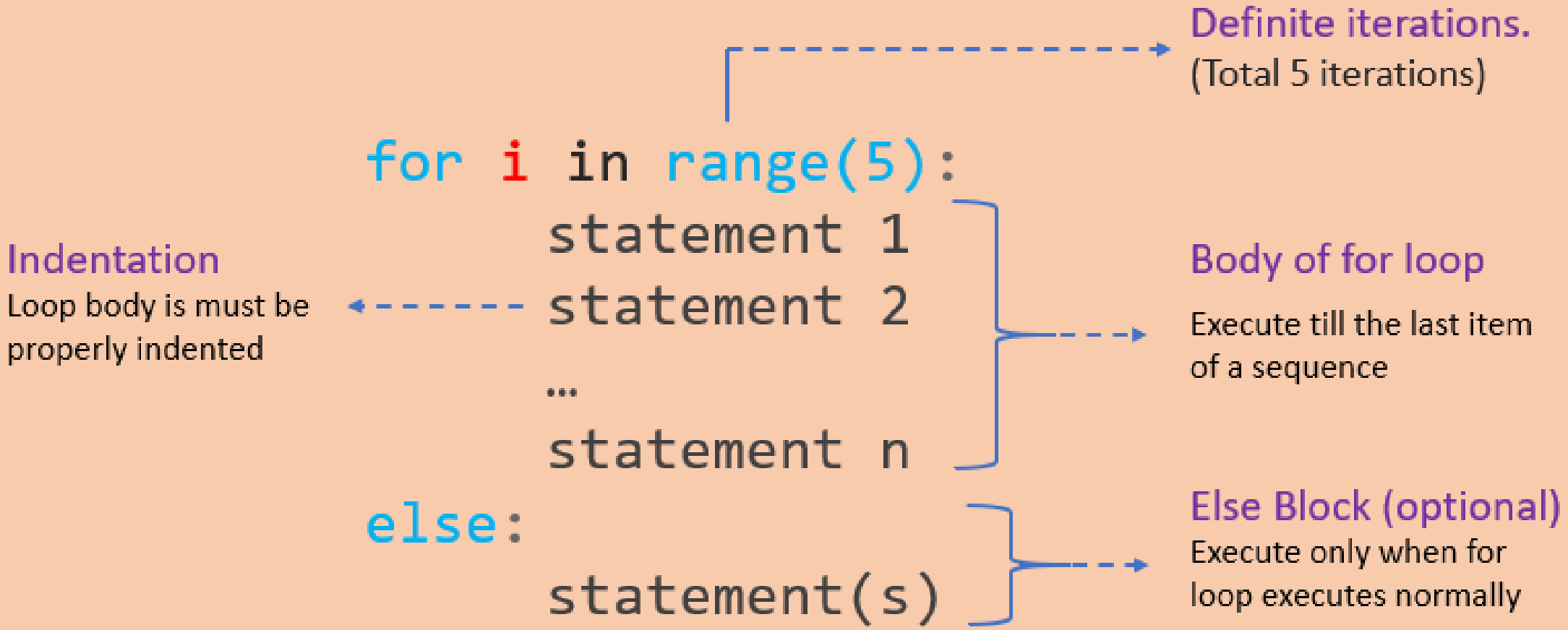
- A for loop can have an optional else block as well.
- The else part is executed if the items in the sequence used in for loop exhausts.
- The break keyword can be used to stop a for loop. In such cases, the else part is ignored.
- Hence, a for loop's else part runs if no break occurs.

```
for i in range(5):
    print(i)
else:
    print("No items left")
print("Done")
```

```
for i in range(5):
    print(i)
    if i==2:
        break
else:
    print("No items left")
print("Done")
```

Python for loop

A for loop is **used for iterating over a sequence and iterables** (like range, list, a tuple, a dictionary, a set, or a string).

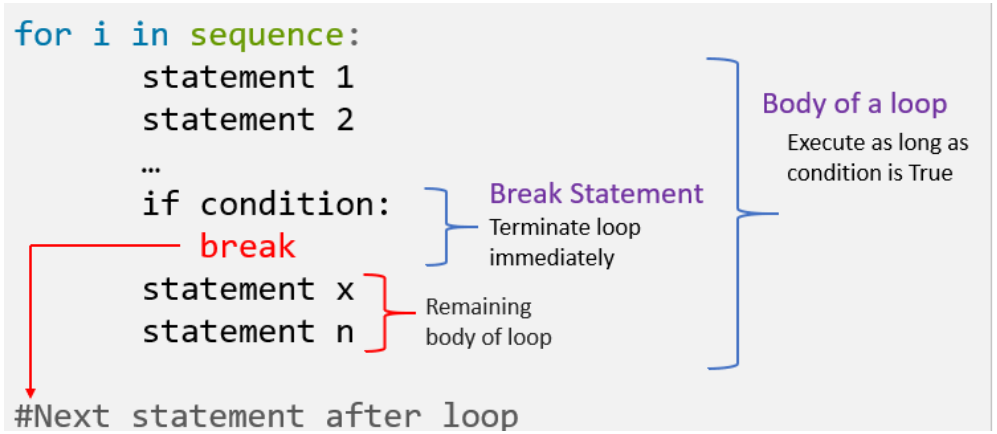


Transfer statements

- transfer statements are used to alter the program's way of execution in a certain manner.
- three types of transfer statements.
  - break statement
  - continue statement
  - pass statements

break statement

- The break statement terminates the loop containing it - it is used inside the loop to exit out of the loop.
- The loop is immediately terminated, when the break is encountered and the program control transfer to the next statement following the loop.
- if the break is used inside a nested loop, it will terminate the innermost loop.



Syntax of `break`:

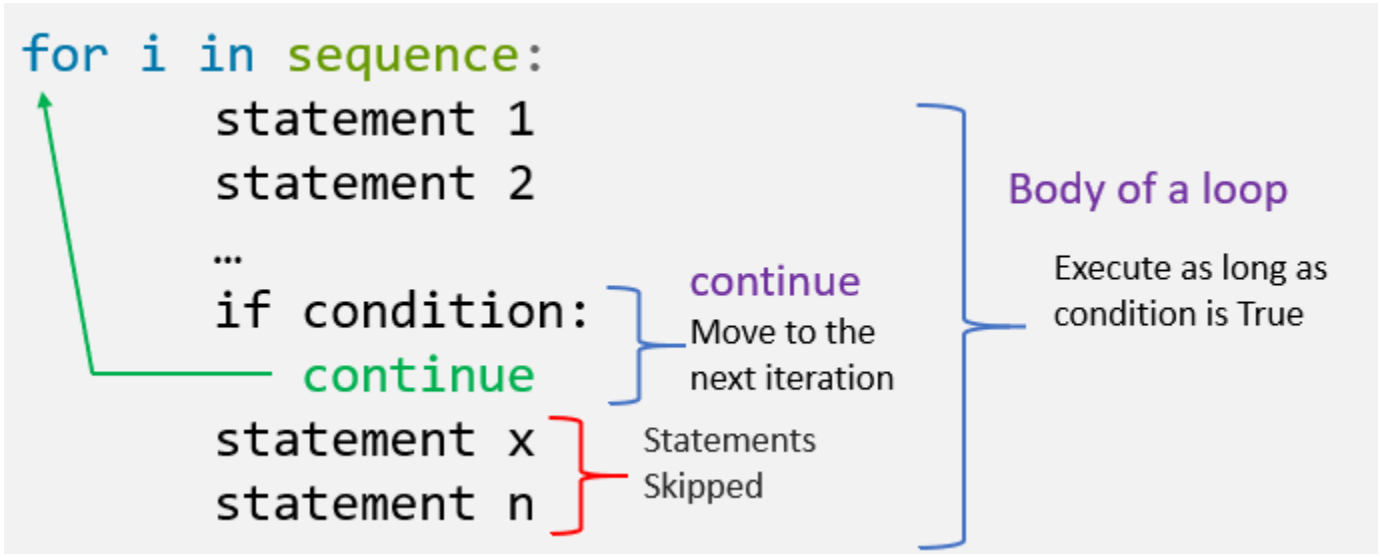
```
break
```

continue statement

- The continue statement skip the current iteration and move to the next iteration.
- The continue statement is used inside loops. Whenever the continue statement is encountered inside a loop, control directly jumps to the start of the loop for the next iteration, skipping the rest of the code present inside the loop's body for the current iteration.

Syntax of `continue`:

```
continue
```



```python
i = 1
while i<=5:
    print(i)
    if i == 3:
        break
    i = i + 1
print("Done")
```

```
1
2
3
Done
```

```python
i = 0
while i<5:
    i = i + 1
    if i == 3:
        continue
    print(i)
print("Done")
```

```
1
2
4
5
Done
```

pass statement

- The pass is the keyword in Python,
- it won't do anything.
- Sometimes there is a situation in programming where we need to define a syntactically empty block. We can define that block with the pass keyword.
- It is a Python null statement. Nothing happens when the pass statement is executed.
- It is useful in a situation where we are implementing new methods or also in exception handling. It plays a role like a placeholder.

Syntax of `pass` statement:

```
for element in sequence:
    if condition:
        pass
```