In [1]:

```
# Conditional Statement

x = int(input("Enter your age:")) # This is called Type Casting.
if(x>=18):
    print("You can cast your vote!")
else:
    print("Come after you are 18...")
```

```
Enter your age:22
You can cast your vote!
```

Looping in Python

In [5]:

```
# Loop or iteration

student_1 =["Devjeet","Roy",76,"CSE","B","NSEC","MAKAUT"]
for i in student_1:
    print(student_1.index(i),i) # Focus on tis line now, we will come back here soon
```

```
0 Devjeet
1 Roy
2 76
3 CSE
4 B
5 NSEC
6 MAKAUT
```

In [6]:

```
# Range Function in Python 3.x

# To iterate over  series of numbers the range() function comes handy.

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

```
0
1
2
3
4
```

In [7]:

```
# To iterate over numbers in a given range
for i in range(5,10):
    print(i)
```

```
5
6
7
8
9
```

In [8]:

```
# range() function is extremely useful to work using Progressive Series or Sequence

# Arithmetic Progresson with AD = 2
for i in range(5,10,2):
    print(i)
```

```
5
7
9
```

In [10]:

```
# Arithmetic Progresson with AD = 5
for i in range(5,30,5):
    print(i)
```

```
5
10
15
20
25
```

In [13]:

```
# break & continue statement in python
# The break statement, like in C, breaks out of the innermost enclosing for or while loop.
names= ["Abhik","Bikash","Navin","Satyam"]
count=0
for i in names:
    if(i=="Navin"):
        print("Navin Found in your list")
        break
    else:
        count=count+1
print("Count is:",count)
```

```
Navin Found in your list
Count is: 2
```

In [14]:

```
# continue statement is used to ignore the statements after it when the condition is met.
myExp=["C","C++","JavaScript","Java","Python"]
c=0
for i in myExp:
    if(i=="C++"):
        continue
        print("I know C++, but it will not get printed.") # continue throws the control to the beg
inning of loop.
    else:
        print("I know:",i)
```

```
I know: C
I know: JavaScript
I know: Java
I know: Python
```

Functions in Python

In [16]:

```
# pass statementin Python
# In Python, pass is a null statement. The interpreter does not ignore a pass statement,
# but nothing happens and the statement results into no operation.

# The pass statement is useful when you don't write the implementation of a function but
# you want to implement it in the future.

def add(num1, num2):
    # Implementation details
    pass
add(3,4)
print("See...nothing happens!")
```

```
See...nothing happens!
```

In [19]:

```
# pass is also used to ignore if statement
def display(number):
    if number is 2:
        pass
    else:
        print("See Buddy....the number is not 2")
display(5)
display(2)
display(44)
```

```
See Buddy....the number is not 2
See Buddy....the number is not 2
```

In [21]:

```
# pass is also used to implement a minimal class, without its implementation details...
class myNewClass:
    pass
myclass = myNewClass()
print("Abstract class ready!")
```

```
Abstract class ready!
```

In [23]:

```
# Playing with Functions
# Why we use Function & when we should not use function!
def addition(n1,n2):
    sum= n1+n2
    print("The sum is:",sum)
addition(10,5)
```

```
The sum is: 15
```

In [24]:

```
# Returning function
def add(a,b):
    c=a+b
    return c
sum= add(40,5)
print("For returning function, the sum is:",sum)
```

```
For returning function, the sum is: 45
```

Coding Activity : Fibonacci Series

In [10]:

```
# Lets code out the fibonacci series
def fib(n):    # write Fibonacci series up to n
     a, b = 0, 1
     while a < n:
        print(a, end=' ')
        a, b = b, a+b
        # print()

# Now call the function we just defined:
fib(10)
```

```
0 1 1 2 3 5 8
```

In [3]:

```
# Fibonacci to find 1st n terms
def fibo(limit):
    i=1
    a=0
    b=1
    print(a,b,end=' ')
    while(i<=limit):
        a,b=b,a+b
        print(b, end=' ')
        i=i+1
    print("\nThe above ones are the Fibo terms!")
n = int(input("Enter the number of terms you wanna print:"))
n=n-2
fibo(n)
```

```
Enter the number of terms you wanna print:8
0 1 1 2 3 5 8 13
The above ones are the Fibo terms!
```

In [6]:

```
# type() method returns class type of the argument(object) passed as parameter.
# type() function is mostly used for debugging purposes.
print(type(fibo))
```

```
<class 'function'>
```

In [9]:

```
# Let us proof that fibo is not an object
mySeries = fibo
new = int(input("Enter the number of terms you wanna print:"))
new=new-2
mySeries(new)
# If fibo was not of type function the staff would not have worked this way!
```

```
Enter the number of terms you wanna print:8
0 1 1 2 3 5 8 13
The above ones are the Fibo terms!
```

Detailed view of Functions, Arguments & Parameters

In [6]:

```
# Working with default arguments in function
def validate(name, age, stream="CSE", college="NSEC"):
    print("The name is:",name,"and the age is:",age)
    print("He is from",stream,",",college)
# Function Call
validate("Devjeet Roy", 21)
```

```
The name is: Devjeet Roy and the age is: 21
He is from CSE , NSEC
```

In [1]:

```
# The default values are evaluated at the point of function definition in the defining scope
i = 6
def f(arg=i):
    print(arg)
i = 8
f()
print(i)
```

```
6
```

8

In [27]:

```
# The default value is evaluated only once. This makes a difference when the default is a mutable
object
# such as a list, dictionary, or instances of most classes.
def count(a,tasks=[]):
    tasks.append(a)
    print(tasks)
count(1)
count("Dj")
count(5)
```

```
[1]
[1, 'Dj']
[1, 'Dj', 5]
```

In [28]:

```
# We can debug the above trouble by...
def count(a,tasks=[]):
    tasks=[] # Everytime initializes the list fresh when ever the function is called.
    tasks.append(a)
    print(tasks)
count(1)
count("Dj")
count(5)
```

```
[1]
['Dj']
[5]
```

In [17]:

```
# Positional Argument & Keyword Argument
def simple_Interest_Calculation(p,t=5,r=7):
    si = p*t*r/100
    return si
interest = simple_Interest_Calculation(p=5000)  #Keyword Argument
print(interest)
interest = simple_Interest_Calculation(10000)  #Positional Argument
print(interest)
interest = simple_Interest_Calculation(p=5000,t=10)  # 2 Keyword Argument
print(interest)
interest = simple_Interest_Calculation(5000, t=10)  #Keyword Argument followed by positial
argument, reverse is an error!
print(interest)
#interest = simple_Interest_Calculation(5000, p=10000)  #Duplicate Argument error!
#print(interest)
```

```
1750.0
3500.0
3500.0
3500.0
```

In [37]:

```
# When a final formal parameter of the form **name is present, it receives a dictionary.
# a formal parameter of the form *name which receives a tuple containing the positional
# arguments beyond the formal parameter list.
# (*name must occur before **name.)
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Special Parameters

In [24]:

```
# Positional Only Arguments
# If positional-only, the parameters' order matters, and the parameters cannot be passed by keywor
d.
# Positional-only parameters are placed before a / (forward-slash).

def pos_only_arg(arg, /):
    print(arg)

# Function Call
pos_only_arg(1)
```

```
1
```

In [29]:

```
# Keyword Only Arguments
# To mark parameters as keyword-only, indicating the parameters must be passed by keyword
argument,
# place an * in the arguments list just before the first keyword-only parameter.
def kwd_only_arg(*, arg):
     print(arg)

# Function Call
kwd_only_arg(arg=4)
```

```
4
```

In [ ]:

```
# NOTE : non-default arguments cannot follow default arguments
```

Python Arbitrary Arguments

In [30]:

```
def greet(*names):
   """This function greets all
   the person in the names tuple."""

   # names is a tuple with arguments
   for name in names:
       print("How are you",name,"?")

greet("Monty","Devjeet","Sumit","JD")
help(greet)
```

```
How are you Monty ?
```

```
How are you Devjeet ?
How are you Sumit ?
How are you JD ?
Help on function greet in module __main__:

greet(*names)
    This function greets all
    the person in the names tuple.
```

In [34]:

```
def concat(*args, sep="."):
     return sep.join(args)

concat("devjeetroy", "dr", "@gmail","com")
```

Out[34]:

```
'devjeetroy.dr.@gmail.com'
```

Lambda Expression

In [14]:

```
def make_incrementor(n):
     return lambda x: x + n
sum = make_incrementor(5)
sum(1)
```

Out[14]:

```
6
```

Playing with docstrings

In [17]:

```
# Python documentation strings (or docstrings) provide a convenient way of associating
documentation with Python modules,
# functions, classes, and methods.

# The first line should be a short description.
# If there are more lines in the documentation string, the second line should be blank,
# visually separating the summary from the rest of the description.
# The following lines should be one or more paragraphs describing the object's calling
conventions, its side effects, etc.

def my_function():
    """Demonstrate docstrings and does nothing really."""

    return None

print ("Using __doc__:")
print (my_function.__doc__ )

print ("Using help:")
help(my_function)
```

```
Using __doc__:
Demonstrate docstrings and does nothing really.
Using help:
Help on function my_function in module __main__:

my_function()
    Demonstrate docstrings and does nothing really.
```

Yield instead of Return in Function

In [5]:

```
'''
The yield statement suspends function's execution and sends a value back to the caller,
but retains enough state to enable function to resume where it is left off. When resumed,
the function continues execution immediately after the last yield run. This allows its code
to produce a series of values over time, rather than computing them at once and sending them back
like a list.
'''
def mySq():
    i=1
    while(True):
        if(i<=10):
            yield i**2
            i=i+1
        else:
            break
for value in mySq():
    print(value,end =" ")
```

```
1 4 9 16 25 36 49 64 81 100
```

Partial Functon in Python 3.x

In [6]:

```
from functools import partial

# A normal function
def f(a, b, c, x):
    return 1000*a + 100*b + 10*c + x

# A partial function that calls f with
# a as 3, b as 1 and c as 4.
g = partial(f, 3, 1, 4)

# Calling g()
print(g(5))
```

```
3145
```

In [10]:

```
from functools import partial
# A Normal function
def interest(p,t,r):
    return (p*t*r)/100
# A partial function that calls interest()
calcu_interest = partial(interest,10000,5)
# calcu_interest(p=10000,t=5,r)
calcu_interest(5)
```

Out[10]:

```
2500.0
```

First Class Function in Python

In [2]:

```
# Python functions are first class objects.
# In the example below, we are assigning function to a variable.
# This assignment doesn't call the function.
# It takes the function object referenced by shout and creates a second name pointing to it, yell.
# Python program to illustrate functions
```

```
# can be treated as objects
def shout(text):
    return text.upper()

print (shout('Hello!'))

yell = shout

print (yell('Hello!'))
```

```
HELLO!
HELLO!
```

In [ ]:

```

```