functions

built-in functions

These functions are defined and pre-coded in python. Some examples of built-in functions are as follows:

min(), max(), len(), sum(), type(), range(), dict(), list(), tuple(), set(), print(), etc.

user-defined functions

We can create functions to perform specific tasks as per our needs. Such functions are called user-defined functions.

Syntax:

def function_name(parameters):

```
    Code and Statements
```

Rules to write function

- Create a function using the def keyword, followed by a function name, followed by a paranthesis (()) and a colon(:).
- Any parameters and arguments should be placed within the parentheses.
- Rules to naming function are similar to that of naming variables.

In [ ]:
```
def my_func(param1='default'):
    """
    Docstring goes here.
    """
    print(param1)
```

In [ ]:
```
my_func
```

In [ ]:
```
my_func()
```

In [ ]:
```
my_func('new param')
```

In [ ]:
```
my_func(param1='vishal')
```

In [ ]:
```
def square(x):
    return x**2

square(3)
```

In [ ]:
```
def name(fname, lname):
    print("Hello,", fname, lname)

name("Abhinav", "Bindra")
```

Function Arguments

There are four types of arguments that we can provide in a function:

1. Default Arguments
2. Keyword Arguments
3. Required Arguments
4. Variable-length Arguments

Default arguments

Default value can be provided while creating a function. This way the function assumes a default value even a value is not given in the function call for that argument.

In [ ]:
```
def pizza(size, name = 'Mushroom pizza'):
    print("I like" , size, name)

pizza("Regular")
```

Keyword arguments

Keyword arguments with key = value, this way the interpreter recognizes the arguments by the parameter name. Hence, the the order in which the arguments are passed does not matter.

In [ ]:
```
def pizza(size, name):
    print("I like" , size, name)

pizza(size="Regular",name= "Mushroom")
```

Required arguments

if we don’t pass the arguments with a key = value syntax, then it is necessary to pass the arguments in the correct positional order and the number of arguments passed should match with actual function definition.

In [ ]:
```
def pizza(size, name, loc):
    print("I like", size, name, " pizza of " , loc)

pizza("Regular","Mushroom","Dominos")
```

Variable-length arguments:

Sometimes we may need to pass more arguments than those defined in the actual function. This can be done using variable-length arguments.

There are two ways to achieve this:

Arbitrary Arguments:

While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of tuple.

In [ ]:
```
def pizza(*t):
    print("I like", t[0], t[1],"pizza of", t[2])

pizza("Regular", "Mushroom", "Dominos")
```

Keyword Arbitrary Arguments

While creating a function, pass a ** before the parameter name while defining the function. The function accesses the arguments by processing them in the form of dictionary.

In [ ]:
```
def pizza(**t):
    print("I like", t["size"], t["name"],"pizza of", t["loc"])

pizza(size= "Regular",name = "Mushroom", loc = "Dominos")
```

return Statement

The return statement is used to return the value of the expression back to the main function.

In [ ]:
```
def pizza(size, name, loc):
    return "I like" +" "+ size + " " + name + " pizza of " + loc

pizza("Regular","Mushroom","Dominos")
```

Python Recursion

We can let the function call itself, such a process is known as calling a function recursively in python

In [ ]:
```
def factorial(num):
    if (num == 1 or num == 0):
        return 1
    else:
        return (num * factorial(num - 1))

num =7
print("Factorial: ",factorial(num))
```

Question 1

Find the largest item from a given list

[10, 40, 50, 264, 389, 1, 4, 10, 29]

In [ ]:
```
max([10, 40, 50, 264, 389, 1, 4, 10, 29])
```

Question 2

Create a function name **eligibility** to check its marks in math and science is greater than 70 and attendance is more than 70 is eligible for test. Note: Marks of maths and science should be entered by user

lambda expressions

Lambda Functions are anonymous function means that the function is without a name. The lambda keyword is used to define an anonymous function in Python.

Syntax: lambda arguments: expression

1. This function can have any number of arguments but only one expression, which is evaluated and returned.
2. One is free to use lambda functions wherever function objects are required.
3. lambda functions are syntactically restricted to a single expression.

In [1]:
```
def times(var):
    return var*2

times(5)
```

Out[1]:
```
10
```

In [2]:
```
lambda var: var*2
```

Out[2]:
```
<function __main__.<lambda>(var)>
```

In [3]:
```
times = lambda var: var*2

times(5)
```

Out[3]:
```
10
```

Python Lambda Function with List Comprehension

In [4]:
```
even = [lambda x: x * 10 for x in range(1, 5)]

# iterate on each lambda function

# and invoke the function to get the calculated value
for i in even:
    print(i)
```

```
<function <listcomp>.<lambda> at 0x000001F1EF65EE50>
<function <listcomp>.<lambda> at 0x000001F1EF65E9D0>
<function <listcomp>.<lambda> at 0x000001F1EF65EC10>
<function <listcomp>.<lambda> at 0x000001F1EF65EA60>
```

map and filter

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Filter out all odd numbers using filter() and lambda function

Python Lambda Function with if-else

In [5]:
```
x = lambda a, b : a if(a > b) else b

print(x(10, 15))
```

```
15
```

In [6]:
```
def times(var):
    return var*2

seq = [1,2,3,4,5]

map(times,seq)
```

Out[6]:
```
<map at 0x1f1ef64e880>
```

In [7]:
```
list(map(times,seq))
```

Out[7]:
```
[2, 4, 6, 8, 10]
```

In [8]:
```
list(map(lambda var: var*2,seq))
```

Out[8]:
```
[2, 4, 6, 8, 10]
```

In [ ]:
```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

t = lambda x: (x % 2 != 0)

m = filter(t,li)

f = list(m)
print(f)
```

In [ ]:
```

```

Question: Filter all people having age more than 18, using lambda and filter() function

a = [13, 90, 17, 59, 21, 60, 5]

In [ ]:
```

```

Python Modules

Python modules are python files which contains python code that can be used within another python files ensuring simplicity and code reusability.

Some built-in modules:

csv, datetime, json, math, random, sqlite3, statistics, tkinter, turtle, etc.

In [9]:
```
# importing math module

import math

print("Sin(0) =", math.sin(0))
print("Cos(30) =", math.cos(math.pi/6))
print("Tanh(45) =", math.tanh(math.pi/4))
```

```
Sin(0) = 0.0
Cos(30) = 0.8660254037844387
Tanh(45) = 0.6557942026326724
```

Creating and using module

Write code and save file with extension .py

In [10]:
```
import operations

num1 = int(input("First number:"))
num2 = int(input("Second number:"))

print("Add", operations.add(num1, num2))
print("Sub", operations.sub(num1, num2))
print("Square", operations.square(num1))
print("Cube", operations.cube(num2))
```

```
First number:20
Second number:5
Add 25
Sub 15
Square 400
Cube 125
```

Using an alias:

alias can also use while importing module. So that, we do not need to write the whole name of the module while calling it.

In [ ]:
```
import operations as op

num1 = int(input("First number:"))
num2 = int(input("Second number:"))

print("Add", op.add(num1, num2))
print("Sub", op.sub(num1, num2))
print("Square", op.square(num1))
print("Cube", op.cube(num2))
```

Use an asterisk(*) while importing.

import everything from a module but we do not need to prefix module name again and again

In [ ]:
```
from operations import *

num1 = int(input("First number:"))
num2 = int(input("Second number:"))

print("Add", operations.add(num1, num2))
print("Sub", operations.sub(num1, num2))
print("Square", operations.square(num1))
print("Cube", operations.cube(num2))
```

dir() function

lists all the function names (or variable names) in a module.

In [12]:
```
import math

lst1 = dir(math)
print(lst1)
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'ac
osh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign',
'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fab
s', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lg
amma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan',
'tanh', 'tau', 'trunc', 'ulp']
```

In [ ]:
```

```