**if, elif, else Statements**

if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with elif and else statements, which allow us to tell the computer:

"if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action."

Let's go ahead and look at the syntax format for if statements to get a better idea of this:

if case1:

perform action1

elif case2:

perform action2

else:

perform action3

Type *Markdown* and LaTeX: 𝛼2α2

**if** **True**:

*#here value of True is 1 hence gets executed*

print('It was true!')

It was true!

Type *Markdown* and LaTeX: 𝛼2α2

x **=** **False** *#here false is 0 hence else part will get executed*

​

**if** x:

print('x was True!')

**else**:

print('I will be printed in any case where x is not true')

I will be printed in any case where x is not true

**Multiple Branches**

if, elif, and else can take us!

We write this out in a nested structure. Take note of how the if, elif, and else line up in the code. This can help you see what if is related to what elif or else statements.

a comparison syntax for Python.

loc **=** 'Bank'

​

**if** loc **==** 'Auto Shop':

print('Welcome to the Auto Shop!')

**elif** loc **==** 'Bank':

print('Welcome to the bank!')

**else**:

print('Where are you?')

Welcome to the bank!

Note how the nested if statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many elif statements as you want before you close off with an else.

Let's create two more simple examples for the if, elif, and else statements:

person **=** 'Sammy'

​

**if** person **==** 'Sammy':

print('Welcome Sammy!')

**else**:

print("Welcome, what's your name?")

Welcome Sammy!

person **=** 'George'

​

**if** person **==** 'Sammy':

print('Welcome Sammy!')

​

​

**elif** person **==**'George':

print('Welcome George!')

**else**:

print("Welcome, what's your name?")

Welcome George!

​

​

Type *Markdown* and LaTeX: 𝛼2α2

# for Loops

A for loop acts as an iterator in Python; it goes through items that are in a sequence or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

Here's the general format for a for loop in Python:

for item in object:

statements to do stuff

​

​

lst **=** [30,23,10,16]

**for** item **in** lst:

print(item)

​

30

23

10

16

lst **=** [1,4,7,8,9]

​

**for** var **in** range(10):

print(var)

0

2

4

6

8

​

​

​

lst **=** [10,20,30]

**for** index,item **in** enumerate(lst):

print(index,item)

​

0 10

1 20

2 30

**for** keys,values **in** dict.items():

print(item)

​

0

1

2

3

4

5

6

7

8

9

​

​

​

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use if statements to perform checks.

Let's go ahead and work through several example of for loops using a variety of data object types.

Iterating through a list

​

**for** num **in** "Asha":

print(num)

A

s

h

a

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an even number!

Back to the for loops!

Let's print only the even numbers from that list!

**for** num **in** list1:

**if** num **%** 2 **==** 0:

print(num)

2

4

6

8

10

We could have also put an else statement in there:

**for** num **in** list1:

**if** num **%** 2 **==** 0:

print(num)

**else**:

print('Odd number')

Odd number

2

Odd number

4

Odd number

6

Odd number

8

Odd number

10

Another common idea during a for loop is keeping some sort of running tally during multiple loops. For example, let's create a for loop that sums up the list:

*# Start sum at zero*

list\_sum **=** 0

list1 **=** [2,3,1,7]

​

**for** num **in** list1:

list\_sum **=** list\_sum **+** num

print(list\_sum)

13

Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a += to perform the addition towards the sum. For example:

*# Start sum at zero*

list\_sum **=** 1

​

**for** num **in** list1:

list\_sum **+=** num

​

print(list\_sum)

56

We've used for loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

**for** letter **in** 'This is a string.':

print(letter)

T

h

i

s

i

s

a

s

t

r

i

n

g

.

Let's now look at how a for loop can be used with a tuple:

tup **=** (1,2,3,4,5)

​

**for** t **in** tup:

print(t)

1

2

3

4

5

Tuples have a special quality when it comes to for loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of tuple unpacking. During the for loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

list2 **=** [(2,4),(6,8),(10,12)]

**for** tup **in** list2:

print(tup)

(2, 4)

(6, 8)

(10, 12)

*# Now with unpacking!*

**for** (t1,t2) **in** list2:

print(t1,t2)

2 4

6 8

10 12

​

​

​

​

With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples.

## Example 7

d **=** {'k1':1,'k2':2,'k3':3}

**for** item,values **in** d.items():

print(item,values)

k1 1

k2 2

k3 3

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: **.keys()**, **.values()** and **.items()**

In Python each of these methods return a dictionary view object. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

*# Create a dictionary view object*

d.items()

Out[18]:

dict\_items([('k1', 1), ('k2', 2), ('k3', 3)])

Since the .items() method supports iteration, we can perform dictionary unpacking to separate keys and values just as we did in the previous examples.

*# Dictionary unpacking*

**for** k,v **in** d.items():

print(k)

print(v)

k1

1

k2

2

k3

3

If you want to obtain a true list of keys, values, or key/value tuples, you can cast the view as a list:

list(d.keys())

Out[20]:

['k1', 'k2', 'k3']

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

sorted(d.values())

Out[21]:

[1, 2, 3]

(<http://www.tutorialspoint.com/python/python_for_loop.htm>)

​

​

​

**while Loops**

The while statement in Python is one of most general ways to perform iteration. A while statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

while test:

code statements

else:

final code statements

Let’s look at a few simple while loops in action.

x **=** 0

​

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**2

x is currently: 0

x is still less than 10, adding 1 to x

x is currently: 1

x is still less than 10, adding 1 to x

x is currently: 2

x is still less than 10, adding 1 to x

x is currently: 3

x is still less than 10, adding 1 to x

x is currently: 4

x is still less than 10, adding 1 to x

x is currently: 5

x is still less than 10, adding 1 to x

x is currently: 6

x is still less than 10, adding 1 to x

x is currently: 7

x is still less than 10, adding 1 to x

x is currently: 8

x is still less than 10, adding 1 to x

x is currently: 9

x is still less than 10, adding 1 to x

Notice how many times the print statements occurred and how the while loop kept going until the True condition was met, which occurred once x==10. It's important to note that once this occurred the code stopped. Let's see how we could add an else statement:

x **=** 0

​

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

**else**:

print('All Done!')

x is currently: 0

x is still less than 10, adding 1 to x

x is currently: 1

x is still less than 10, adding 1 to x

x is currently: 2

x is still less than 10, adding 1 to x

x is currently: 3

x is still less than 10, adding 1 to x

x is currently: 4

x is still less than 10, adding 1 to x

x is currently: 5

x is still less than 10, adding 1 to x

x is currently: 6

x is still less than 10, adding 1 to x

x is currently: 7

x is still less than 10, adding 1 to x

x is currently: 8

x is still less than 10, adding 1 to x

x is currently: 9

x is still less than 10, adding 1 to x

All Done!

**break, continue, pass**

We can use break, continue, and pass statements in our loops to add additional functionality for various cases. The three statements are defined by:

break: Breaks out of the current closest enclosing loop.

continue: Goes to the top of the closest enclosing loop.

pass: Does nothing at all.

Thinking about break and continue statements, the general format of the while loop looks like this:

while test:

code statement

if test:

break

if test:

continue

else:

break and continue statements can appear anywhere inside the loop’s body, but we will usually put them further nested in conjunction with an if statement to perform an action based on some condition.



x **=** 0

​

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

**if** x**==**3:

print('x==3')

**else**:

print('continuing...')

**continue**

x is currently: 0

x is still less than 10, adding 1 to x

continuing...

x is currently: 1

x is still less than 10, adding 1 to x

continuing...

x is currently: 2

x is still less than 10, adding 1 to x

x==3

x is currently: 3

x is still less than 10, adding 1 to x

continuing...

x is currently: 4

x is still less than 10, adding 1 to x

continuing...

x is currently: 5

x is still less than 10, adding 1 to x

continuing...

x is currently: 6

x is still less than 10, adding 1 to x

continuing...

x is currently: 7

x is still less than 10, adding 1 to x

continuing...

x is currently: 8

x is still less than 10, adding 1 to x

continuing...

x is currently: 9

x is still less than 10, adding 1 to x

continuing...

Note how we have a printed statement when x==3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x ==3 and see if the result makes sense:

x **=** 0

​

**while** x **<** 10:

print('x is currently: ',x)

print(' x is still less than 10, adding 1 to x')

x**+=**1

**if** x**==**3:

print('Breaking because x==3')

**break**

**else**:

print('continuing...')

**continue**

x is currently: 0

x is still less than 10, adding 1 to x

continuing...

x is currently: 1

x is still less than 10, adding 1 to x

continuing...

x is currently: 2

x is still less than 10, adding 1 to x

Breaking because x==3

Note how the other else statement wasn't reached and continuing was never printed!

**A word of caution however! It is possible to create an infinitely running loop with while statements. For example:**

*# DO NOT RUN THIS CODE!!!!*

**while** **True**:

print("I'm stuck in an infinite loop!")

A note: If you *did* run the above cell, click on the Kernel menu above to restart the kernel!

a**=** 10

b**=**20

​

**if** a**<**b:

**pass**

**if** a**<**b:

​

**def** function():

**pass**



​

​

**import** numpy **as** np

v1 **=** [1, 22, 33]

v2 **=** ["Hi", "there", "friend"]

v3 **=** [11, 3, 2016]

D **=** np.array([v1,v2,v3])

print(D[2,1])

3

​

​