Minia University

Faculty of Computers & information

Artificial Neural Networks and Deep Learning

Slides By:

T.A. Sarah Osama Talaat

✉ E-mail: SarahOsama.fci@gmail.com

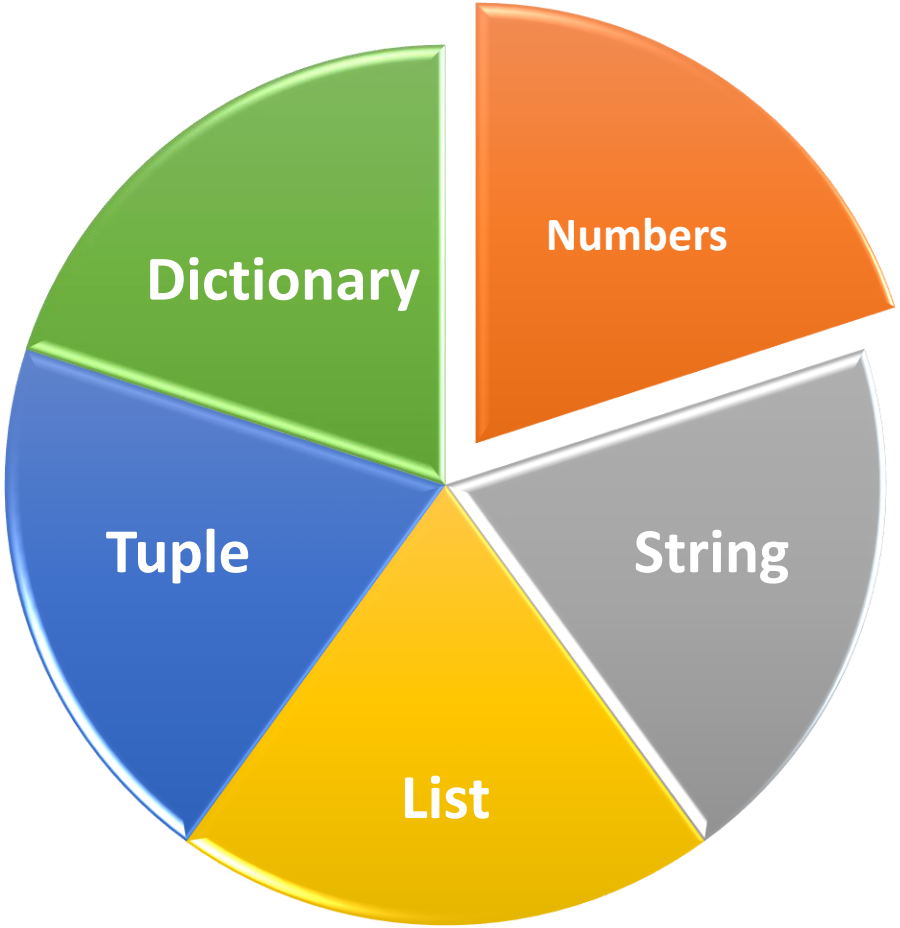
Slides were prepared based on set of references mentioned in the last slide

Agenda

- Review
- Type casting
- Basic operators
- Decision making
- Loops
- Functions

Let's Start

Review



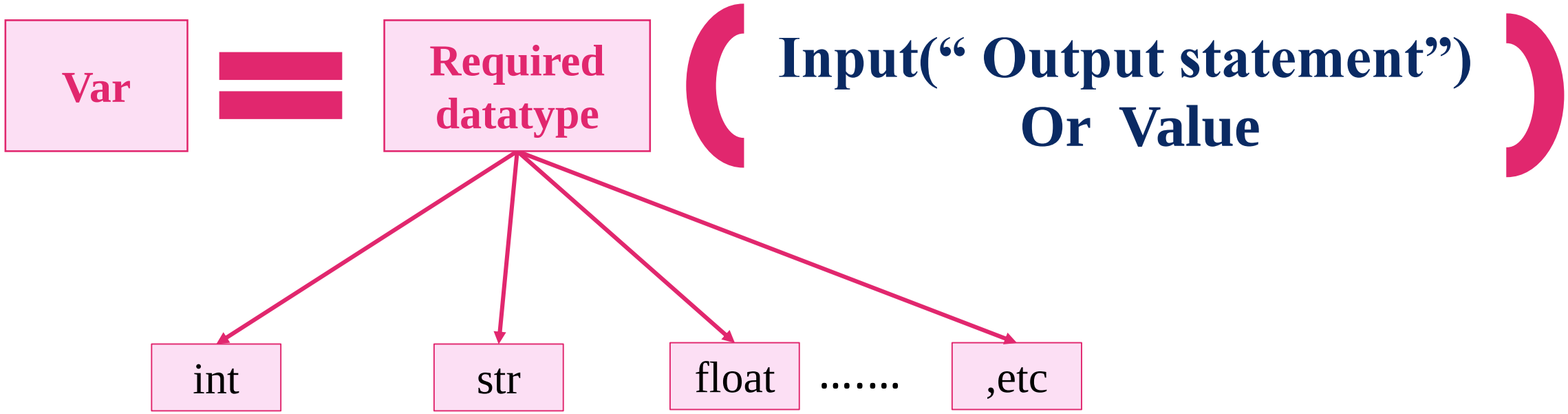
Review

❑Read input from user:

```
In [23]: Var1 = input("Please enter the first number: ")

         Please enter the first number: 100

In [24]: type(Var1)

Out[24]: str
```

What is the type of variable *Var1*?

Type Casting



Type Casting

```
In [27]: Var1 = int(input("Please enter the first number: "))
         Please enter the first number: 100

In [28]: type(Var1)
Out[28]: int

In [29]: Var1 = float(input("Please enter the first number: "))
         Please enter the first number: 0.5

In [30]: type(Var1)
Out[30]: float
```

Type Casting

- **Type Casting (cont.):**

```
In [31]: Value = 101

In [32]: type(Value)
Out[32]: int

In [33]: str(Value)
Out[33]: '101'

In [34]: float(Value)
Out[34]: 101.0

In [35]: complex(Value)
Out[35]: (101+0j)

In [36]: str(101+0j)
Out[36]: '(101+0j)'
```

Type Casting

- **Type Casting (cont.):**
  - We can not convert the complex number to the float number but we can convert the float number to the complex number.

```
In [8]: float(1256+0j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-d6c3dcab07d3> in <module>()
----> 1 float(1256+0j)

TypeError: can't convert complex to float
```

```
In [9]: complex(1256.0)
Out[9]: (1256+0j)
```

Type Casting

- **Type Casting (cont.):**
  - We can not convert the integer number to the list.

```
In [37]: Value = 12345

In [38]: List = list(Value)
```

```
---------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-38-ac4e8c4903f8> in <module>()
----> 1 List = list(Value)

TypeError: 'int' object is not iterable
```

Type Casting

- **Type Casting (cont.):**
  - We can not convert the integer number to the list.

```
In [39]: List = list(str(Value))

In [40]: List
Out[40]: ['1', '2', '3', '4', '5']

In [41]: List = int(List)
```

```
---------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
<ipython-input-41-46b7a190d31d> in <module>()
----> 1 List = int(List)

TypeError: int() argument must be a string, a bytes-like object or a numb
er, not 'list'
```

Type Casting

Type Casting (cont.):

- We can convert the list to tuple and vice versa.

In [46]:
```
1 Value = 123456
2 List = str(Value)
3 print(List)
```

```
123456
```

In [49]:
```
1 Tuple = tuple(List)
2 print(Tuple)
```

```
('1', '2', '3', '4', '5', '6')
```

In [51]:
```
1 List = list(Tuple)
2 print(List)
```

```
['1', '2', '3', '4', '5', '6']
```

Type Casting

Type Casting (cont.):

```
In [52]:
1 Value = 123456
2 Dictionary = dict(str(Value))
3 print(Dictionary)
```

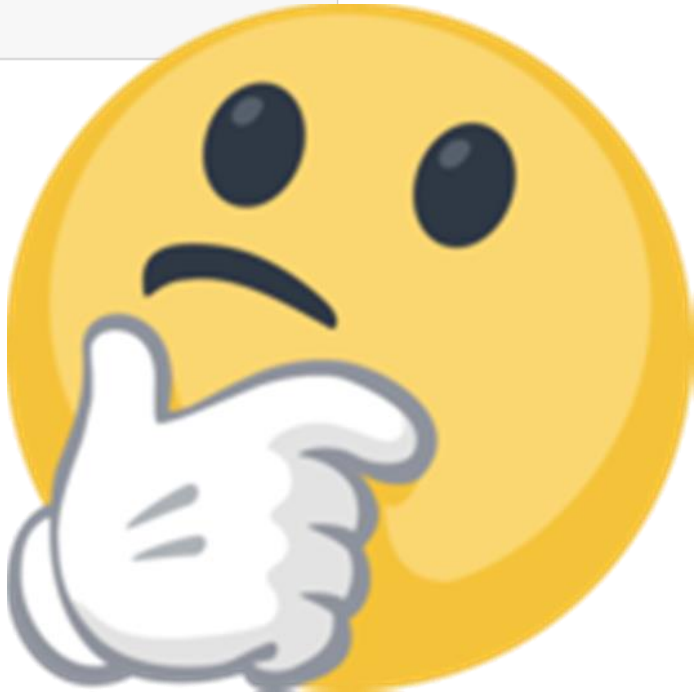
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-52-42c88e932bc1> in <module>()
      1 Value = 123456
----> 2 Dictionary = dict(str(Value))
      3 print(Dictionary)

ValueError: dictionary update sequence element #0 has length 1; 2 is required
```

Type Casting

Type Casting (cont.):

In [59]:
```
1 Value = 123456
2 List = list(str(Value))
3 Dictionary = dict(List)
4 print(Dictionary)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-59-7d92e1bcc67f> in <module>()
      1 Value = 123456
      2 List = list(str(Value))
----> 3 Dictionary = dict(List)
      4 print(Dictionary)

ValueError: dictionary update sequence element #0 has length 1; 2 is required
```



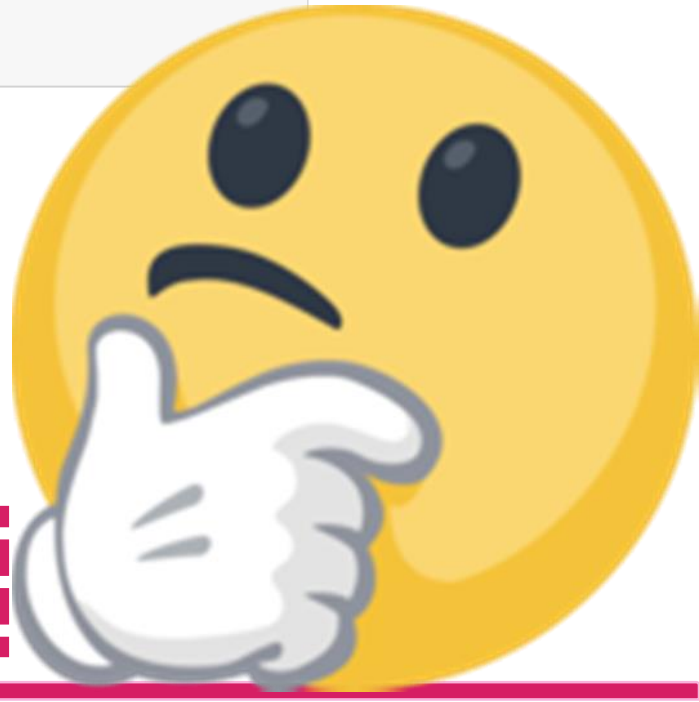
Type Casting

Type Casting (cont.):

In [62]:

```
1 Value = 123456
2 List = list(str(Value))
3 Tuple = tuple(List)
4 Dictionary = dict(Tuple)
5 print(Dictionary)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-ecebe80d0f51> in <module>()
      2 List = list(str(Value))
      3 Tuple = tuple(List)
----> 4 Dictionary = dict(Tuple)
      5 print(Dictionary)

ValueError: dictionary update sequence element #0 has length 1; 2 is required
```



Type Casting

- **Type Casting (cont.):**
  - To creates a dictionary from variable d, d must be a sequence of (key ,value) tuples (list of tuples).

```
In [66]: 1 List = [('Age',25),('Name','Hadeer')]
         2 Dictionary = dict(List)
         3 print(Dictionary)

{'Age': 25, 'Name': 'Hadeer'}
```

Basic Operators

- Arithmetic Operators
- Comparison Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Basic Operators:
Arithmetic Operators

In [68]:
```
1 number1 = 10
2 number2 = 5
```

In [71]:
```
1 # Addation
2 number1+number2
```

Out[71]: 15

In [72]:
```
1 # Subtraction
2 number1-number2
```

Out[72]: 5

In [73]:
```
1 # Division
2 number1/number2
```

Out[73]: 2.0

Basic Operators:
Arithmetic Operators

In [79]:
```
1 number1 = 10
2 number2 = 3
```

In [80]:
```
1 # Power
2 number1**number2
```

Out[80]: 1000

In [81]:
```
1 # Modulus
2 number1%number2
```

Out[81]: 1

In [84]:
```
1 # Floor division
2 number1//number2
```

Out[84]: 3

In [85]:
```
1 # Multiplication
2 number1*number2
```

Out[85]: 30

Basic Operators:

Comparison Operators

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) **is not true**.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) **is true**.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) **is true.**
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) **is not true**.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) **is true.**
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) **is not true.**

Basic Operators:

Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b, assigns value of a + b into c
+=	It adds right operand to the left operand and assign the result to left operand	c += a, is equivalent to c = c + a
-=	It subtracts right operand from the left operand and assign the result to left operand	c -= a, is equivalent to c = c – a
*=	It multiplies right operand with the left operand and assign the result to left operand	c *= a, is equivalent to c = c * a
/=	It divides left operand with the right operand and assign the result to left operand	c /= a, is equivalent to c = c / a
%=	It takes modulus using two operands and assign the result to left operand	c %= a, is equivalent to c = c % a
**=	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a, is equivalent to c = c ** a
//=	It performs floor division on operators and assign value to the left operand	c //= a, is equivalent to c = c // a

Basic Operators:
Logical Operators

- Assume variable **a** holds **True** and variable **b** holds **False**

Operator	Description	Example
and	If both the operands are true then condition becomes true.	(a and b) **is False.**
or	If any of the two operands are non-zero then condition becomes true.	(a or b) **is True.**
not	Used to reverse the logical state of its operand.	Not(a and b) **is True.**

Basic Operators:
Bitwise Operators

- Assume if a = 0011 1100 and b = 0000 1101

Operator	Description	Example
& (Binary AND)	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)
| (Binary OR)	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ (Binary XOR)	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ (Binary Ones Complement)	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< (Binary Left Shift)	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> (Binary Right Shift)	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Basic Operators:
Bitwise Operators

- Python's built-in function bin() can be used to obtain binary representation of an integer number.

```
In [38]: bin(60)

Out[38]: '0b111100'
```

```
In [63]: bin(13)

Out[63]: '0b1101'
```

Basic Operators:
Bitwise Operators

- By another way 👇

```
In [53]: "{0:b}".format(60)
Out[53]: '111100'
```

- a & b

```
In [40]: a=bin(60)

In [61]: b=bin(6)

In [58]: a&b
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-1da7d614f5f9> in <module>()
----> 1 a&b

TypeError: unsupported operand type(s) for &: 'str' and 'str'
```

Basic Operators:
Bitwise Operators

- You can convert between a string representation of the binary using bin() and int()

```
In [69]: bin(int(a,2)&int(b,2))
Out[69]: '0b1100'

In [56]: a="{0:b}".format(60)

In [70]: b="{0:b}".format(13)

In [71]: c="{0:b}".format(int(a)&int(b))

In [72]: c
Out[72]: '1001100'
```

Basic Operators:
Bitwise Operators

- You can convert between a string representation of the binary using bin() and int()

```
In [69]: bin(int(a,2)&int(b,2))
Out[69]: '0b1100'

In [56]: a="{0:b}

In [70]: b="{0:b}

In [71]: c="{0:b}

In [72]: c
Out[72]: '1001100'
```

Basic Operators:
Membership Operators

- Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

Operator	Description	Example

in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	`x=10`
		`y=[5,10,15,20]`
		`if(x in y):`
		`  print('Yes')`
		`Yes`

not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	`x=50`
		`y=[5,10,15,20]`
		`if(x not in y):`
		`  print('Yes')`
		`Yes`

Decision Making

- **Syntax:**

```
if condition1:
    statement1
    if condition2:
        statement2
    elif condition3:
        statement3
    else
      statement4
elif condition4:
    statement5
else:
    statement6
```

Loops:
While Loop

- Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

In [1]:
```
count = 0
while count < 5:
    print (count, " is  less than 5")
    count = count + 1
```

```
0  is  less than 5
1  is  less than 5
2  is  less than 5
3  is  less than 5
4  is  less than 5
```

Loops:
While Loop

- **Using *else* Statement with *while* Loop:**
  - If the ***else*** statement is used with a ***while*** loop, the ***else*** statement is executed when the condition becomes false.

```
In [2]: count = 0
        while count < 5:
            print (count, " is  less than 5")
            count = count + 1
        else:
            print (count, " is not less than 5")
```

```
0  is  less than 5
1  is  less than 5
2  is  less than 5
3  is  less than 5
4  is  less than 5
5  is not less than 5
```

Loops:
For Loop

- The ***for*** statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

- **Syntax**:

```
for iterating_var in sequence:
    statements(s)
```

Loops:
For Loop

- **Examples:**

```
In [1]: for var in range(5):
            print (var)
```

```
0
1
2
3
4
```

Loops:
For Loop

```
In [3]: for letter in 'Python':
            print ('Current Letter :', letter)
        print()
        fruits = ['banana', 'apple',  'mango']
        for fruit in fruits:
            print ('Current fruit :', fruit)
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

Current fruit : banana
Current fruit : apple
Current fruit : mango
```

Loops:
For Loop

- **Iterating by Sequence Index:**

```
In [4]: fruits = ['banana', 'apple',  'mango']
        for index in range(len(fruits)):
            print ('Current fruit :', fruits[index])
```

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
```

Loops:
For Loop

- **Using *else* Statement with *for* Loop:**
  - If the ***else*** statement is used with a **for** loop, the ***else*** block is executed only if for loops terminates normally **(and not by encountering break statement)**.

In [5]:
```
numbers = [11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
    if num%2 == 0:
        print ('the list contains an even number')
        break
else:
    print ('the list doesnot contain even number')
```

```
the list doesnot contain even number
```

Functions:
Syntax

- **Syntax:**

```
def functionname( parameters ):
    function_suite
    return [expression]★
```

★ *return statement is optional*

Functions:
Syntax

- **Example:**

```
def printme( str ):
    print (str)
```

```
printme("Welcom to Python Course")
```

```
Welcom to Python Course
```

Functions:
Pass by Reference Vs. Value

- **All parameters (arguments) in the Python language are passed by reference**

In [10]:
```
def changeme( mylist ):
    print ("Values inside the function before change: ", mylist)
    mylist[2]=50
    print ("Values inside the function after change: ", mylist)
```

In [11]:
```
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

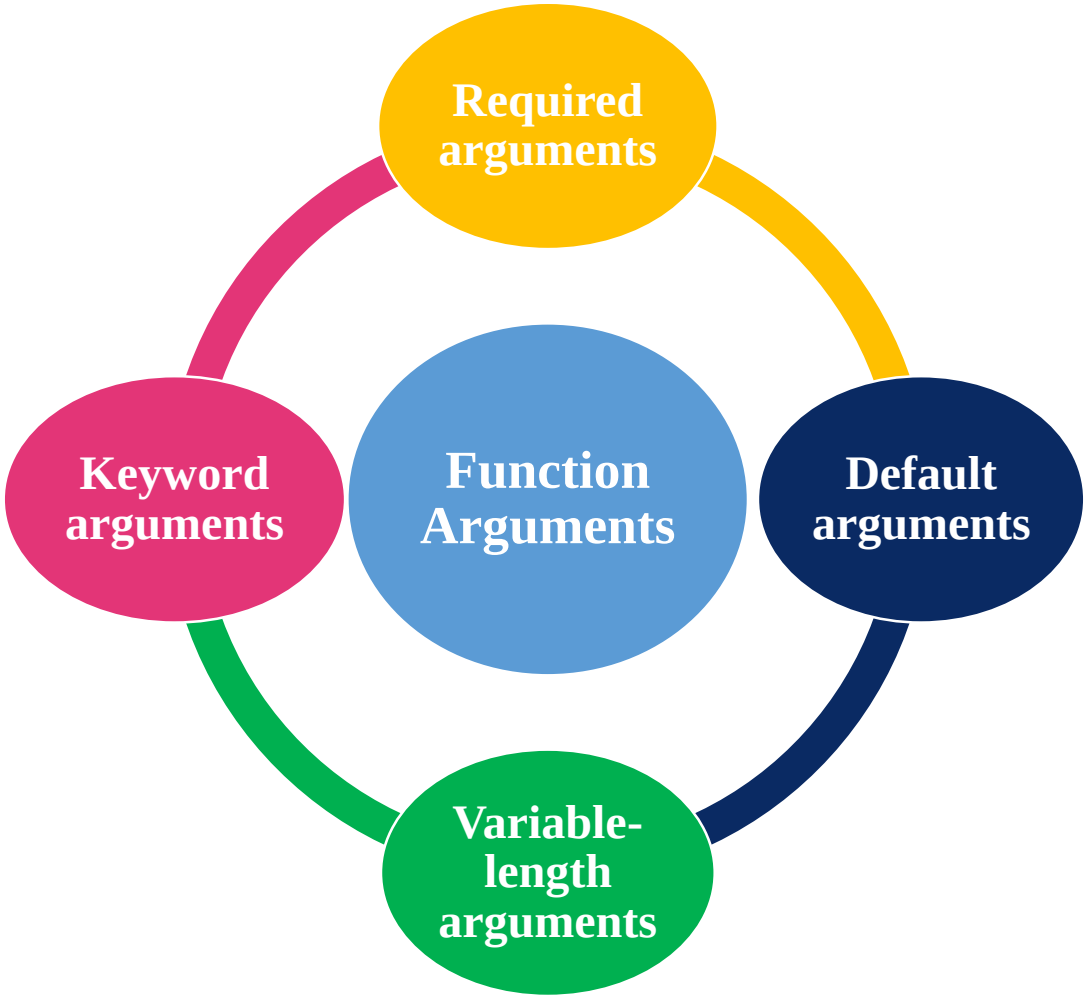
Functions:
Global vs. Local variables

- **Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.**

In [12]:
```
def sum( arg1, arg2 ):
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return total

# Now you can call sum function
total = 0
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

```
Inside the function local total :  30
Outside the function global total :  0
```

Functions:
Function Arguments



Functions:
Function Arguments

▪ Required Arguments:

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
def printme( str ):
   print (str)
```

```
printme("Welcom to Python Course")
```

```
Welcom to Python Course
```

Functions:
Function Arguments

- **Keyword Arguments:**
  - Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name

```
In [14]: def printinfo( name, age ):
             print ("Name: ", name)
             print ("Age ", age)
         # Function Call
         printinfo( age = 50, name = "miki" )
```

```
Name:  miki
Age  50
```

Functions:
Function Arguments

- **Default Arguments:**
  - A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

```
In [15]: def printinfo( name, age =35 ):
             print ("Name: ", name)
             print ("Age ", age)
         # Function Call
         printinfo( age = 50, name = "miki" )
         printinfo( name = "miki" )
```

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Functions:
Function Arguments

- **Variable-length Arguments:**
  - You may need to process a function for more arguments than you specified while defining the function.

```
In [16]: def printinfo( arg1, *vartuple ):
             print ("Output is: ")
             print (arg1)
             for var in vartuple:
                 print (var)
         # Now you can call printinfo function
         printinfo( 10 )
         printinfo( 70, 60, 50 )
```

```
Output is:
10
Output is:
70
60
50
```

Functions:
The Anonymous Functions

- The functions called anonymous because they are not declared in the standard manner by using the ***def*** keyword. You can use the ***lambda*** keyword to create anonymous function.

In [18]:
```
sum = lambda arg1, arg2: arg1 + arg2

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))
```

```
Value of total :  30
```

References

- https://docs.python.org/3.6/library/math.html
- https://docs.python.org/3.6/library/operator.html

Any Questions!?

Thank you