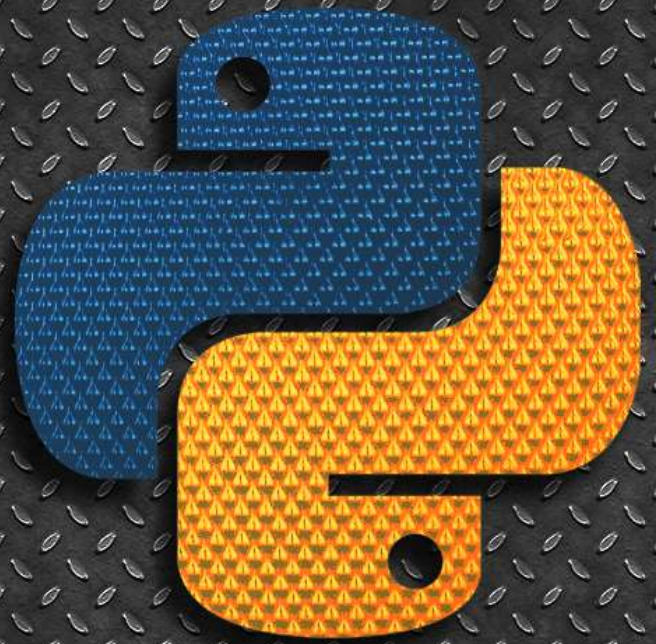
Python Fundamentals



Regular Expressions

Regular Expression

- A Regular Expression (RegEx) is a sequence of characters that defines a search pattern.

- Regular expressions aren't built-in to Python.

- So if we want to use them, you need to put an import re at the top of our code.

RegEx Functions

- The re module contains a set of functions that allows us to search within a string

	Function	Description
1.	findall()	Returns a list containing all matches
2.	search()	Returns a Match object if there is a match anywhere in the string
3.	split()	Returns a list where the string has been split at each match
4.	sub()	Replaces one or many matches with a string

Pattern Search using findall()

```
import re

txt = "bits of paper bits of paper"
x = re.findall("bi", txt)
print(x)
```

```
['bi', 'bi’]
```

```
# The list will have the matches in the order they are found.
# If no matches, an empty list is returned
```

Pattern Search using search()

```
import re

txt = "bits of paper bits of paper"
x = re.search("bi", txt)
print(x)

#.span() returns tuple containing the start-, and end positions of the match.
#.string returns the string passed into the function
#.group() returns the part of the string where there was a match

print(x.span())
print(x.string)
print(x.group())
```

```
<re.Match object; span=(0, 2), match='bi'>
(0, 2)
bits of paper bits of paper
bi
```

Pattern Search using split()

```
import re

txt = "bits of paper bits of paper"
x = re.split(" ", txt)
x = re.split(" ", txt, 1)
#specify split only at one occurrence

print(x)
```

```
['bits', 'of', 'paper', 'bits', 'of', 'paper’]
['bits', 'of paper bits of paper']
```

```
# returns a list where the string has been split at each match.
Here it will split at all spaces
```

Pattern Search using sub()

```
import re

txt = "bits of paper bits of paper"
x = re.sub(" ", "-", txt)
x = re.sub(" ", "-", txt, 1) #specify replace only at one
occurance
print(x)
```

```
bits-of-paper-bits-of-paper
bits-of paper bits of paper

# replaces the matches with the text we give
```

Pattern Search using Metacharacters

Examples:

```
# [] will search for A set of characters

Eg:
import re
txt = "Hello world"
#Find all lower case characters alphabetically between "a" and "m":
x = re.findall("[a-m]", txt)
print(x)
```

Output: ['e', 'l', 'l', 'l', 'd']

Pattern Search using Metacharacters

```
# \ denote a special sequence
txt = "I will give 100 rupees"
#Find all digit characters:
x = re.findall("\d", txt)
print(x)
Output: ['1', '0', '0']

# . Any character (except newline character)
txt = "hello world"
#Search for a sequence that starts with "he",
#followed by two (any) characters, and an "o":
x = re.findall("he..o", txt)
print(x)
Output: ['hello']
```

Pattern Search using Metacharacters

```
# ^ Starts with
txt = "hello world"
#Check if the string starts with 'hello':
x = re.findall("^hello", txt)
Print(x)
Output: ['hello’]

# $ Ends with
txt = "hello world"
#Check if the string ends with ‘world':
x = re.findall("world$", txt)
print(x)
Output: [‘world']
```

Pattern Search using Special Sequences

```
# \A if at the beginning of the string
txt = "hello world"

#Check if the string starts with 'hello':
x = re.findall("\Ahello", txt)
print(x)
Output: ['hello']
```

Pattern Search using Special Sequences

```
# \b at the beginning or at the end of a word
txt = "hello world“

#Check if the string ends with ‘world':
x = re.findall(r"\bworld", txt)
print(x)

# r’text here’ is a fantastic trick in Python that can help you avoid numerous conflicts such as back slash misinterpretations. ‘r’ is for ‘raw text’

#Check if the string starts with ‘hello':
x = re.findall(r"\bhello", txt)
print(x)
Output: [‘world’]
```

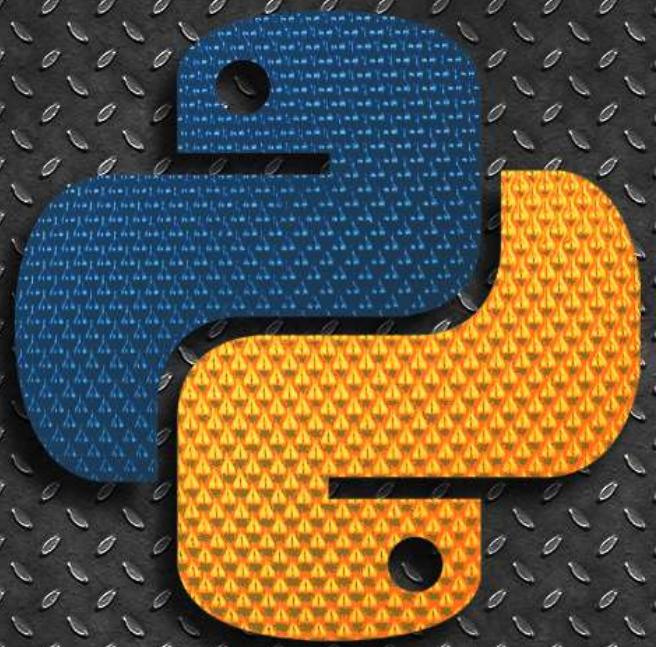
Pattern Search using Special Sequences

```
\S : Non-space characters except space
Extract Email from the text
txt = "hello test@gmail.com how are you?"

#Steps:
1- include everything that's non-space before the "@" sign
2- adding the "@" sign
3- everything non-space after the "@" sign.

regex = r'\S+@\S+'
x = re.findall(regex, txt)
print(x)
Output: ['test@gmail.com']
```

Python Fundamentals



LISTS

LISTS

- Collection of data which are normally related Instead of storing these as separate variables

- Example:

```
studentsAge = [18, 21, 23, 20, 21]
print(studentsAge)
print(studentsAge[0])
```

LISTS Access Options

```
print(studentsAge[-1])          #access from back

Print(studentsAge[2:4])         #slice list 2 to 4

Print(studentsAge[1:5:2])       #slice list 1 to 5
                                #every second number

Print(studentsAge[:4])          #slice till 4
```

LISTS - append() and del

```
studentsAge = []

studentsAge.append(20)
studentsAge.append("hi")
Print(studentsAge)


del studentsAge[1]
Print(studentsAge)
```

All list access formats can be used for del too.

LISTS - extend( ), in, insert( ), len( )

Extend() : Combine two lists

```
myList1 = ['h', 'e', 'l', 'l', 'o']
myList2 = [1, 2, 3, 4]
myList1.extend(myList2)
```

In: Check if an item is in a list

```
myList1 = ['h', 'e', 'l', 'l', 'o']
'e' in myList1
```

len: number of items in the list

```
len(myList1)
```

LISTS - reverse(), sort(), sorted()

reverse() Reverse the items in a list

```
myList1 = ['h', 'e', 'l', 'l', 'o']
myList1.reverse()
print (myList1)
```

sort() sort items alphabetically or numerically

```
myList1.sort() #remove item at 0 index
```

sorted() sort items alphabetically or numerically no change to original list

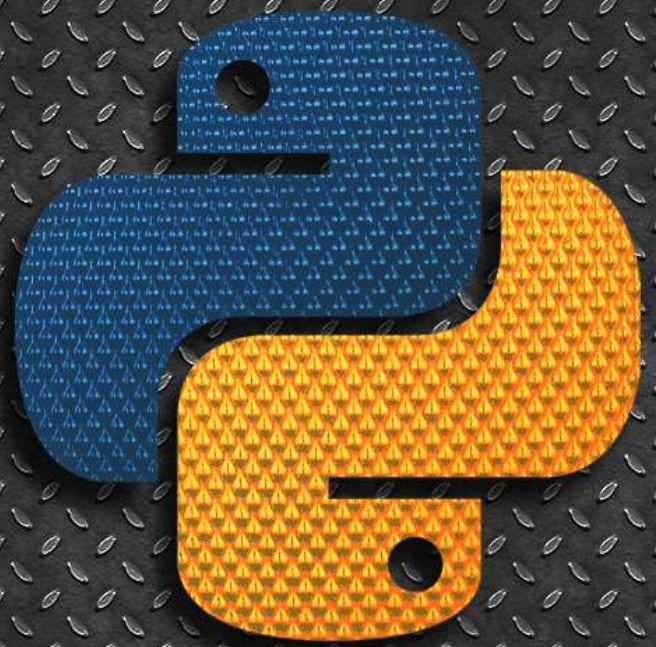
```
myList1 = sorted(myList1)
```

LISTS – Addition (+) and Multiplication (*)

```
# Concatenate a list
myList1 = ['h', 'e', 'l', 'l', 'o']
print (myList1+['w','o','r','l','d'])


# Duplicate a list and concatenate it to the end of the list
myList1 = ['h', 'e', 'l', 'l', 'o']
print (myList1*3)
```

Python Fundamentals



TUPLES

Tuples

- Tuples are like lists, but unlike lists, we cannot modify their initial values.
- Eg: to store the names of the months of the year.
- we use round brackets ( ) when declaring a tuple.

```
months = ("Jan", "Feb", "Mar", "Apr", "May",
"Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")

print(months[0])
print(months[-1])

months[0] = "test"
TypeError: 'tuple' object does not support item assignment
```

Tuples – del(), in , and len()

```
myTuple = ('hello', 'world', 2022)
```

- Len() – number of items in tuple

```
print (len(myTuple))
```

- In – Checks if an item is in the tuple

```
print('world' in myTuple)
```

- Del() – Delete the entire tuple

```
del myTuple
print(myTuple)
```

Tuples – del(), in , and len()

```
myTuple = ('hello', 'world', 2022)
```

- Len() – number of items in tuple

```
print (len(myTuple))
```

- In – Checks if an item is in the tuple

```
print('world' in myTuple)
```

- Del() – Delete the entire tuple

```
del myTuple
print(myTuple)
```

Tuples – addition + and multiplication *

```
myTuple = ('hello', 'world', 2022)
```

- Addition Operator: + Concatenate Tuples

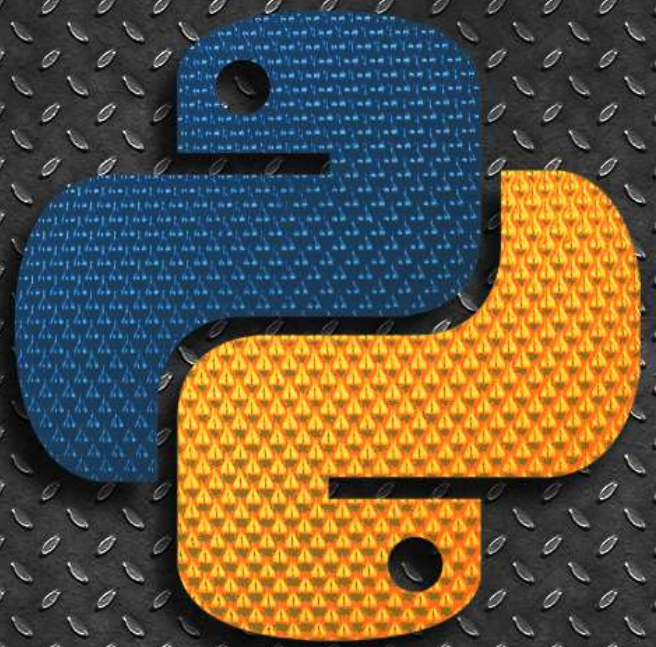
(+ and * will not alter the original tuple)

```
print (myTuple + ('how', 'are', 'you'))
```

- Multiplication Operator: * Duplicate a tuple and concatenate it

```
print(myTuple*3)
```

Python Fundamentals



DICTIONARY

Dictionary

- Dictionary is a collection of related data PAIRS.
- dictionaryName = {dictionarykey : data}
- (dictionary keys must be unique in one dictionary)
- For example, if we want to store the name and age of students

```
myStudents = {"Abhi":30, "Sibi":28, "Subi":"not updated"}
```

- Or you can also declare using dict() method
- use round brackets ( ) instead of curly brackets { } no quotes for dictionary keys.

```
myStudents = dict(Abhi:30, Sibi:28, Subi:"not updated"}
```

```
print(myStudents["Abhi"])
myStudents["Subi"] = 25
myStudents["Bibi"] = 22
print(myStudents)
```

Dictionary – get(), items(), keys(), values()

```
myStudents = {"Abhi":30, "Sibi":28, "Subi":"not updated"}
```

- Get() : Returns value of given key

```
print(myStudents.get("Abhi")
```

- Items() : Returns a list of dictionary's pairs as tuples

```
"Abhi" in dic1
Output: dict_items([('Abhi', 30), ('Sibi', 28), ('Subi', 'not updated')])
```

- keys() : Returns a list of dictionary's keys

```
myStudents.keys()
Output: dict_keys(['Abhi', 'Sibi', 'Subi'])
```

- values() : Returns a list of dictionary's values

```
myStudents.values()
Output: dict_values([30, 28, 'not updated'])
```

Dictionary – in, len(), update()

```
myStudents = {"Abhi":30, "Sibi":28, "Subi":"not updated"}
```

- In : Check if an item is in a dictionary

```
"Abhi" in myStudents
28 in myStudents.values()
```

- len( ) : Find the number of items in a dictionary

```
Print(len(myStudents))
```

- Update(): Adds one dictionary's key-values pairs to another. Duplicates are removed.

```
day1 = {1: 'monday', 2: 'tuesday'}
day2 = {1: 'wednesday', 3: 'thursday'}
day1.update(day2)

print (day1)
{1: 'wednesday', 2: 'tuesday', 3: 'thursday'}
```

Dictionary – clear(), del

```
myStudents = {"Abhi":30, "Sibi":28, "Subi":"not updated"}
```

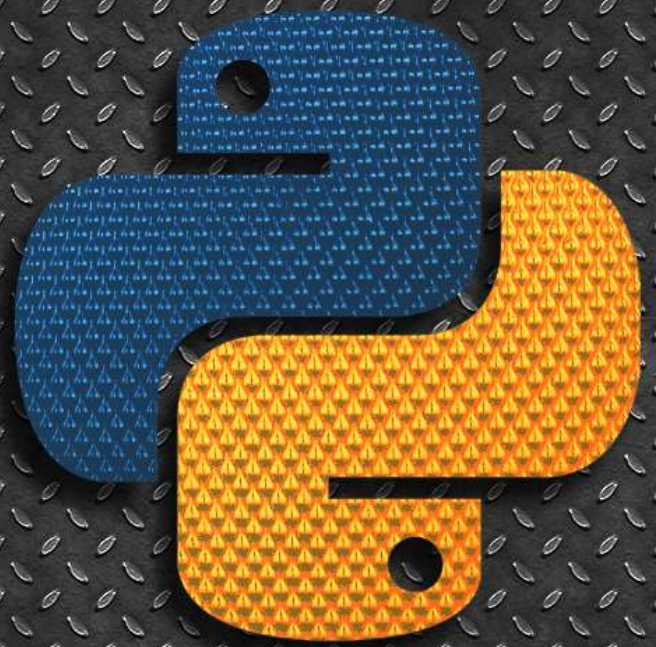
- Clear() : Deletes all items in dictionary

```
myStudents.clear()
print(myStudents)
```

- del: Deletes the entire dictionary

```
del myStudents
print(myStudents)
```

Python Fundamentals



Expressions

Expressions

- An expression is a combination of operators and operands that is interpreted to produce some other value

- 1. Constant Expressions: These are the expressions that have constant values only.

```
x = 15 + 1.3
print(x)
```

- 2. Arithmetic Expressions: An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis.

```
x = 40
y = 12
add = x + y
sub = x - y
pro = x * y
div = x / y
```

Expressions

- 3. Integral Expressions: produce only integer results after all computations

```
a = 13
b = 12.0
c = a + int(b)
print(c)
```

- 4. Floating Expressions: produce floating point numbers as result

```
a = 13
b = 5
c = a / b
print(c)
```

Expressions

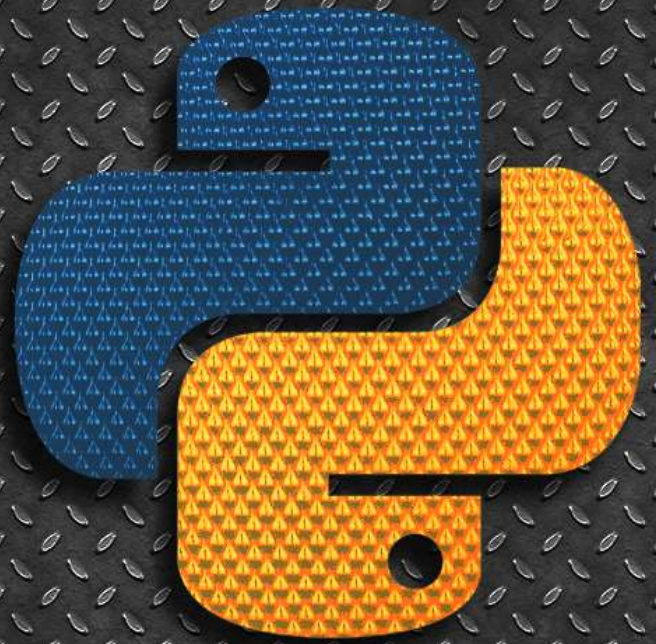
- 5. Relational Expressions: arithmetic expressions are written on both sides of relational operator (> , < , >= , <=).

```
a = 21
b = 13
c = 40
d = 37
p = (a + b) >= (c - d)
print(p)
```

- 6. Logical Expressions: These are kinds of expressions that result in either True or False.

```
P = (10 == 9)
Q = (7 > 5)
# Logical Expressions
R = P and Q
S = P or Q
T = not P
print(R)
print(S)
print(T)
```

Python Fundamentals



INPUT and OUTPUT

Input() and print()

- Input() statement prompts the user for input
- It stores the information as a string

```
studName = input("Enter student name: ")
studAge = input("Enter student  age: ")
```

print() statement for output – three ways to print variables

```
print("The student name is", myName, "and is",myAge, "years old.") OR

print("The student name is %s and is %s years old." %(studName, studAge)) OR

print("The student name is {} and is {} years old.".format(studName, studAge))
```

Triple Quotes and Escape Chars

- To display a long message in multi lines, we can use the triple-quote

```
print (’’’Hello World.
How are you doing.’’’)
```

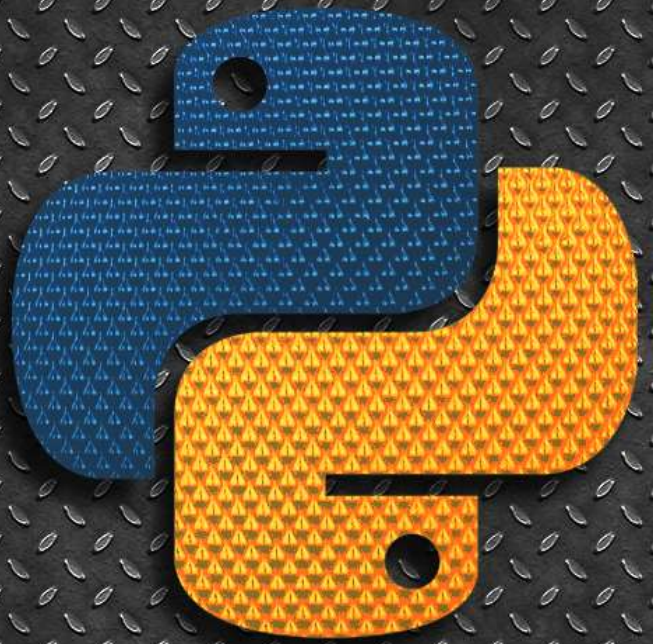
Escape chars to print “unprintable” characters

```
print (‘Hello\nWorld’)
```
\n (Prints a newline)

```
print (‘\\’)
```
(Prints a backslash)

```
print (“My hight 5’5\" tall”)
print (‘My hight 5\’5" tall’)
```

Python Fundamentals

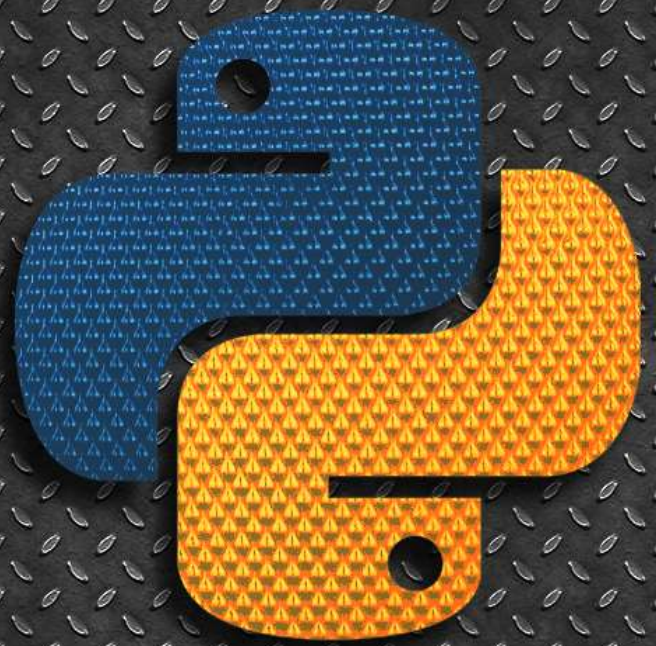


COMPARISION OPERATORS

Comparison Operators

5 == 2	Equality
5 != 2	Non-Equality
5>2	Greater than
2<5	Smaller than
5>=2	Greater than or equal to
2 <= 5	Less than or equal to

Python Fundamentals



LOGICAL OPERATORS

LOGICAL Operators – and, or , not

5 == 5 and 2 < 1	will return False
5 == 5 or 2 < 1	will return True
not 2 < 1	will return True
not 2 > 1	will return False

Triple Quotes and Escape Chars

- To display a long message in multi lines, we can use the triple-quote

```
print (’’’Hello World.
How are you doing.’’’)
```

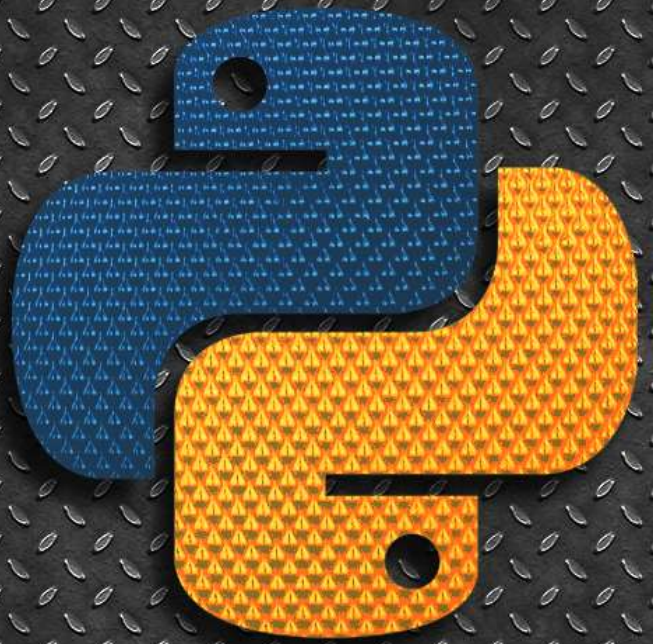
Escape chars to print “unprintable” characters

```
print (‘Hello\nWorld’)
```
\n (Prints a newline)

```
print (‘\\’)
```
(Prints a backslash)

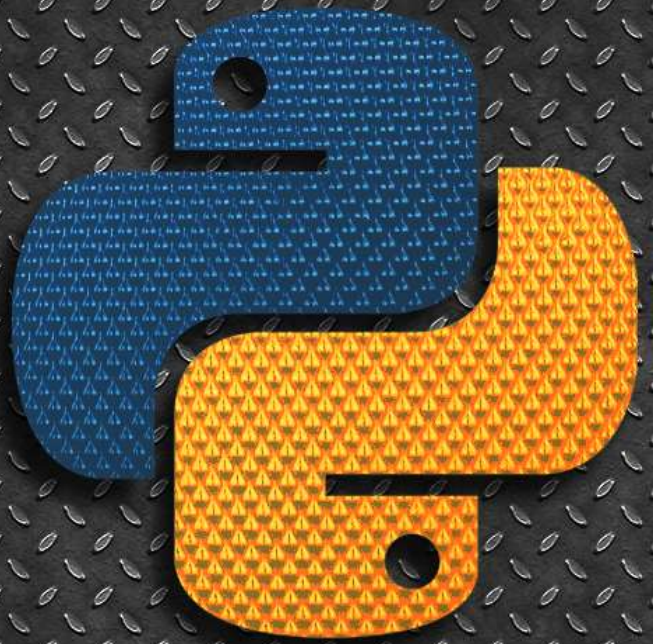
```
print (“My hight 5’5\" tall”)
print (‘My hight 5\’5" tall’)
```

Python Fundamentals



CONTROL FLOW STATEMENTS

Python Fundamentals



CONDITIONAL STATEMENTS

If Statement

The structure of an if statement is as follows:

if condition 1 is met: do A

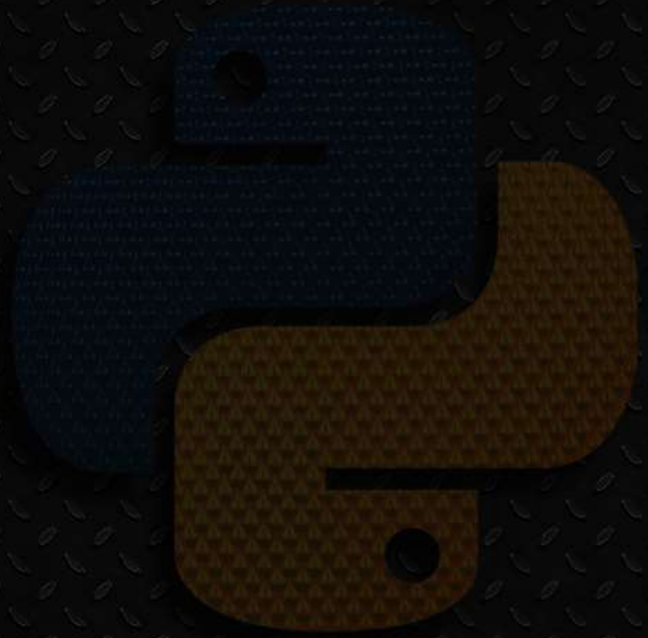
elif condition 2 is met: do B

elif condition 3 is met: do C

elif condition 4 is met: do D

else:

do E



If Statement

Python uses indentation to separate blocks of code

Other programming languages uses braces { }

```
userInput = input('Enter 1 or 2: ')
if userInput == "1":
    print (“You entered 1")
    print (“Are you number one?")
elif userInput == "2":
    print (“Runner up")
    print (“Good that you finished")
else:
    print ("You did not enter a valid number")
```

Inline If Statement

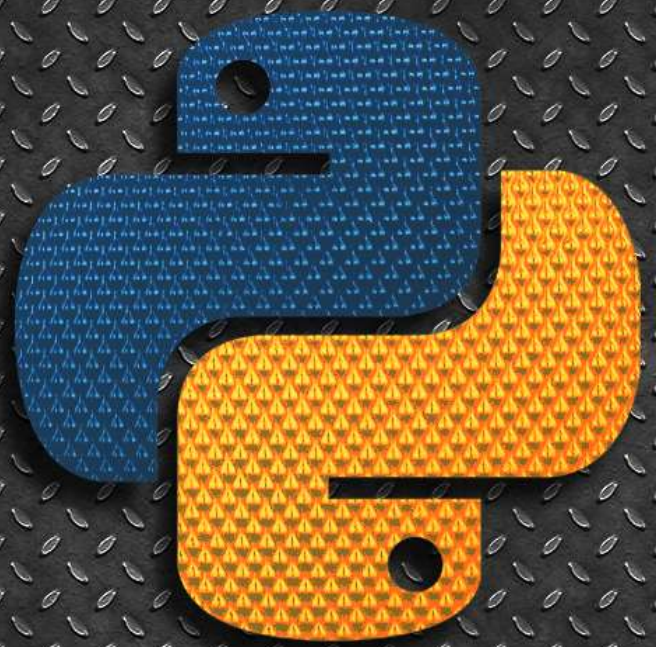
A simpler form of an if statement to perform a simple task.

do Task A if condition is true else do Task B

```
B=12
A = 12 if B==10 else 13
print(A)

print ("B is ten" if B == 10 else "B is not 10")
```

Python Fundamentals



LOOPS

FOR Loop – Looping through list

for loop executes a block of code repeatedly until the condition in the for statement is no longer valid.

For a in iterable: (iterable refers to anything that can be looped over, such as a string, list or tuple.)

print (a)

```
fruits = ['apples', 'oranges', 'banana', 'cherry’]
for fruit in fruits:
    print (fruit)

#display the index of the members in the list using enumerate.
for index,fruit in enumerate(fruits):
    print (index,fruit)
```

FOR Loop – Looping through numbers

range() function generates a list of numbers and has the syntax range (start, end, step).

range(5) will generate the list [0, 1, 2, 3, 4]
range(3, 10) will generate [3, 4, 5, 6, 7, 8, 9]
range(4, 10, 2) will generate [4, 6, 8]

```
for i in range(5):
    print (i)
```

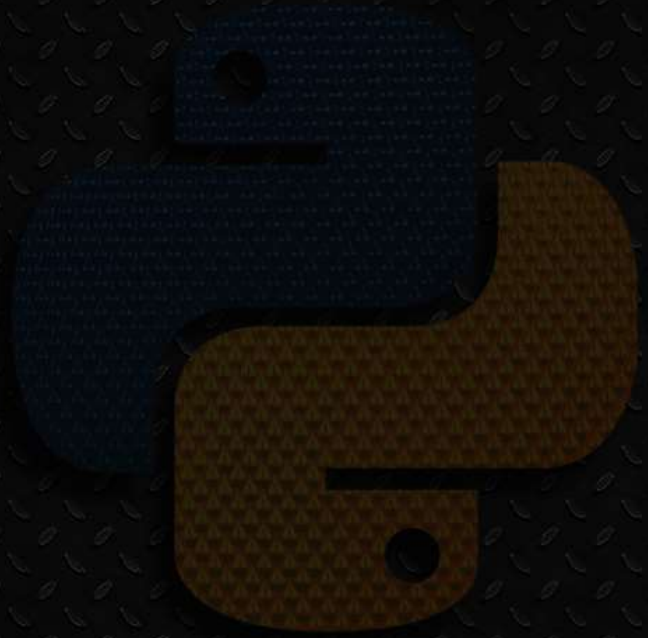
```
0
1
2
3
4
```

While Loop

while loop repeatedly executes instructions inside the loop while a certain condition remains valid.

while condition is true:

do A



```
counter = 5
while counter > 0:
    print ("Counter = ", counter)
    counter = counter - 1
```