Group 30 : The python reference manual

Indrani,Vasavi,Bhavani,Srilekha

Compiler Design - CS335A

INDEX

1 Python Syntax:

- **indentation:**
  - It refers to the spaces at the beginning of a code line. Python uses indentation to indicate a block of code.
  - The number of spaces should be at least one and should be the same throughout a block of code, otherwise Python will give you an error.
  - This block structure with indentation is replacement of Flower brackets in C/C++.
  - example:

```
n=2
if n<=1:
    print("n<=1")
else:
    print("n>1")
#if and else are in the same line, print is written after space to indicate its inside the if-else block
```

- **Identifiers:**
  - Identifier is a name used for identifying variables, functions, new data types, etc.
  - The identifier is a combination of characters except special characters $\#, @, \$, \%, !$ in identifiers.
  - Rules for using Python Identifiers:
    - An identifier name must start with a letter or underscore character
    - An identifier name cannot start with a number
    - An identifier can contain alpha numeric characters and underscores
    - An identifier name is case sensitive
    - Reserved keywords cannot be used as identifier names.
- **Comments:**
  - Comments are the lines in the code that are ignored by the compiler during the execution of the program.
  - Single line Comments - Comments start with a #, and Python will render the rest of the line as a comment

```
a = 3
print(a)
#single line commments
```

  - Multiline Comments - Since Python will ignore string literals that are not assigned to a variable, we can add a multiline string (triple quotes) in our code as a multiline comment.

```
 """
This is a comment
written in
more than just one line
"""
a = "compilers"
print(a)
```

- **Keywords:**
  - **and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, True, try, while, with, yield.**

- These are a set of keywords that are reserved words that cannot be used as variable names, function names, or any other identifiers.

- **Declaration:**
  - Python is a dynamic-typed language, which means we don't need to mention the variable type or declare before using it.
  - It makes to Python the most efficient and easy to use language.
  - Every variable is treated as an object in Python.

2 Native Data types:

In python there are many predefined data types, from those we are going to implement native data types:

- **Numeric data types:** int , float , complex.
  - **int:** "int" class contains integers containing positive or negative whole numbers.
  - **float:** "float" class contains real numbers with floating point representation. It is specified by a decimal point.
  - **complex:** "complex" class contains complex numbers. It is specified as (real part) + (imaginary part)j.
  - To find the data type of a particular variable we can use inbuilt function **type**() :

```
x = 1
print(type(x)) #outputs <class 'int'>
y = 2.8
print(type(y)) #outputs <class 'float'>
z = 1+1j
print(type(z)) #outputs <class 'complex'>
```

  - There is no bound on the maximum or minimum value an integer/floating point/complex variable can hold. But we may implement only for a limited range of values

- **Sequence data types:** string , list , tuple.
  - **String:**
    * Strings in python are a sequence of Unicode characters. We can use single quotes, double quotes, or triple quotes to define a string literal.It is represented by str class.
    * Python does not have a character data type, a single character is simply a string with a length of 1.

```
Str1 = 'compilers'
print(type(Str1))  #ouputs <class 'str'>
Str2 = "a"
print(type(Str2))  #ouputs <class 'str'>
```

    * In Python,string characters are ordered that is they are indexed, the first item has index [0], the second item has index [1],Each character in a string has a negative index value. These values let us count backward from the end of a string. They start at -1.
    * While accessing an index out of the range will cause an IndexError. Only Integers are allowed to be passed as an index, float or other types that will cause a TypeError.

```
Str1 = 'compiler'
print(Str1[0]) #ouputs c
print(Str1[-1]) #outputs r
```

    * To access a range of characters in the String, the method of slicing is used. Slicing in a String is done by using a Slicing operator (colon :).

```
Str1 = "compiler"
print(Str1[2:7]) #outputs mpile
print(Str1[3:-2]) #outputs pil
```

* Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name and whole string can be updated and deleted using **del**

```
Str1 = "compiler"
del Str1
print(Str1) #gives error saying no such string exists
```

* **String Concatenation:** To concatenate, or combine, two strings you can use the + operator.

```
a = "Hello"
b = "World"
c = a + b
print(c) # prints HelloWorld
```

* The **strip()** method removes any whitespace from the beginning or the end

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

* **index():** The index() method returns the position at the first occurrence of the specified value.

```
s='helooworld'
print(l.index('o')) # outputs 3
```

* **find():** It finds the first occurrence of the specified value. The find() method returns -1 if the value is not found.

```
txt = "Hello, welcome to my world."
x = txt.find("e")
print(x)
```

* **join():** The join() method takes all items in an iterable and joins them into one string.A string must be specified as the separator.

```
#Syntax: string.join(iterable)
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
```

* **count():** The count method returns the number of elements with the specified value.

```
l=[1,1,2,4,7,6,7]
print(l.count('o')) # outputs 3
```

* The **split()** method returns a list where the text between the specified separator becomes the list items.

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

– **List:**

* Lists in python are a comma-separated values between square brackets.
* List items are indexed, the first item has index [0], the second item has index [1] etc.
* lists can be sliced.

* List items are ordered that is the items have a defined order and that order will not change.
* List items are changeable that means we can change, add, and remove items in a list after it is created.
* List allow duplicates as they are indexed.
* A list can contain different data types.
* Lists can be concatenated using '+' operator
* len() function gives the length of the list.
* We can have lists inside a list

```
nums =[1,2,3,9,4]
print(nums[0]) #outputs 1
print(nums[-1]) #outputs 4
print(nums[-3]) #outputs 3
print(nums[:]) #outputs [1,2,3,9,4]
print(nums[-3:]) #outputs [3,9,4]
print(nums[0:4]) #outputs [1,2,3,9]
mixed=["John", 1, True, 1.23]
len(nums) #outputs 5
s=[1,2]+[1,3]
print(s) #outputs [1,2,1,3]
nestedlist=[[1,2,'a'],[True, 'a', 1]]
print(nestedlist[0][2]) #outputs a,
print(nestedlist[1]) #outputs [True, 'a', 1]
```

* **append():** it inserts the value at the end of a list.

```
v=[1,2,4]
v.append(5)
print(v) #outputs [1,2,4,5]
```

* **insert():** it inserts the element at the given position.

```
v=[1,2,6,8,5]
v.insert(2,4)
print(v) #outputs [1,2,4,6,8,5]\\
```

* **pop():** it pops the element at the given index.

```
v=[1,2,4]
v.pop(1)
print(v) #ouputs [1,4]
```

* **reverse a list :** we can reverse a list using [::-1]

```
v=[1,2,3]
l=v[::-1]
print(l) #outputs [3,2,1]
```

* **index():** The index() method returns the position at the first occurrence of the specified value.

```
l=[1,1,2,4,7,6,7]
print(l.index(7)) # outputs 4
```

* **count:** The count method returns the number of elements with the specified value.

```
l=[1,1,2,4,7,6,7]
print(l.count(7)) # outputs 2
```

* **reverse:**The reverse() method reverses the sorting order of the elements.

```
v=[1,5,4,3]
v.reverse()
print(v) # outputs [3,4,5,1]
```

* **sort:**
  - The sort() method sorts the list ascending by default.
  - syntax: list.sort(reverse = True|False, key = myFunc).
  - (reverse = True) will sort the list in descending order and the default is (reverse = False).
  - A function to specify the sorting criteria(s) is given as key.
  - reverse parameter and key parameter are Optional.

```
k = ['e', 'c', 'w']
k.sort(reverse=True)
print(k) #outputs ['w','e','c']

def myFunc(e):
  return len(e)
fruits = ["bananas","apples", "cherries","grapes"]
fruits.sort(key=myFunc)
print(fruits) #outputs ['apples', 'grapes', 'bananas', 'cherries']
```

* With **list comprehension** we can declare a list with only one line of code. example:

```
doubles = [ 2 * x for x in range (5)]
print(doubles) # [2,4,6,8,10]
```

– **Tuple:**

* A Tuple is a collection of Python objects separated by commas that is they are used to store multiple items in a single variable.
* Tuples are created with the () syntax
* Empty tuples are constructed by an empty pair of parentheses
* A tuple with one item is constructed by following a value with a comma
* Tuple items are ordered that is tuple items are indexed, the first item has index [0], the second item has index [1],Each item in a tuple has a negative index value. These values let us count backward from the end of a tuple. They start at -1.
* Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
* Tuples allow duplicate values because tuples are indexed, they can have items with the same value
* Tuple slicing: To get range of items within our tuple, we can specify a range of indexes to retrieve. In order to do so, we need to specify where to start and end our range.
* To reverse a tuple we use t[::-1] We can concatenate (merge) or multiply the contents of a tuple, syntax : + operator can be used to concatenate two or more together.
* To determine how many items a tuple has, use the **len**() function
* Nesting of Tuples can be done

```
t = (12345, 54321, 'hello!')
print(t)
empty_tuple = ()
print(empty_tuple) #prints nothing
singleton = (1,)
print(singleton) #outputs (1)
t = ('hello',1,2,4)
print(t[0]) #outputs hello
```

```
print(t[1]) #outputs 1
print(t[-1]) #outputs 4
print(t[-2]) #outputs 2
t = ('hello',1,2,4,7)
print(t[::-1]) #outputs (7,4,2,1,'hello')
print(t[1:4]) #outputs (1,2,4)
print(t[:2]) #outputs('hello',1)
print(t[-2:]) #outputs(4,7)
t1 = ('hello',1,True)
t2 = (1,'a',3)
t3 = t1 + t2
print(t3) #outputs ('hello',1,True,1,'a',3)
t4 = (t1,t2)
print(t4) #outputs (('hello',1,True),(1,'a',3))
len(t1) #outputs 3
```

* **count():** The count method returns the number of elements with the specified value.

```
t=(1,1,2,4,7,6,7)
print(t.count(7)) # outputs 2
```

* **index():** The index() method returns the position at the first occurrence of the specified value.

```
t=(1,1,2,4,7,6,7)
print(t.index(7)) # outputs 4
```

- **Boolean data type:**
  - **bool:** "bool" class contains two built-in values, **True** and **False**.
  - It is a data type used to represent truth values of expressions.
  - The **bool**() function allows you to evaluate any value, and give you True or False in return.
  - Boolean objects that are equal to True are truthy, and those equal to False are falsy.
  - But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the class "bool".

```
print(type(True)) #outputs "bool"
print(type(False)) #outputs "bool"
print(10>9) #outputs "True"
print(10<9) #outputs "False"
print(bool("Hello")) #outputs "True"
print(bool(-1)) #outputs "True"
print(bool(0)) #outputs "False"
```

Note: True and False with capital 'T' and 'F' are valid booleans otherwise python will throw an error.

3 Variables and Expressions:

- **Variables :**
  There is no command for declaring a variable, We can directly assign the values to the variables for declaration, and can even change type after they have been set.
  Variable names are case-sensitive.
  If we want to specify data type of a variable, we can use casting.
  x=**str**(3) then x='3' and type of x is string.
  - **global variables**
    * Variables that are created outside of a function are known as global variables.

* Global variables can be used by everywhere, both inside of functions and outside.
* If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function.
* The global variable with the same name will remain as it was, global and with the original value.
example:

```
def func():
    v="wow"
    print(s) #it prints hello
    print(v)# it prints wow
s="hello"
func()
```

Here, func, s are a global variables.

- **local variables:**
  * Local variables are those which are initialized inside a function and belongs only to that particular function.
  * It cannot be accessed anywhere outside the function. In the above example, v is a local variable.
  * To change a global variable value inside a function we should declare it as global in that scope otherwise even if the variable has same name it is treated as a local variable and changes are not reflected globally.
  example:

```
def func1():
    global s
    s="helloworld"
func1()
print(s) # it prints helloworld
```

- **Operators:**
  - **Assignment operators**:
    * '+' , '-', '*', '/' have an usual definitions and '%' is modulo.
    * '//' is called integer division and '**' is exponentation operation.
  - **Assignment operators**:
    * Assignment operators are used to assign values to variables.
  - **comparision operators**:
    * $==, <=, >=, >, <, != $ have their usual meanings
  - **Logical Operators**: Logical operators are used to combine conditional statements
    * **and** Returns True if both statements are true
    example: if x=3, x < 5 **and** x < 10 , the statement returns true
    * **or** Returns True if one of the statements is true
    example: if x=3, x < 5 **or** x < 10 , the statement returns true
    * **not** Reverse the result, returns False if the result is true
    example: if x=3, not(x < 5 **and** x < 10) , the statement returns False.
  - **Identity Operators**:Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:
    * **is** Returns True if both variables are the same object
    If x = 2 and y = 3 then x **is** y gives False
    * **is not** Returns True if both variables are not the same object
    If x = 2 and y = 3 then x **is not** y gives True
  - **Membership Operators**:

    - **in** : Returns True if a sequence with the specified value is present in the object
    - **not in**: Returns True if a sequence with the specified value is not present in the object
  - Bitwise Operators:
    - & - AND - Sets each bit to 1 if both bits are 1
    - | - OR - Sets each bit to 1 if one of two bits is 1
    - ^ - XOR - Sets each bit to 1 if only one of two bits is 1
    - ∼ - NOT - Inverts all the bits
    - << - Zero fill left shift - Shift left by pushing zeros in from the right and let the leftmost bits fall off
    - >> - Signed right shift - Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

- **Expressions:**
  - **Constant expressions:**
    - These are the expressions that have constant values only.
      example: x = 18 + 1.4
  - **Arithmetic expressions:**
    - An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis.
    - The result of this type of expression is also a numeric value
    - some **Arthematic Operations:**
      - Preference order and expression evaluation is same as that of C/C++.

```
# Arithmetic Expressions
x = 40
y = 12
add = x + y
print(add) #52
```

  - **Integral Expressions:** These are the kind of expressions that produce only integer results after all computations and type conversions.

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

  - **Floating Expressions:** These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

```
# Floating Expressions
a = 13
b = 5
c = a / b
print(c)
```

  - **Relational Expressions or Boolean expressions:**
    - In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < ,>= , <=).
    - Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end.

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

- **Logical Expressions**:
  - These are kinds of expressions that result in either True or False.
  - we also come across some logical operators which can be seen in logical expressions most often

```
P = (0 == 8)
Q = (3 > 5)
# Logical Expressions
R = P and Q
S = P or Q
T = not P
print(R)
print(S)
print(T)
```

- **Bitwise Expressions:**
  - These are the kind of expressions in which computations are performed at bit level.

```
# Bitwise Expressions
a = 12
x = a >> 2
y = a << 1
print(x, y)
```

- **Combinational Expressions:**
  - We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.

```
# Combinational Expressions
a = 16
b = 12

c = a + (b >> 1)
print(c)
```

when there are multiple operators in expressions we use precedence order to evaluate the expression.

4 Control structures:

- **if :**
  The if statement is used for conditional execution.

```
if condition1:
    #statement1
```

example:

```
a=10
if a<20:
   print("a<20") #it prints a<20
```

- **if-then-else :**

```
if condition1:
    statement1
elif condition2:
    statement2
else:
    some other statement3
```

example:

```
a=20
if a>=45:
    print("a is greater or equals to  45")
elif 25<=a<=45:
    print("a lies between 25 and 45")
else:
    print("a is less than 25")
```

output: a is less than 25

  - The expression and if can be separated using a space or using parenthesis.
  - If statement1 evalues to true, then statement1 is executed and elif is not executed. If statement1 evalues to false then statement2 is executed and so on.

- **else Statement**: With the else statement we can run a block of code once when the condition no longer is true

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
#it prints 123456is no longer less than 6
```

- **match case:**
this is simlar to switch case in C/C++

```
match value:
    case val1 :
       # statement 1 as value is equal to val1
    case val2 :
       # statement 2 as value is equal to val2
    case val3 :
        #statement 3 as value is equal to val3
    case _ :
       # default statement
```

- **break:**

  - The break statement breaks out of the innermost enclosing loop and transfers execution to the statement immediately following the loop.

- **continue**:
  The continue statement, also borrowed from C, continues with the next iteration of the loop
- **return**:
  A return statement is used to end the execution of the function call and "returns" the result.
- **pass**:
  The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.
  example :
  while True:
  pass

5 Loops:

- **While** loops consist of a single conditional statement. While this statement is true, the body of the loop will be continually executed.

```
while condition:
    #sequence of statements
```

- **for statement**
  It is used to iterate over the elements of a sequence like list,string or tuple.

```
for val in valuesList:
    #loop statements
```

  Here var is the variable that takes the value of the item inside the list valuesList in each iteration. And the loop continues until we reach the last value in the sequence.

- **Nested Loops:** A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop"

```
adj = ["red", "big"]
fruits = ["apple","orange"]
for x in adj:
  for y in fruits:
    print(x,y)
```

6 Functions:

We define a function using the keyword **def** followed the function name.

```
# params refer to the parameters that we pass to a function.
def functionName(params) :
    #statements
```

The keyword **def** marks the start of the function header. functionName is used to uniquely identify the function. Params are the arguments through which we pass values to a function and are optional.
A colon(:) marks the end of function header.
example:

```
def my_func():
        x = 10
        print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

A **return** statement is used to return value from the function which is optional too. We can define nested functions in the same way.
Call a function using function name, functionName(params)

7 Class:

We define a class with a keyword class followed by the class name.
send the params if any using parenthesis else we can skip parenthesis and only write the name

```
class ClassName(params):
      #some statements
```

An Object is an instance of a Class. Class objects support two kinds of operations: attribute references and instantiation. valid attribute names are all the names that were in the class's namespace when the class object was created.
example:

```
class MyClass:
    val='1234'
    def f(self):
        return 'hello'
```

for the above class MyClass.val , MyClass.f are the valid attribute references.
We can instantiate a class as below:

```
x=ClassName() # pretending class object is parameterless
```

Many classes like to create objects with instances customized to a specific initial state.Therefore a class may define a special method named __init__().
__init__() function :
All classes have this function and get executed when the class is initiated.
self parameter:
It is a reference to the current instance of the class and it is used to access variables that belongs to that class.
It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class somename:
    value=5
    def __init__(self,val1):
        self.val1=val1
        self.val='hello'
    def addSum(v,val2): # Here v is self parameter
        return v.val1+val2+v.value
x=somename(2)
print(x.addSum(4))
```

instance objects
The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods. for a instance we can define data attributes and also methods as shown below. They are accessible to that particular instance obj and not by other instances of the class. example: for the above instance of the class someone we now define another data attribute extraval

```
def withExtraAttributesSum(att):
    return att.val1+att.extraval
x.extraval=3
print(withExtraAttributesSum(x))
```

The above code output 5.

```
y=somename(7)
print(withExtraAttributesSum(y))
```

It gives error because extraval attribute is defined only for the instance x and it is not attribute of class. so y has no attribute extraval so y.extraval is not defined so, it raises an exception.
The other kind of instance attribute reference is method. A method is a function that belongs to an object. working with function attributes is also same as that working with data attributes.
Method Objects:

```
xadd=x.addSum
print(xadd(5)) # this outputs 9
```

here x.addSum is the method object

8 Input and Output in Python:

Python provides us with the two inbuilt functions input() and print() for input and output respectively.
The syntax for input is input(Prompt)
the python will automatically identify whether the user entered a string, number, or list
Python takes all the input as a string input by default. To convert it to any other data type we have to convert the input explicitly. For example, to convert the input to int or float we have to use the int() and float() method respectively.
if the input entered from the user is not correct, then python will throw a syntax error.
And the syntax for the output in python is print(variable/"string"), normally used to print the output.
Example for input and output :

```
name = input()
#input given was "python"
print(name) #prints python as output
```

File input and output:
Python provides basic functions and methods necessary to manipulate files by default. You can do most of the file manipulation using a file object.

- open a file(syntax): FileObject = open(filename)

```
#if the file is in the same directory
f = open("File.txt")
#if the file is in a different directory
f = open("C:/users/Python/File.txt")
```

- Reading data from the File(syntax): FileObject = open("File.txt", 'r') opening the file in reading mode

```
#To print the content of the whole file
print(f.read())
#To read only one line
print(f.readline())
```

- Writing Data to File(syntax): FileObject = open("File.txt", 'w') opening the file in write mode.

```
f.write("Hello Python \n")
#in the above code '\n' is next line which means in the text file it
#will write Hello. Python and point the cursor to the next line
f.write("Hello World")
```

- closing a file(syntax):When we close the file, it will free up the resources that were tied with the file.

```
f = open("File.txt", 'r')
print (f.read())
f.close()
```

9 References:

A Python program accesses data values through references. A reference is a name that refers to the specific location in memory of a value (object). References take the form of variables, attributes, and items. In Python, a variable or other reference has no intrinsic type. The object to which a reference is bound at a given time does have a type, however. Any given reference may be bound to objects of different types during the execution of a program.

A variable references an object that holds a value. In other words, variables are references.The following example assigns a number with a value of 100 to a variable **counter** = 100.
In the above example, Python creates a new integer object (int) in the memory and binds the counter variable to that memory address. When you access the counter variable, Python looks up the object referenced by the counter and returns the value of that object.So, variables are references that point to objects in the memory.

To find the memory address of an object referenced by a variable, you pass the variable to the built-in id() function.For example, the following returns the memory address of the integer object referenced by the counter variable:

```
counter = 100
print(id(counter))
```

The id() function returns the memory address of an object referenced by a variable as a base-10 number.
An object in the memory address can have one or more references.For example,

```
counter = 100
```

The integer object with the value of 100 has one reference which is the counter variable. If you assign the counter to another variable e.g., max:

```
counter = 100
maximum = counter
```

Now, both counter and maximum variables reference the same integer object. The integer object with value 100 has two references:
If you assign a different value to the maximum variable, the integer object with value 100 has one reference, which is the counter variable.And the number of references of the integer object with a value of 100 will be zero if you assign a different value to the counter variable.

10 Inbuilt functions:

- **abs()**
  - It returns the absolute value of a specified number
    example: abs(-2) is 2, abs(3+5j) is 5.830951894845301
- **bin()**
  - The bin() function returns the binary version of a specified integer.
  - The result will always start with the prefix 0b syntax: bin(n)
    example: bin(8) is 0b1000
- **bool()**

- The bool() function returns the boolean value of a specified object.
- syntax: bool(obj), obj can be list, string,object etc.
- The object will always return True, unless: it is empty, like [], (), or False or 0 or None

- **chr()**
  - The chr() function returns the character that represents the specified unicode.
  - syntax: chr(n) where n represents a valid Unicode code point.
    example: chr(99) is c

- **delattr()**
  - The delattr() function will delete the specified attribute from the specified object.
  - syntax: delattr(obj,attribute) where obj is the object and name of the attribute to be removed.
    example:

```
class Person:
    name="Hormonie"
    age="18"
delattr(Person,age)
```

- **hasattr()**
  - The hasattr() function returns True if the specified object has the specified attribute, otherwise False
  - syntax: hasattr(obj,attribute) where obj is the object and name of the attribute that is to be checked.
    example:

```
print(hasattr(Person,age)) # prints True
```

- **getattr:**
  - The getattr() function returns the value of the specified attribute from the specified object.
  - syntax: getattr(obj,attribute,default) where obj is the object and name of the attribute and default is a optional parameter the value to return if the attribute does not exist.
    example:

```
print(getattr(Person,age)) # prints 18
```

- **hex()**
  - The hex() function converts the specified number into a hexadecimal value.
    example:

```
x= hex(240)
print(x) #it prints 0xf0
```

- **iter()**
  - The iter() function returns an iterator object.
    example:

```
x = iter(["apple", "banana", "cherry"])
print(next(x)) # apple
print(next(x)) # banana
print(next(x)) # cherry
```

- **map()**
  - The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.
    example:

```
def myconsFunc(a):
    return a+2
x = map(myfunc, [1,2,3])
print(list(x)) # [3,4,5]
```

- **max()**
  - The max() function returns the item with the highest value, or the item with the highest value in an iterable.
    example:

```
print(max(2,3)) # it prints 3
l=[1,4,8,3]
print(max(l)) # prints 8
```

- **min()**
  - The max() function returns the item with the lowest value, or the item with the highest value in an iterable.
    example:

```
print(min(2,3)) # it prints 3
l=[1,4,8,3]
print(max(l)) # prints 1
```

- **oct():**
  - The oct() function converts the specified number into a octal string.
    example:

```
x= oct(240)
print(x) #it prints 0o360
```

- **ord():**
  - The ord() function returns the number representing the unicode code of a specified character.
    example:

```
print(ord(a))  #prints  97
```

- **pow():**
  - The pow() function returns the value of x to the power of y (x^y).
  - If a third parameter is present, it returns x to the power of y, modulus z.
    example:

```
x = pow(4, 3, 5) # the value x is 64%5=4
```

- **reversed():**

- The reversed() function returns a reversed iterator object.
- syntax: reversed(sequence) , where sequence is any iterable object.
  example:

```
l=[1,2,3,4]
v=reversed(l)
print(v) #  it prints [4,3,2,1]
```

- **range()**:
  we use the function to iterate over a sequence of numbers**range**(start,end,step)# its the more precise way of range function it iterates over from start to end-1 with the given step **range**(start,end) # in this case it iterates over from start to end-1 with step=1 **range**(l) # in this case it iterates over from 0 to l-1 with step=1

- **round():**
  - The round() function returns a floating point number that is a rounded version of the specified number, with the specified number of decimals.
  - The default number of decimals is 0, meaning that the function will return the nearest integer.
  - **syntax:** round(number, digits).
  - Here, the number is the number to be rounded. And digits are the number of decimals to be used while rounding the number. Default value is 0.
    example:

```
x = round(5.76543, 2)
print(x)
```

    output: 5.77

- **sorted():**
  - The sorted() function returns a sorted list of the specified iterable object.
  - You can specify ascending or descending order. Strings are sorted alphabetically, and numbers are sorted numerically.
  - **syntax:** sorted(iterable, key=key, reverse=reverse)
  - Iterable is the sequence to sort, list, dictionary, tuple etc.
  - Key is optional and it is a function to execute to decide the order. Default is None.
  - Reverse is a Boolean. False will sort ascending, True will sort descending. Default is False. It is also optional.
    example:

```
a = ("b", "g", "a", "d", "f", "c", "h", "e")
x = sorted(a)
print(x)
```

    output: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'].

- **sum():**
  - The sum() function returns a number, the sum of all items in an iterable.
  - **syntax:** sum(iterable, start)
  - Iterable is the sequence of which sum is calculated and start is the value that is added to the return value.
    example:

```
a = (1, 2, 3, 4, 5)
x = sum(a)
print(x)
```

output: 15(which is sum of 1,2,3,4,5).

- **zip():**
  - The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
  - If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.
  - **syntax:** zip(iterator1, iterator2, iterator3 ...)
    example:

```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica")
x = zip(a, b)
print(tuple(x))
```

output: (('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica'))

References:

https://www.w3schools.com/python/python_datatypes.asp
http://metagenome.cs.umn.edu/pubs/2015_Meulemans_Hydrocarbon_Lipid_Microbiology_Protocols.pdf
https://docs.python.org/3/tutorial/classes.html#class-definition-syntax
https://www.geeksforgeeks.org/python-programming-language/
https://www.softwaretestinghelp.com/python/input-output-python-files/