**Parameters and Arguments**

This function assigns the argument to a parameter named value. When the function is called, it assigns the value to x and then prints the value of the parameter (whatever it is).

The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for print\_name:

**print\_name(str(math.cos(0)))**

The argument is evaluated before the function is called, so in the above example, str(math.cos(0)) is evaluated only once.

**Passing arguments as args and kwargs**

We can pass any number of arguments to a function using special symbols. There are two ways to do this:

* Non-keyworded Arguments -> Represented as \*args
* Keyworded Arguments -> Represented as \*\*kwargs

These ways are used when we don't know how many arguments we might need to pass to the function. These types of arguments can appear before or after the fixed or known arguments in the function definition. If known arguments are appearing after the args, then while calling the function, you will need to pass the fixed parameters by keyword.

**Passing as \*args**

# It allows passing zero or more arguments which can be iterated over inside the function. The syntax is to define the args with a single star. Example: Below function multiplies all numbers passed as arguments and returns the output suffixed with a pre-defined message passed to the function.

**def multiply(message, \*args):**

**output = 1**

**for arg in args:**

**output \*= arg**

**return message + " " + str(output)**

**multiply\_value = multiply("My Sum:", 1, 2, 4, 6)**

**print(multiply\_value)**

Another variation:

**def multiply(\*args, message):**

**output = 1**

**for arg in args:**

**output \*= arg**

**return message + " " + str(output)**

**multiply\_value = multiply(1, 2, 4, 6, message="My Sum:")**

**print(multiply\_value)**

**Passing as \*\*kwargs**

It allows passing zero or more keyworded arguments which can be iterated over inside the function. The syntax is to define the args with two stars. Example: Below function sums up the count of each type of fruit from the passed arguments. Each argument is key-value pair of fruit type and count.

**def groupdata(message, \*\*kwargs):**

**grouped\_data = {}**

**for key, value in kwargs.items():**

**grouped\_data[key.split("\_")[0]] = grouped\_data.get(key.split("\_")[0], 0) + value**

**return message + " " + str(grouped\_data)**

**summary = groupdata( "My fruits:", apple\_indian = 3, orange\_indian = 5, apple\_mexican = 6,orange\_mexica = 7, banana=11)**

**print(summary)**

# Boolean Expressions

The == operator is one of the comparison operators; the others are:

x != y # x is not equal to y

x > y # x is greater than y

x < y # x is less than y

x >= y # x is greater than or equal to y

x <= y # x is less than or equal to y

x is y # x is the same as y

x is not y # x is not the same as y

# Logical Operators

There are 3 logical operators in Python: and, or, and not. The meaning of these operators is similar to their meaning in English.

**3 < 4 and 4 > 1**

**It returns True**

**x % 2 == 0 or x % 4 == 0**

It returns True if either of the conditions is true, i.e. if the number x is divisible by 2 or 4.

The not operator negates a boolean expression, so,

**not (a > b )**

It returns True if a > b is false i.e. if a is less than or equal to b.

# String methods

An str object just like other objects in Python has 2 components-  
1) data (string itself), and  
2) methods associated with it i.e. built-in functions associated with any instance of the object.

**a = "cloudxlab"**

**dir(a)**

The dir function lists the methods associated with the object. Try it in the notebook.

To get information regarding any method, you can use the help function.

**help(a.translate)**

It provides simple documentation on the translate method of str object. You can also write as,

**help(str.translate)**

We call a method just like the function call but here the variable or the instance of the object is followed by a . and then the method name. For eg,

**print(a.upper())**

**List Methods**

Just like strings, you can check for methods for lists as well using dir and help. Some important methods are like append, extend and sort.

* append adds a new element to the list,
* extend takes a list as an argument and appends all of the elements,
* sort arranges the elements of the list from small to large.
* sum() only works when the list elements are numbers to calculate the sum of all elements
* other functions (max(), len(), etc.) work with lists of strings and other types that can be comparable.

**Deleting Elements of List**

As we know that lists are mutable, we can delete elements within it. There are several ways to delete elements from a list. Like, if we know the index of the element to be deleted, we can use pop function,

**list1 = [12,23,43,[2,4,5], 2, 4,5]**

**list1.pop(3)**

It deletes the element at index 3.

pop modifies the list and returns the element that was removed. If we don't provide an index, it deletes and returns the last element.

If we do not want the deleted value, we can use the del operator which deletes the element but doesn't return anything.

list2 = ['cloudx', 'lab', 'provides', 'cloud', 'lab']

**del list2[1]**

**print(list2)**

It prints ['cloudx', 'provides', 'cloud', 'lab']

And, if we know the element that needs to be removed, we can use remove,

**list2.remove('lab')**

The return value from remove is None.

The list function **splits** a string into individual letters. Suppose, if we want to break a long string into words, we can use split function to create a list of words,

**s = "I am learning Python at CloudxLab"**

**sp = s.split()**

**print (sp)**

It prints the list of words in s as ['I', 'am', 'learning', 'Python', 'at', 'CloudxLab']

We can also split the string on the basis of a certain character or a substring called as a delimiter. It is an optional argument for the split function.

**Working with Dictionaries**

Dictionaries have a method called get that takes a key and a default value as arguments. If the key is found in the dictionary, get returns the corresponding value, else returns the default value.

**d = {"apples" : 2, "bananas" : 3, "carrots" : 12}**

**print(d.get("oranges", 0))**

It prints 0, because "oranges" isn't a key available and 0 is the default value.

If you use a for loop to traverse in the dictionary, it traverses over the keys and using which you can iterate over the values as well.

**for fruit in d:**

**print(fruit)**

It prints the keys present in d.

**Tuples**

Tuple is also a sequence of values much like a list. The values stored in a tuple can be of any type, and they are indexed by integers.

Tuples are **immutable** but also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

**t = (12,323, 'd', [1,23], False)**

Here t is a tuple with values of different data types as int, str, list and bool. Each value is separated by a comma.

Even if a tuple has a single element, we need to mention the comma,

t = (1,)

type(t)

We can also create a tuple using the function tuple. With no argument, it creates an empty tuple:

**t = tuple()**

**print(t)**

Comparison with Tuples

Comparisons work with sequences. So, it works well with the tuples.

Python compares the tuples element-wise starting with the 1st element. If they turn out to be equal, it proceeds to the next one. If it finds any difference in elements, it gives the result without considering the further elements.

print((0, 1, 2) < (0, 3, 4))

It prints True. It doesn't depend on the number of elements in each tuple. It is only concerned with the two elements that it is comparing at one time.

sort function also works the same way by comparing elements inside each tuple when given a list of tuples.

l = [(0,23,34), (2,34,23), (1,34,23)]

**l.sort()**

If you want to do the sorting in reverse order i.e. from large to small, we can add an argument like this,

**l.sort(reverse=True)**

Tuple Assignment

You can have tuple on the left-hand side of the assignment as well.

**l = [1,2,3]**

**(a,b,c) = l**

Here we assigned a list with values 1, 2 and 3. And, then we assign the list to a tuple containing 3 elements. As a result, a gets assigned with l[0], b with l[1] and c with l[2].

We can also write it without any parenthesis (brackets) and it is equally valid like this,

**l = [1,2,3]**

**a, b, c = l**

It helps us in swapping elements in a pretty way,

a , b = b, a

Both sides of this statement are tuples, but the left side is a tuple of variables. The right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side.

All the expressions on the right side are evaluated before any of the assignments.

**The number of variables on the left and the number of values on the right must be the same.**

If you are wondering why we used the only list on the right-hand side here. So, for your information, the right side can be any kind of sequence (string, list, or tuple).

# Tuples and Dictionaries

There is a function with the name **items** associated with dictionaries that returns a list of tuples, where each tuple is a key-value pair:

**d = { "one" : 1, "two" : 2, "three" : 3}**

**k = d.items()**

**print(k)**

**It prints dict\_items([('one', 1), ('two', 2), ('three', 3)]). We can cast to list using list,**

**l = list(k)**

**print(l)**

It prints a list of tuples [('one', 1), ('two', 2), ('three', 3)].

Since it is a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key.

We can print the content of the dictionary with below code

**for key, value in list(k):**

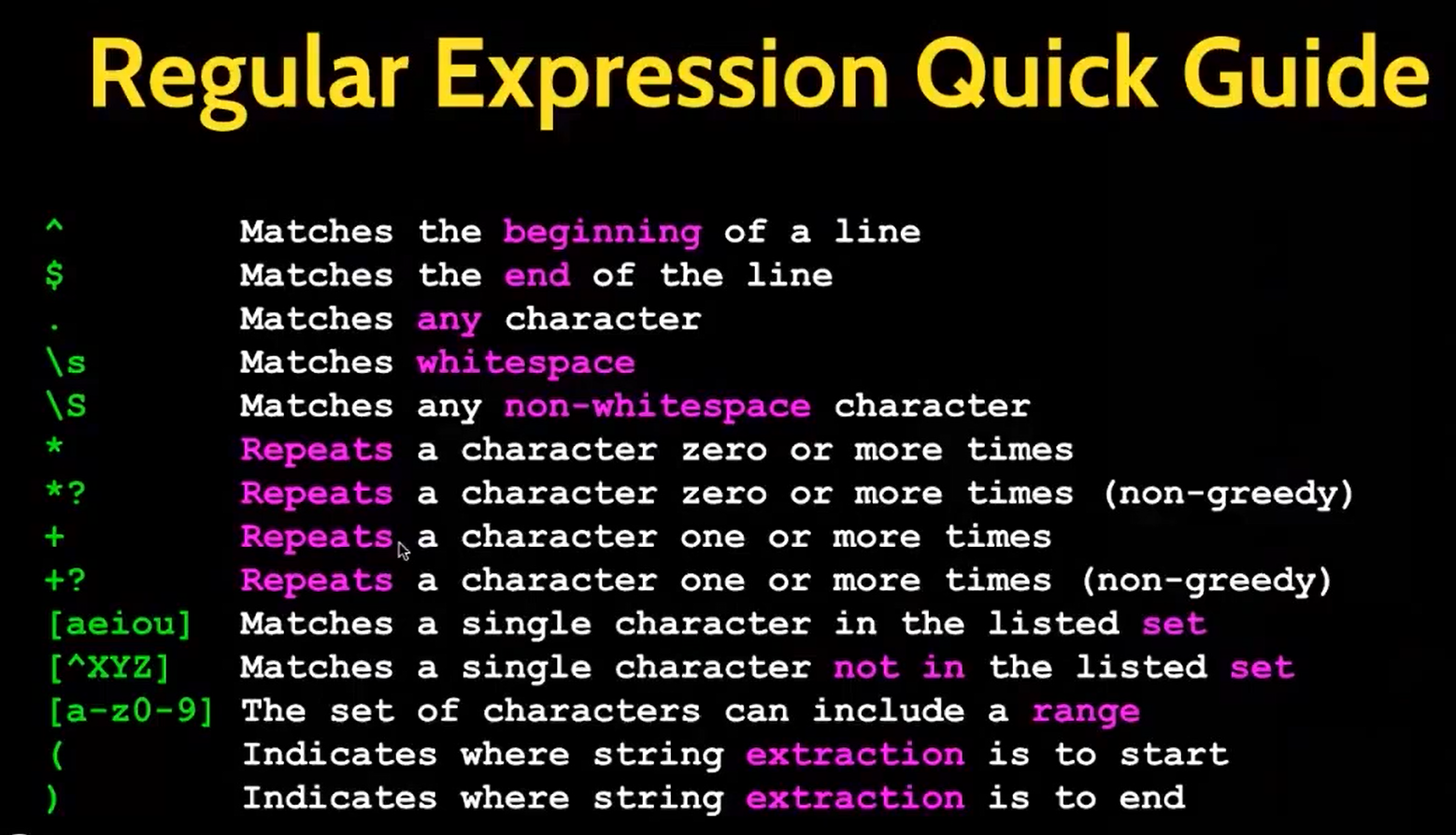
**print(key, value)**

**It prints,**

**one 1**

**two 2**

**three 3**



**A Lamba function**

A lambda function is a small anonymous function and it is defined by the keyword "lambda".

A lambda function can take any number of arguments, but can only have one expression.

A lambda function is defined and used in the same statement.

Syntax:

**lambda arguments: expression**

Examples:

Increment a number by 10.

**x = lambda a : a + 10**

**print(x(5))**

Multiply two arguments return the result:

**x = lambda a, b : a \* b**

**print(x(3, 4))**

The concept of lambda can be leveraged further by using this inside a normal function. In the below example, a generic function is created which acts a doubler, tripler or quadrupler or any multiplier.

**def multiplier(n):**

**return lambda a : a \* n**

**doubler = multiplier(2)**

**print(doubler(11))**

**tripler= multiplier(3)**

**print(tripler(11))**

**# or can do this directly**

**print(multiplier(4)(11))**

You can also do operations like filter, map, etc using lambda which can also be using normal functions.

In the example below, we will filter the numbers divisible by 3 using different methods.

**Method 1: Using a normal function**

**foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]**

**def filter\_divisbile\_by\_n(n, numlist):**

**return [num for num in numlist if num%n==0]**

**divisible\_by\_3 = filter\_divisbile\_by\_n(3, foo)**

**print("Method 1 output is:")**

**print(divisible\_by\_3, "\n")**

**Method 2: Using a function as an argument**

**def is\_divisbile\_by\_n(n, num):**

**return num%n==0**

**def filter\_divisbile\_by\_n(is\_divisbile\_by\_n, n, numlist):**

**return [num for num in numlist if is\_divisbile\_by\_n(n, num)]**

**divisible\_by\_3 = filter\_divisbile\_by\_n(is\_divisbile\_by\_n, 3, foo)**

**print("Method 2 output is:")**

**print(divisible\_by\_3, "\n")**

**Method 3: Using a lambda as an argument**

**def filter\_divisbile\_by\_n(div\_func, numlist):**

**return [num for num in numlist if div\_func(num)]**

**divisible\_by\_3 = filter\_divisbile\_by\_n(lambda x: x%3 == 0, foo)**

print("Method 3 output is:")

print(divisible\_by\_3, "\n")

**Method 4: Using a filter and normal function**

**def is\_divisbile\_by\_3(num):**

**return num%3==0**

**divisible\_by\_3 = filter(is\_divisbile\_by\_3, foo)**

**print("Method 4 output is:")**

**print(list(divisible\_by\_3),"\n")**

**Method 5: Using a filter and lambda function**

**divisible\_by\_3 = filter(lambda x: x%3 == 0, foo)**

**print("Method 5 output is:")**

**print(list(divisible\_by\_3),"\n")**

**Method 6: Using a lambda inside a function**

def is\_divisbile\_by\_n(n):

return lambda num: num%n == 0

divisible\_by\_3 = filter(is\_divisbile\_by\_n(3), foo)

print("Method 6 output is:")

print(list(divisible\_by\_3),"\n")

**Using lambda with map** In the below example, we double the value of each element.

doubled\_list = map(lambda x: x\*2, foo) print(list(doubled\_list))

**Using lambda with reduce** In the below example, we sum up all elements. from functools import reduce summed\_list = reduce(lambda x, y: x + y, foo) print(summed\_list)

**Itertool**

#lambda

f= lambda x,y:x+y

foo=[2,18,9,22,24,8,12]

**# filter() function**

division\_by\_3= filter(lambda x:x%3==0,foo)

print(list(division\_by\_3))

**#map () function**

lambda\_map= map(lambda x:x\*2+10, foo)

print(list(lambda\_map))

**# reduce() function**

import functools

lambda\_reduce = functools.reduce(lambda x,y:x+y, foo)

print (lambda\_reduce)