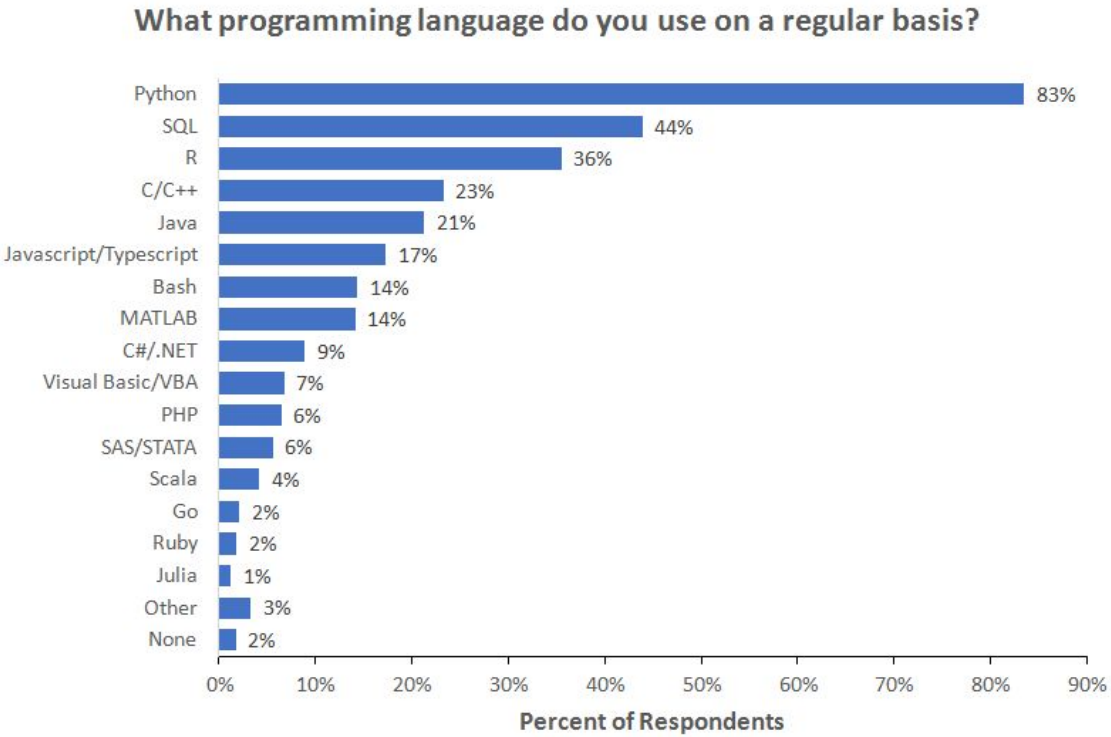

PASCAL
node
JS
php
Microsoft®
.NET
JS
RAILS
TS
HTML
5
Java
Lua
free pascal

WHICH ONE TO CHOOSE AND WHY ?



Note: Data are from the 2018 Kaggle Machine Learning and Data Science Survey. You can learn more about the study here: http://www.kaggle.com/kaggle/kaggle-survey-2018. A total of 18827 respondents answered the question.

BUSINESS BROADWAY
DATA SCIENCE | CUSTOMER ANALYTICS | MACHINE LEARNING

What is Python ?

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

What is Python ?

Python is one of those rare languages which can claim to be both simple and powerful. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

FEATURES OF PYTHON

- **Simple** - Reading a good Python program feels almost like reading English, although very strict English!
- **Easy to learn** - As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.
- **Free and Open Source** - In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs.
- **High Level Language** - When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.
- **Portable** - All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

  You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

FEATURES OF PYTHON

- **Interpreted** - Python, on the other hand, does not need compilation to binary. You just run the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it.
- **Object oriented** -Python supports procedure-oriented programming as well as object-oriented programming. In procedure-oriented languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In object-oriented languages, the program is built around objects which combine data and functionality.
- **Extensive Libraries** - The Python Standard Library is huge indeed. It can help you do various things involving regular expressions,documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed.

What is Python ?

- Free and open source.
- Interpreted, object oriented, high level programming language.
- We can do any type of development that we want.
- It is designed to be human readable.
- Hundreds of python libraries and frameworks.
- Huge community.

What can we do with Python ?

GAMES	Data Analysis	AI/ML
3D GRAPHICS	Web Apps	Mobile Apps
GUIs	Hacking	Desktop Apps
ROBOTICS	Testing	Automation

16 Famous Companies that uses PYTHON

DIFFERENT PYTHON IDE


IDLE


SPYDER


PC
PyCharm


eric


Vim


Atom


jupyter


Anaconda


PyDev


Thonny
Python IDE for beginners

Python Installation and Setting Up Environment

BASIC PYTHON SYNTAX

INPUT / OUTPUT

To display any output we use **print()** function.

```
print('Welcome Everyone')
```

```
Welcome Everyone
```

To take input from the user we use **input()** function.

```
name  = input('Enter your Name :')
```

```
Enter your Name :
```

All about **print()**

The print() function prints the specified message to the screen, or other standard output specified.

```
>>> print()

>>>
```

Just calling the print function without any message or without passing any argument will result in a empty line. We can call print multiple times.

```
>>> print('hello pytonista')
hello pytonista
>>>
```

We can pass message a string directly in print and it will display that string in the python shell. Not only string we can pass any data type in python and it will print that to the shell.

```
>>> x = 'hello'
>>> print(x)
hello
```

We can also pass variables in print and it will print the data stored in that variable. Not only one variable we can pass n number of variables separated by commas.

All about **print()**

The print() function prints the specified message to the screen, or other standard output specified.

```
>>> print('hello','I am learning python')
hello I am learning python
```

We can pass multiple arguments all at once and it will print it in the same line.

```
print('hello')
print('I am learning python')
```

```
hello
I am learning python
```

Multiple print will result in multiple lines. First print will be printed in the first line and second print will be printed in the second line.

We can print in the same line by specifying the end argument in print. By default it is '\n' that means new line. We can change that to empty space just by giving end = ''

```
print('hello',end=' ')
print('I am learning python')
```

```
hello I am learning python
```

All about **print()**

We can also print the values passed in one print statement into multiple lines and format it according to our wish.

```
print('hello \nworld')
```

```
hello
world
```

‘\n’ is used to break the line and print everything after \n in new line

```
print('hello \n\tworld')
```

```
hello
        world
```

Here we have specified ‘\n\t’. ‘\n’ will break the line and ‘\t’ is used for tab or four spaces.

Comments in Python

Comments are non-executable python code.

We use comments to -

- Explain python code.
- Make the code more readable.
- Prevent Execution when testing the code

In python we use # and “‘ ”’ to put comment in our code.

```
#this is comment
print('Hello world')
```

```
Hello world
```

is used for single line comment.

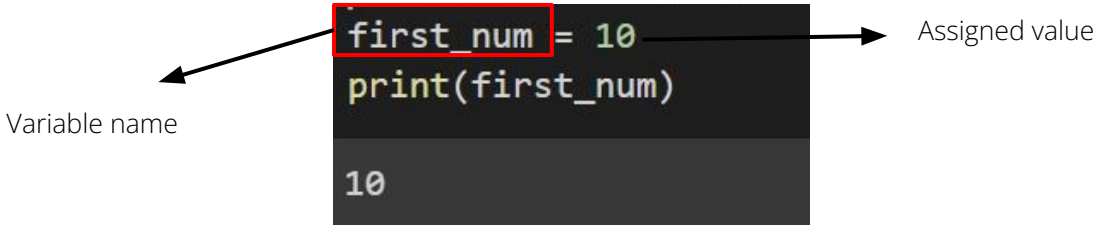
```
''' This is line 1
This is line 2'''
print('Hi Pythoninsta')
```

```
Hi Pythoninsta
```

“‘ ”’ is used for multi line comment.

VARIABLES

- Variables are named memory locations that store data.
- Python is dynamically typed which means you don't have to declare the type of variables during initialization.



To check the memory location we use
id(variable name)

```
print(id(first_num)
)
```

```
10914784
```

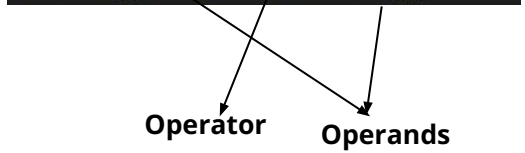
RULES FOR NAMING VARIABLES

- Variable name should start with a letter or underscore.
- We cannot use python keywords as variable names.
- Do not use numbers at the start of variable names.
- The only allowed special character is underscore (_) .
- Variable names can only have alphanumeric characters (A-Z,a-z,0-9) and underscore.
- Variable names are case sensitive (e.g Age, age and AGE are different variable names)

OPERATORS

Operators are used to perform operations on operands (variables and values) .

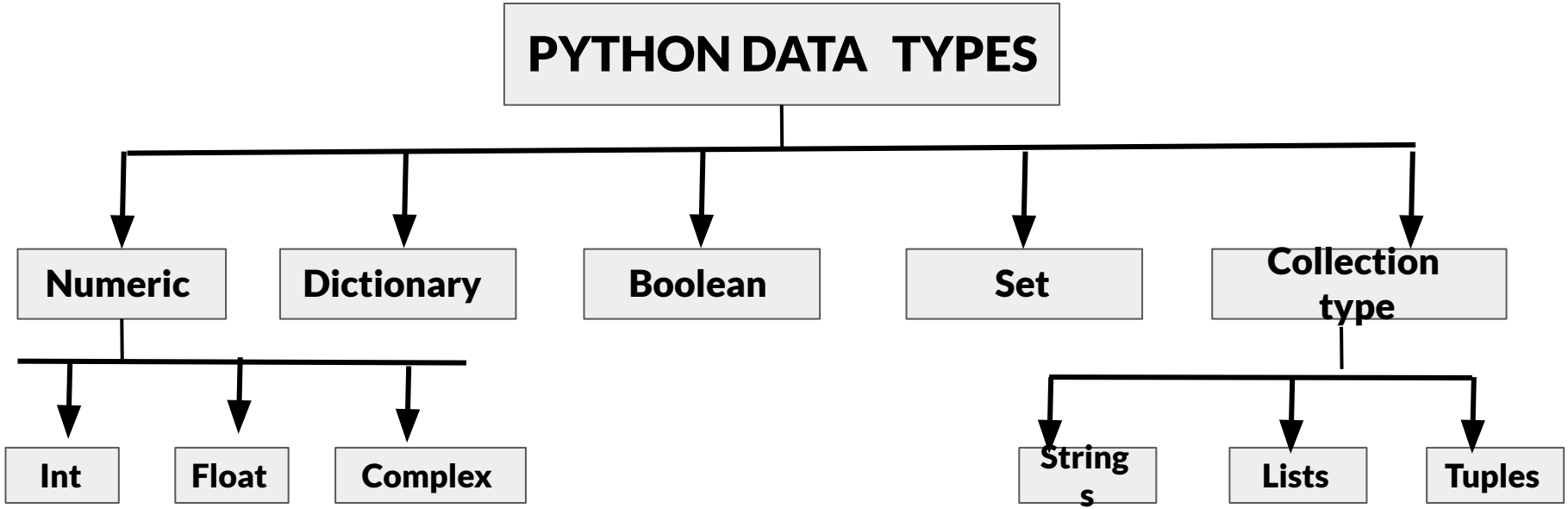
```
num_1 = 9
num_2 = 20
print(num_1 + num_2)
```



Types of operators in python

```
#Arithmetic operators(+,-,*,/,%,**,//)
#Assignment operators(=,=+,-=,*=,/=,%=,//=,**=)
#Comparison operators(==,!=,>,<,>=,<=)
#Logical operators(And,Or,Not)
#Identity operators(is,is not)
#Membership operators(in,not in)
```

Data types


PYTHON DATA TYPES
Numeric
Dictionary
Boolean
Set
Collection type
Int
Float
Complex
Strings
Lists
Tuples

Numeric Types:

Integer type -

Whole number without decimal points (10,1,-1,240, etc.)

```
first_num = 10
print(type(first_num))
```

```
<class 'int'>
```

Float type -

Floats are the numbers with decimal values.

```
second_num = 10.05
print(type(second_num))
```

```
<class 'float'>
```

Complex -

Complex types has a real and imaginary part.

```
comp = 3 +4j
print(type(comp))
```

```
<class 'complex'>
```

Numeric Types:

Hex function -

hex() function is used to convert a decimal number into a hexadecimal form

```
>>> hex(10)
'0xa'
```

Bin Function -

bin() is used to convert a decimal number into a binary number.

```
>>> bin(10)
'0b1010'
```

Here ob is used to represent binary number

Oct function

oct() is used to convert a decimal number into a octal form.

```
>>> oct(10)
'0o12'
```

Here Oo is used to represent octal numbers.

String:

String literals in python are surrounded by either single quote,double quotes or triple quotes.

```
course = "Data Science"
print(course)
print(type(course))
```

```
Data Science
<class 'str'>
```

```
course = 'Data Science'
print(course)
print(type(course))
```

```
Data Science
<class 'str'>
```

```
course = '''Data Science'''
print(course)
print(type(course))
```

```
Data Science
<class 'str'>
```

Boolean:

Booleans are **True** and **False** types.

```
t_data = True
f_data = False
print(type(t_data))
print(type(f_data))
```

```
<class 'bool'>
<class 'bool'>
```

Data Type Conversion:

int
float
string
boolean

Conversion from int to :

float
string
boolean

```
first_num = 10  #integer type
print(first_num)

print(float(first_num)) #convert to float

print(type(first_num))  #check the type
```

```
10
10.0
<class 'int'>
```

```
[8]  #converting to string
     print(str(first_num))

     print(type(str(first_num)))
```

```
     10
     <class 'str'>
```

```
#converting to boolean

#If there's any value stored in variable apart from zero then it will return True otherwise 0

print(bool(first_num))  #converting to boolean

print(type(bool(first_num))) #check type
```

Conversion from float to :

int
string
boolean

```
a = 10.10 #foat

print(type(a)) #type of a

print(int(a))  #convert to int

print(type(int(a))) #check type after conversion
```

```
<class 'float'>
10
<class 'int'>
```

```
a = 10.10 #float

print(type(a)) #type of a

print(str(a))  #convert to string

print(type(str(a))) #check type after conversion
```

```
<class 'float'>
10.1
<class 'str'>
```

```
a = 10.10 #float

print(type(a)) #type of a

print(bool(a))  #convert to boolean

print(type(bool(a))) #check type after conversion
```

```
<class 'float'>
True
<class 'bool'>
```

Conversion from string to :

int
boolean
float

```
a = '10'  #string

print(type(a)) #check type of a

print(int(a))  #convert to int

print(type(int(a)))  #check type after conversion
```

```
10
<class 'int'>
```

```
a = '10.10'  #string

print(type(a)) #check type of a

print(float(a))  #convert to float

print(type(float(a)))  #check type after conversion
```

```
<class 'str'>
10.1
<class 'float'>
```

```
a = 'codegnan'  #string

print(type(a)) #check type of a

print(bool(a))  #convert to boolean

print(type(bool(a)))  #check type after conversion
```

```
<class 'str'>
True
<class 'bool'>
```

Conversion from bool to :

int
string
float

```
a = True #bool value

print(type(a)) #check type of a

print(int(a))  #convert to int

print(type(int(a)))  #check type after conversion
```

```
<class 'bool'>
1
<class 'int'>
```

```
a = True #bool value

print(type(a)) #check type of a

print(str(a))  #convert to str

print(type(str(a)))  #check type after conversion
```

```
<class 'bool'>
True
<class 'str'>
```

```
a = True #bool value

print(type(a)) #check type of a

print(float(a))  #convert to float

print(type(float(a)))  #check type after conversion
```

```
<class 'bool'>
1.0
<class 'float'>
```

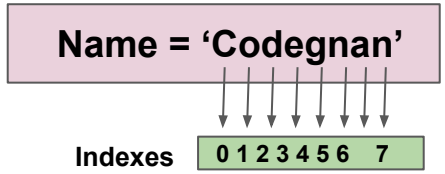
String operations

Strings are the ordered sequence of the character that means we can access the individual characters by using the index at which the character is present.

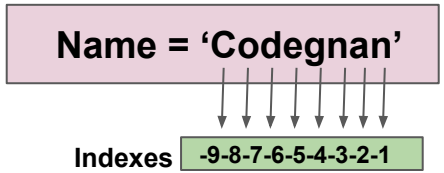
String Indexing :

- Strings are immutable data types.
- Strings are structured data types so they can be indexed and sliced.
- We can do positive and negative indexing. Positive indexing starts with 0 from left to the right.
- Negative indexing starts from -1 from

Indexing starts from 0



Negative Indexing




```
name = 'Codegnan'
print(name[0])

print(name[-1])
```

```
C
n
```

Positive Indexing

```
name = 'Codegnan'
print(name[0])
print(name[1])
print(name[2])
```

```
C
o
d
```

Negative Indexing

```
name = 'Codegnan'
print(name[-1])
print(name[-2])
print(name[-3])
```

```
n
a
n
```

```
name = 'Codegnan'
print(name[9])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-6b90ae1d00b0> in <module>()
      1 name = 'Codegnan'
----> 2 print(name[9])

IndexError: string index out of range
```

Attempting to index beyond the end of string results in error.

```
name = 'Codegnan'
print(name[-9])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-4-eb69a8556c37> in <module>()
      1 name = 'Codegnan'
----> 2 print(name[-9])

IndexError: string index out of range
```

Attempting to index beyond the start of string using negative indexing also results in error.

String Slicing :

- By slicing a string we are creating a substring
- When slicing you have to specify the start index and end index.

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
--> returns the string between
start index and end index
a[0 : 10]
```

```
'Codegnan c'
```

The string a is sliced starting from index 0 to index 10. When slicing the end index in exclusive. It is not included and whitespaces are also indexed so we got the output ‘codegnan c’

String Slicing :

- We can also omit the start and end index.
- Omitting start index means the string starts from zero till the end index specified.
- Omitting the end index means the string will start from the start index specified till the end.

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[ : 10] )
```

```
Codegnan c
```

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[2 : ] )
```

```
degnan code stars
```

If you choose to omit the start and end index the result will be the whole string itself.

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[ : ] )
```

```
Codegnan code stars
```

String Slicing :

- We can also do slicing using the negative indexing.
- Slicing string using mix of positive and negative index is also possible.

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[-5 : -1] )
```

```
star
```

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[5 : -1] )
```

```
nan code star
```

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[-5 : 18] )
```

```
star
```

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[-5 : 25] )
```

```
stars
```

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end]
print(a[-55 : 5] )
```

```
Codeg
```

In slicing operations if the end or the start index is not in the range it does not throws any error. It will just slice the element till the last index if the end index is out of range or it will slice the element from 0 index if the start index is out of range.

String Slicing And Striding :

- In addition to slicing we can also include striding using another :
- Striding or step size is specified to skip the character while slicing.
- By default the stride is 1.

```
#slicing
a = 'Codegnan code stars'
#slicing --- a[start : end : stride]
print(a[0 : 10 : 2] )
```

```
Cdga
```

Here we have given stride of 2. That means it will slice from start index to end index specified and give back every second element from that slice.

String Slicing And Striding :

- We can also use negative values for striding. But when giving negative stride value we have to give the start index bigger than the end index.

```
#slicing
a = 'Codegnan'
#slicing --- a[start : end : stride]
print(a[5 : 0 : -1] )
```

```
ngedo
```

Here we have given start index larger than the end index. It simply means that start from 5 go backwards with step of 1 till 0.

```
#slicing
a = 'Codegnan'
#slicing --- a[start : end : stride]
print(a[5 : 0 : -2] )
```

```
neo
```

Here it gives us every second element starting from 5 going backwards with step of 2 till 0.

```
#slicing
a = 'Codegnan'
#slicing --- a[start : end : stride]
print(a[ :  : -1] )
```

```
nangedoC
```

Not specifying the start and the end index will result in the string itself in entirety and giving step size as -1 will reverse all the element of the string.

```
name = 'codegnan'
name[0] = 'd'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-aa213d330373> in <module>()
      1 name = 'codegnan'
----> 2 name[0] = 'd'

TypeError: 'str' object does not support item assignment
```

Modifying a string element like above will result in error as strings are immutable data types.

Old Style String Formatting

```
lang = 'python'
print('hello %s'%lang)
```

```
hello python
```

```
num = 10
print('hello %d'%num)
```

```
hello 10
```

```
n = 10.2
print('hello %f'%n)
```

```
hello 10.200000
```

```
n = 10.2
print('hello %.2f'%n)
```

```
hello 10.20
```

Strings in python have a unique built in operation that can be accessed by using % operator to do positional string formatting.

New String Formatting :

```
#string formatting
first_name = 'John'
last_name = 'Harris'
print('Hello',first_name,last_name)
```

```
#string concatenation
print(first_name + ' ' + last_name)
```

```
#place holders and format() method
print('Hello, {} {}'.format(first_name,last_name))
```

```
#using 'f' string
print(f'Hello, {first_name} {last_name}')
```

String Formatting :

str.format() method

```
a = 10
b = 20
print('a is',a,'b is',b)
```

```
a is 10 b is 20
```

Old way of string formatting.

```
a = 10
b = 20
print('a is {} b is {}'.format(a,b))
```

```
a is 10 b is 20
```

We can use .format method to specify the values in place holders. Represented by {} empty curly braces. Here the first argument is passed in the first place holder and second argument is passed in second place holder.

String Formatting :

str.format() method

```
name = 'codegnan'
course = 'python'
print('Name of insitute is {name} and the course is {course}'.format(name=name,course=course))
```

```
Name of insitute is codegnan and the course is python
```

We can also give names to these place holders. Here we have given the name ‘name’ to first place holder and ‘course’ to the second place holder. While passing the arguments we have to pass keyword arguments to fill these place holders.

```
name = 'codegnan'
course = 'python'
print('Name of insitute is {name} and the course is {course}'.format(course=course,name=name))
```

```
Name of insitute is codegnan and the course is python
```

String Formatting :

str.format() method

```
a = 10
b = 20
c = 30
print('Values {0} {0} {1}'.format(a,b,c))
```

```
Values 10 10 20
```

Here we have given the arguments the place holders are going to store the values. We have given 0 in first place holder and also 0 in second place holder and 1 in third place holder that means in first and second place holder the values will be the one passed and 0th argument.

```
a = 10
b = 20
c = 30
print('Values {0:.2f} {0:3d} {1}'.format(a,b,c))
```

```
Values 10.00  10 20
```

String Formatting :

str.format() method

We can directly pass the values in the place holders itself without using any explicit format method.

```
a = 10
b = 20
c = 30
print(f'Values {a:.2f} {b:3d} {c}')
```

```
Values 10.00  20 30
```

```
a = 10
b = 20
c = 30
print(f'Values {a} {b} {c}')
```

```
Values 10 20 30
```

String Functions:

```
a = 'Codegnan'
b = 'Python'
print(a + b)
```

```
CodegnanPython
```

Using ‘+’ operator on string will result in string concatenation.

```
a = 'Codegnan'
print(a * 3)
```

```
CodegnanCodegnanCodegnan
```

Using ‘*’ operator will result in multiplying the string that many number of times.

```
ord('a')
```

```
97
```

ord() gives the numeric equivalent of a character string.

```
chr(97)
```

```
'a'
```

chr() gives the character equivalent of a numeric value.

String Methods :

```
#to check all the methods in str
help(dir(str))
```

```
#count total occurrence of a letter
a = ''' twinkle twinkle little star'''
a.count('t')
```

```
#replace a string with another string
a.replace('star','moon')
```

```
#find method() gives the index value of
first occurence of the string
a.find('twinkle')
```

String Methods :

```
#lower all the letters
b = "Hello Codegnan stars"
b.lower()
```

```
b = "Hello Codegnan stars"
print(b.lower())          #changes all the letters to lowercase
print(b.upper())          #changes all the letters to uppercase
print(b.capitalize())     #changes the first letter into caps, other in lower
print(b.title())          #changes first letter of each word into caps
```

```
c = 'codegnan'
print(c.isupper())   #check whether the string is uppercase or not
print(c.islower())   #check whether the string is lowercase or not
print(c.isdigit())   #check whether the string is a digit or not
print(c.isalpha())   #check whether the string is alphanumeric(A-Z,a-z) or not
print(c.startswith('lo'))   #check whether the string starts with "lo"
print(c.endswith('l'))      #checks whether the string ends with "l"
```

→ **Returns Boolean True or False**

String Methods

Some more methods -

- **index()** - gives the index of the string.
- **rindex()** - looks from right to left.
- **isalnum()** - checks whether the string is alphanumeric or not.
- **isspace()** - checks if the string is having only whitespace.
- **isascii()** - checks if the string has ascii representation or not.
- **rfind()** - looks for the specified string from right side.
- **rjust()** - right align the string
- **ljust()** - left align the string.
- **swapcase()** - swaps the case of the string.
- **isdecimal()** - checks if the string has decimal values.
- **casefold() -** converts all the alphabets in lower.

String Methods :

split
join
strip

```
languages = "java,python,c,c++"
d = languages.split(',')        #splits the string
after each occurrence of ',' into different
substrings and return a list of strings
print(d)
```

```
a = " ".join(d)    #join method takes all the strings in
iterable and joins into single string
print(a)
```

```
name = "     John    "
print(name.strip())    #removes the whitespaces from string
print(name.lstrip())   #removes whitespace from the left
print(name.rstrip())   #removes the whitespace from the
right
```

LISTS

Lists

- Lists are ordered collection of elements enclosed within a bracket and separated by commas.
- Lists are mutable data types that means we can add, update and delete the element of the lists.
- Lists can be indexed and sliced.
- Lists are also a type of iterables in python.
- We can store multiple kind of data types in lists.

Syntax to create list

```
lst = [element1, element2,.. ,elementn]
```

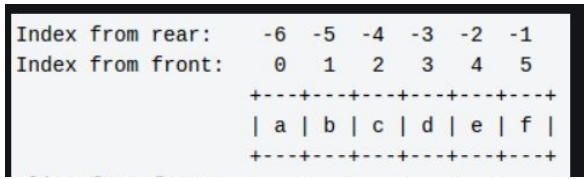
```
lst = ['John',23,True,89.5]
print(lst)
print(type(lst))
```

```
['John', 23, True, 89.5]
<class 'list'>
```

Lists:

Indexing
Slicing
length of list

Indexing and slicing operations in lists are same as in strings




```
course = ["Python","ML","IoT"]
```

Lists Methods

Lists:

Append() - adds the element at the end of the list.

```
course = ["Python","Html","Java","Machine Learning"]  #create list - course
print(course)    #print course
course.append('IoT')  #append IoT to list course
print(course)    #print course after appending 'IoT'
```

Lists:

extend() - extends the list by appending all the elements from the iterable.

```
#Extend
course = ["Python","Html","Java","Machine Learning"]
#create list - course
add_course = ['IoT','ReactJs']    #create another list
add_course
course.extend(add_course)   #extend add_course to list
course
print(course)    #print course after extending add_course
```

Lists:

Clear ()- clears all the element from the list.

```
#clear
course = ["Python","Html","Java","Machine Learning"]  #create list - course
print(course)    #print course
course.clear() #append IoT to list course
print(course)    #print course after clearing
```

Lists:

insert ()- insert an element at the given position.

```
#insert
course = ['Machine Learning','Data Science','Python','JavaScript']
print(course)
course.insert(1,'Java')  #insert 'java' at index 1
print(course)
```

Lists:

remove (x)- removes first element from the list whose value is x.

```
#remove
course = ['Machine Learning','Data Science','Python','JavaScript','Data Science']
print(course)
course.remove('Data Science')  #removes the first element with the value 'Data Science'
print(course)
```

Lists:

copy ()- returns the copy of the specified lists.

```
#copy
course = ['Machine Learning','Data Science','Python','JavaScript','Data Science']
x = course.copy()  #copy list course in x
print(course)
print(x)
```

Lists:

count ()- returns the number of times element appears in the list.

```
#count
course = ['Machine Learning','Data Science','Python','JavaScript','Data Science']
course.count('Data Science') #returns number of times data science appears in the list
```

Lists:

index ()- returns the index of the first element with the specified value.

```
#index
course = ['Machine Learning','Data Science','Python','JavaScript','Data Science']
course.index('Data Science') #returns first index of "data science"
```

Lists:

pop()- remove the item at the given position in the string and return it. If index is not specified it removes the last element and returns it.

```
#pop
last_course = course.pop(1)
#removes the element at index 1 and stores it
into last_course
print(course)
print(last_course)
```

Lists:

reverse()-reverse the element of the list in place.

```
#reverse
course = ['Machine Learning', 'Python',
'JavaScript', 'Data Science']
print(course)
course.reverse() #reverse the element in the list
print(course)
```

Lists:

sort()-sort the item of the lists in place.

```
#sort
course = ['Machine Learning', 'Python',
'JavaScript', 'Data Science']
print(course)
course.sort() #sort the element in the list
print(course)
```

Lists:

del()-permanently deletes the list

```
#delete
lst = ['Ram','John','Sam','Kartik']
print(lst)
del lst     #delete the lst
print(lst)
```

Lists:

Concatenate-the most conventional way of concatenating two lists is the use of “+” operator.

```
#concatenation
course_1 = ['Python','Java']
course_2 = ['ML','Data Science']
print(course_1 + course_2)  # concat course_1 and course_2
```

Tuples

Tuples

- Tuples are ordered collection of elements enclosed within a round bracket and separated by commas.
- Tuples are immutable data types that means we cannot add, update and delete the element of the tuples.
- Tuples can be indexed and sliced in similar ways as strings and lists.
- Tuples are also a type of iterables in python.
- We can store multiple kind of data types in tuples

Syntax to create tuples

```
tup = (element1, element2,.. ,elementn)
```

```
#tuples canot update, delete or remove
#we can store multiple data types
a = ('Ram','Rahul',23,23,True)
print(type(a))
print(a)
```

```
<class 'tuple'>
('Ram', 'Rahul', 23, 23, True)
```

Tuple Methods

Tuple:

count() - return the count of the specified element.

```
a = ('Ram','Rahul',23,23,True)
a.count(23)   #returns the count of element '23'
```

Tuple:

index() - returns the first index of specified elements.

```
a = ('Ram','Rahul',23,23,True)
a.index(23)   #returns the index of element '23'
```

Sets

Sets

- Sets are unordered collection of elements enclosed within a curly bracket and separated by commas.
- Sets are randomly ordered
- Set does not stores duplicate values.
- The elements contained in a set must be of immutable type.
- Sets cannot be indexed and sliced.

```
set_a = {1,5,8,9,10}  #create sets using curly braces
print(set_a)
print(type(a))

set_b = [2,5,6,9,7,12] # create sets using set()
print(set_b)
print(type(set_b))
```

```
{1, 5, 8, 9, 10}
<class 'tuple'>
[2, 5, 6, 9, 7, 12]
<class 'list'>
```

Syntax to create sets

```
s = {element1, element2,.. ,elementn}
s = set(iterable)
```

Set Methods

Set:

add() - add method adds the element to the set.

```
#add
a = {1,5,6,8,9}
print(a)
a.add(10)   #add 10 to set a
print(a)
```

Set:

len(setname) - returns the number of elements in the set

```
a = {1,5,6,8,9}
print(len(a))   #check the length of set a
```

Set:

s.issubset(t) - check if every element of s is in t.

```
#issubset()
t = {1,2,3,4,5,8}
s = {2,3,5}
s.issubset(t)   #check if every element of s is in t
```

Set:

`s.issuperset(t)` - returns True if all element in s is present in t

```
#issupersetset()
t = {1,2,3,4,5,8}
s = {2,3,5}
t.issuperset(s)   #checks if every element in t is in s
```

Set:

a.union(b) - returns all the elements from both the sets.

```
a = {1,5,6,8,9}  #set -a
print(a)
b = {5,8,9,7,10} #set -b
print(b)
a.union(b)    #union on set a and b
```

Set:

a.intersection(b) - returns the common element from bot the sets.

```
a = {1,5,6,8,9}  #set -a
print(a)
b = {5,8,9,7,10} #set -b
print(b)
a.intersection(b)    #intersection on set a and b
```

Set:

a.difference(b) - returns the elements from set a that are not there in set b.

```
a = {1,5,6,8,9}  #set -a
print(a)
b = {5,8,9,7,10} #set -b
print(b)
a.difference(b)  #you will get element from set a that are not there in set b
```

Set:

a.symmetric_difference(b) - returns the elements from set a and b that are not common.

```
a = {1,5,6,8,9}  #set -a
print(a)
b = {5,8,9,7,10} #set -b
print(b)
a.symmetric_difference(b) #returns new set with elements
either in s or t but not in both
```

Set:

update()- updates the items in current set by adding items in another set.

```
a = {1,2,3,4,5,8}
b = {10,12}
a.update(b) #updates set a with items of set b
print(a)
```

Set:

remove()- removes specified element from the set.

```
a = {1,2,3,4,5,8}
print(a)
a.remove(2) #removes 2 from set a
print(a)
```

Set:

discard()- removes specified element from the lists.

```
a = {1,2,3,4,5,8}
print(a)
a.discard(2) #removes 2 from set a
print(a)
```

Set:

clear()- removes all elements from set.

```
a = {2,9,8,7,5}
print(a)
a.clear()    #removes all the element from a
print(a)
```

Set:

del()- deletes the set.

```
a = {2,9,8,7,5}
print(a)  #deletes the set a
del a
print(a)
```

Set:

s.intersection_update(t)- return set s with only the elements in t

```
s = {1,2,3,4,5,8}
t = {2,3,5}
print(s)
s.intersection_update(t)  #return set s with only the elements
in t
print(s)
```

Set:

s.difference_update(t)- return set s with only the elements in t

```
s = {1,2,3,4,5,8}
t = {2,3,5}
print(s)
s.difference_update(t)  #return set s with only the elements
not present in t
print(s)
```

Set:

s.symmetric_difference_update(t)- return set s with only the elements in t

```
s = {1,2,3,4,5,8}
t = {2,3,5}
print(s)
s.symmetric_difference_update(t)  #return set s with  the elements from s or t but not both
print(s)
```

Set:

frozenset(t)- it makes the iterable immutable.

```
>>> a = {1,2,3,410,90,80}
>>> frozenset(a)
frozenset({80, 1, 2, 3, 90, 410})
```

Dictionary

Dictionary

Syntax to create dictionary

```
a = dict()  #creates empty dictionary a
print(a)
dct = {'Name' : 'Rahul','Age':12,'Class' : 12}
#creates a dictionary with the key and the value pair
print(dct)
```

- Dictionaries are mutable unordered collections written with curly braces
- Data is stored in key and value pair.
- The keys must be unique and should be hashable.

Accessing Items:

We can access the item of the dictionary by referring to its key name inside square bracket.

```
a = dict()  #creates empty dictionary a
dct = {'Name' : 'Rahul','Age':18,'Class' : 12}
dct['Name']  #returns value associated with the key-'Name'
```

Dictionary Methods

Dictionary:

clear() - removes all the item form the dictionary.

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964}
print(car)
car.clear()  #removes all the elements from the car
print(car)
```

Dictionary:

copy() - return a copy of the specified dict.

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964
       }
print(car)
new_c = car.copy()  #creates a copy of car
print(new_c)
```

Dictionary:

items() - returns an iterable list of (key,value) tuples

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964
       }
car.items()    #return an iterable list of (key,value) tuples
```

Dictionary:

keys() - returns an iterable of all the keys from the dictionary

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964
       }
car.keys()    #return an iterable of all keys in car
```

Dictionary:

Values() - returns an iterable of all the values from the dictionary

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964
       }
car.values()    #return an iterable of all keys in car
```

Dictionary:

popitem() - removes the last element entered in the dictionary and returns it

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964}
print(car)
a = car.popitem()    #removes ('year', 1964) from car and store
it in a
print(car)
print(a)
```

Dictionary:

pop() - removes the specified element from the dictionary and returns it.

```
car = {"brand": "Ford",
      "model": "Mustang",
      "year": 1964}
print(car)
#removes the key and value assosciated with specified key
print(car.pop('year'))
print(car)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
1964
{'brand': 'Ford', 'model': 'Mustang'}
```

Dictionary:

setdefault() - returns the value of the item specified. If the item is not present it return the default value that we set.

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964}
x = car.setdefault('model', 'Bronco')  #returns "Mustang" if the key is not present return 'Bronco'
print(x)
y = car.setdefault('place','India')    #returns "India" as the key "place" is not in car
print(y)
print(car)
```

Dictionary:

update() - adds the specified item to the dictionary.

```
car = {"brand": "Ford",
       "model": "Mustang",
       "year": 1964}
print(car)
car.update({'Color' : 'black'}) #updates the car with new sets
of key value pair
print(car)
```

Dictionary:

dict.fromkeys() - creates a dictionary from the elements of the iterable.

```
a = ["python","java"]
dict.fromkeys(a)
```

```
{'java': None, 'python': None}
```

```
a = ["python","java"]
dict.fromkeys(a,90)
```

```
{'java': 90, 'python': 90}
```

Conditional Statements

Conditional Statements in python performs different computations or actions depending on whether a specific boolean constraint evaluates to True of False.

If Statements:

Conditional Statements are handled by if statement in python. It is used for making decisions.

Syntax

```
If <expression> :
    statement-1
    statement-2
    ……..
    statement-n
```

```
a = 15
if a > 10 :
  print("A is greater than 10")
```

If- else Statements:

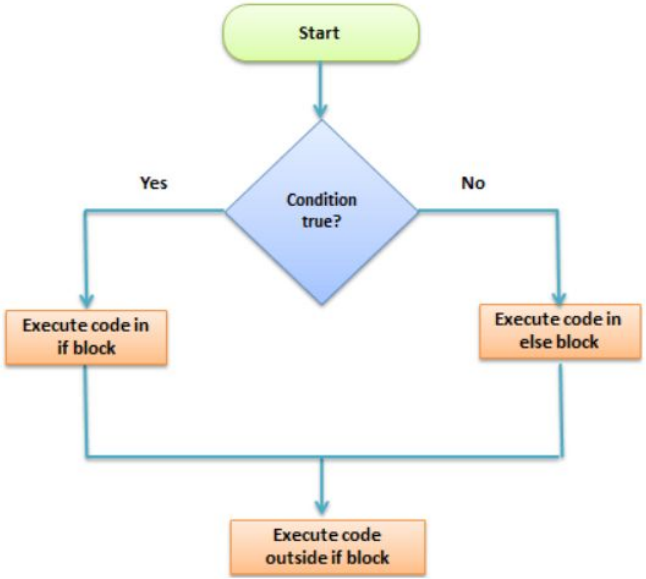
Syntax

```
If <expression> :
        statement
else :
        statement
```

When expression evaluates to True then if block is executed else the control is transferred to else block

```
a = 15
if a > 10 :
  print("A is greater than 10")
else:
  print("A is less than 10")
```

If- else Statement flowchart



If- elif Statements:

Syntax

```
If <expression> :
        statement
 elif <expression> :
        statement
 elif <expression> :
        statement
 else:
        statement
```

```
num = 10
if (num==0):
  print("Number is 0")
elif (num > 5):
  print("Number is greater than 5")
else:
  print("Number is less than 5")
```

Nested IF condition

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

```
Enter a number: 10
Positive number
```

Loops

While Loop:

Syntax

```
while <condition>:
    statement
```

While Loop first checks the condition. The loop will execute unless the condition becomes False.

```
a = 1
while a < 10:
    print(a)
    a +=1
```

For Loop:

Syntax

```
for var in <iterable>:
    statement
```

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string)

```
lst = [10,20,30,40,50]
for var in lst:
    print(var)
```

range():

```
range(5,10)
```

```
range(5, 10)
```

range() function generates the integer numbers between the start number and end number

range() function is mostly used with for loops

```
for var in range(10,20):
    print(var)
```

break:

```
a = int(input("Enter a number : "))
num = [11,22,33,44,55,66,70]
for i in num:
  if i == 44:
    print("Number is present in the list")
    break
else:
  print("Number is not there in the list")
```

```
Enter a number : 44
Number is present in the list
```

break is used for the premature termination of the loop.

continue:

```
for i in range(9):
  if i == 3:
    continue
  print(i)
```

```
0
1
2
4
5
6
7
8
```

continue is used to end the current iteration in a loop and continue from next iteration.

pass:

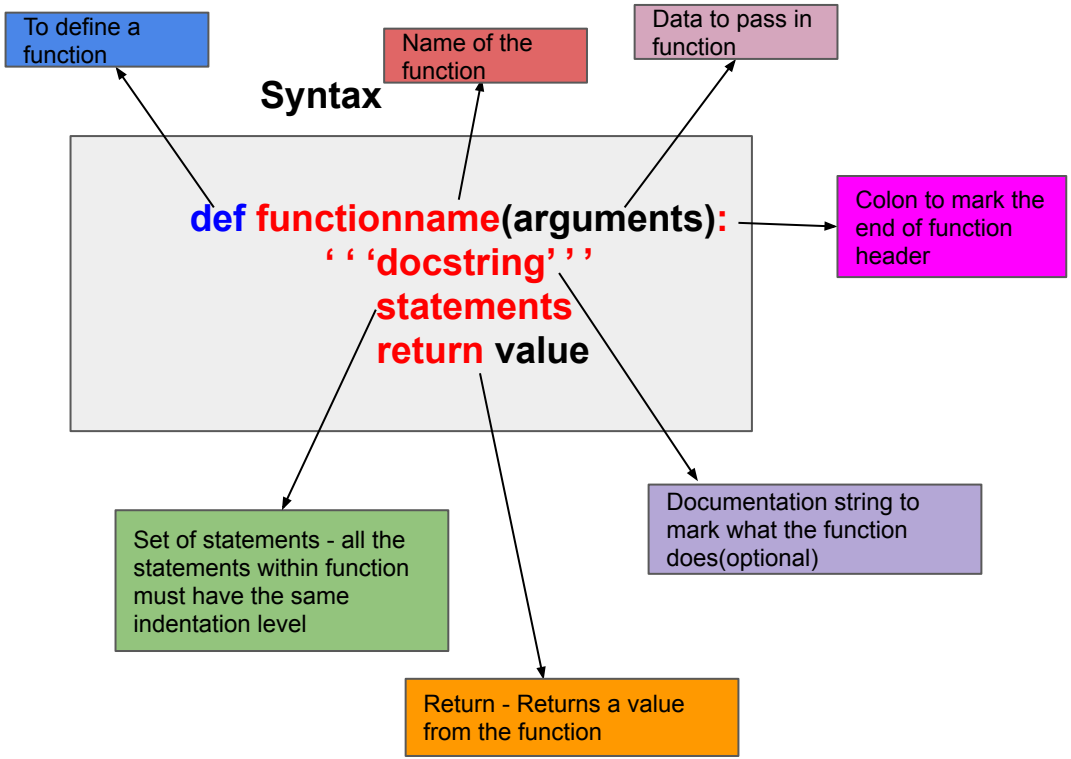
```
sequence = {'p', 'a', 's', 's'}
for val in sequence:
    pass
```

pass it is used when a statement is required syntactically but you do not want any command to execute

Functions

Functions:

- A function is a group of related statements that performs a specific task.
- Functions helps break our code into smaller & modular chunks.
- In python a function is defined using a **def** keyword.



Function Example:

```
def add(a,b):
  ''' Function to Add two Numbers
  Arguments:
     a : First number
     b : Second number '''
  print(a + b)
#a function executes when it is called we
call a function using the function name.
add(10,20)
```

Functions:

return statement

- The return statement is used to return some value from the function when it is called.
- When a return statement is not present in a function the returned value is **None**

```
def add(a,b):
  ''' Function to Add two Numbers
  Arguments:
    a : First number
    b : Second number
  Return:
    Returns sum of a and b'''
  sum = a + b
  return sum

addition = add(50,60)
print(addition)
```

Functions:

parameters / arguments

- Positional arguments
- Keyword arguments
- Default arguments
- Variable length arguments
- Keyword variable length arguments

Functions:

Positional arguments

Positional arguments are the arguments that need to be included in the proper position or order.

```
def greet(name,message):
    ''' A Function to greet a person
with message'''
    print(name + " " + message)

greet('Monica','good morning')
```

Functions:

Keyword arguments

Keyword arguments are the arguments that identifies the arguments by their name.

```
def grocery(item,price):
  '''Function with two keyword arguments
'item','price'
  '''
  print(f'Price: {price}')
  print(f'Item name : {item}')

#call the function and pass the keyword arguments
value
grocery(item='Soap',price=44)
```

Functions:

Default arguments

default arguments are used to set the default value for the arguments.

```
def grocery(item,price=50.5):
  '''Function with default
  '''
  print(f'Price: {price}')
  print(f'Item name : {item}')

#call the function and pass the value for
parameter item
grocery(item='Soap')
```

Functions:

Variable length arguments

variable length arguments can accept n number of arguments. It is written with a “*” symbol (*args).

```
def my_sum(*args):
  ''' Variable length arguments'''
  return sum(args)

my_sum(10,20,30)
```

Functions:

Keyword Variable length arguments

Keyword variable length arguments can accept n number of arguments provided in key value pair. It is written with a “**” symbol (*kargs).

```
def k_arg_func(**kargs):
  print(kargs)

  for x,y in kargs.items():
    print(f'Keyowrd : {x}, value : {y}')

k_arg_func(name='Codegnan',course_1 = 'Data science')
```

```
{'name': 'Codegnan', 'course_1': 'Data science'}
Keyowrd : name, value : Codegnan
Keyowrd : course_1, value : Data science
```

Global and Local Variables

Functions:

Global variable

A variable declared outside of a function or in global scope is known as global variable. Global variables can be accessed inside and outside of a function.

```
x = "Gloabl scope variable" #declare a global scope variable
def foo():
  '''A function that uses global variable'''
  print(x)
foo()  #call the function
```

If we change the value of x inside the function it will throw an error. We have to use global keyword while declaring variable x, if we want to change the value inside the function

```
x = 10
def add5():
  global x
  x = x + 5
  print(x)
add5(10)
```

Functions:

Local variable

The variables declared inside a function body are known as local variables.
Local variables cannot be accessed outside a function body.

```
def local_example():
  ''' Example of local variable'''
  x = "Hello"
  print(x)
local_example()
```

If you try to access a local variable outside a function body it will throw an error.

```
def local_example():
  ''' Example of local variable'''
  z = "Hello"
  print(z)
local_example()
print(z)
```

```
Hello
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-baad818253d7> in <module>()
      4   print(z)
      5 local_example()
----> 6 print(z)

NameError: name 'z' is not defined
```

Functions:

Function inside another function

```
def display(name): #main function
  ''' Function inside a function to greet'''
  def message():  #function inside a function
    return "Good Morning!"
  result = message () + ' ' +  name
  return result

display('Manvendra')
```

Functions:

Function as a parameter to another function.

```
def display(name,func):
  '''A function that accepts
another function as argument'''
  return name + ' ' + func
def message():
  return 'Good Morning!'

display('Manvendra',message())
```

Functions:

A function can return another function.

```
def display():
  def message():
    return "Good morning"
  return message()
display()
```

Anonymous Function : Lambda

Lambda:

An anonymous function is a function which has no name.

- An anonymous function is a function which has no name.
- Normal functions are defined using **def** keyword, while lambda functions are defined using **lambda** keyword.
- Lambda function can have any number of argument but only 1 expression

Syntax

lambda arguments : expression

```
double = lambda x : x*2
print(double(5))
```

Use of Lambda :

With map() & filter()

Lambda with filter():

The filter function in python takes a function and a list as arguments.

```
lst = [1,2,3,4,5,6,7,8,9,10]
#filter out only even values from a list
new_lst = list(filter(lambda x : (x%2 == 0),lst))
print(new_lst)
```

Lambda with map():

The map() function is similar to the filter() function but acts on each element and perhaps changes that element.

```
lst = [1,2,3,4,5,6,7,8,9,10]
#program to square each element in the list using map()
new_lst = list(map(lambda x : x**2,lst))
print(new_lst)
```

List Comprehension

List Comprehension

- Lists can be built by leveraging any iterable strings,tuples.
- List comprehension consists of an iterable containing an expression followed by for clause

Iterating through a loop using for loop.

```
h_letters = []
for letter in 'human':
    h_letters.append(letter)
print(h_letters)
```

Conditionals in list comprehension

```
lst = [x for x in range(10) if x%2==0]
print(lst)
```

Iterating through a loop using list comprehension

```
h_letters = [x for x in 'human']
print(h_letters)
```

Dict Comprehension

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
# Double each value in the dictionary
double_dict1 = {k:v*2 for (k,v) in dict1.items()}
print(double_dict1)
```

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f':6}

# Identify odd and even entries
dict1_tripleCond = {k:('even' if v%2==0 else 'odd') for (k,v) in dict1.items()}

print(dict1_tripleCond)
```