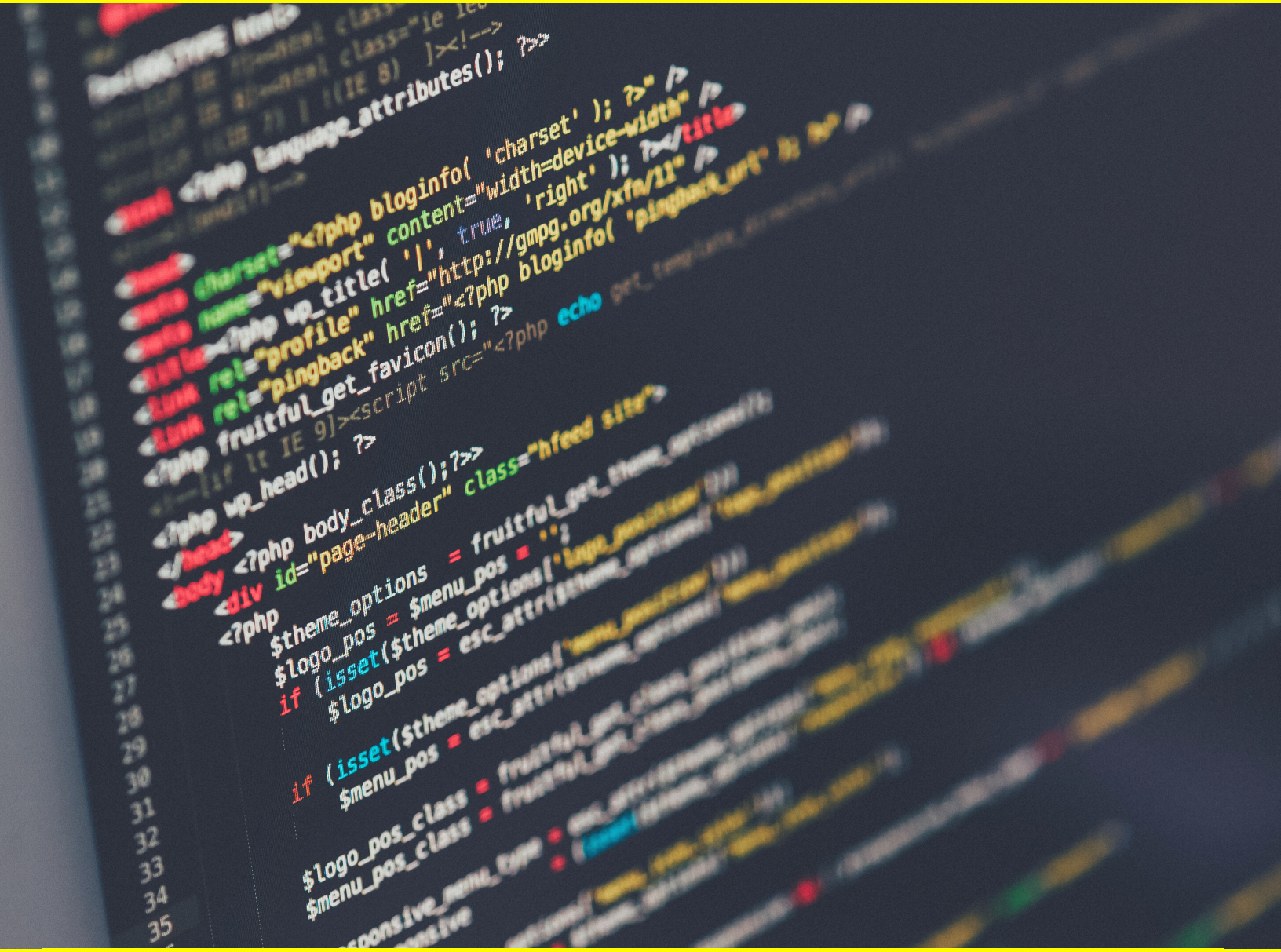
the fast lane to python

PART 7



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

“LEARNING NEVER EXHAUSTS THE MIND!!!
For the things we have to learn before we can do them, we learn by doing them, and i believe whoever thought that it will be tough to learn Python so quick and with so much clarity must think again :D

So proud that we have reached Part 7, with so much consistency, but we still have a long way to so push ourselves, cause no one else is going to do it for you

too much motivation, LET's ROLL:

Tuple Packing

we ended Last part after successfully learning everything about function, except one. We returned values whenever the calling function needed some response from the function but what if we have to return more than one values? Will see in detail about this

We have seen how we can create tuples by placing elements between parentheses () but if multiple expressions are provided, separated by commas, even they are automatically packed into a tuple.

we can omit the parentheses when assigning a tuple of values to a single variable.

```
tup_var=("python", "is", "number", 1)
tup_var1= "python", "is", "number", 1
```

both tup_var & tup_var1 variables are referencing to a tuple elements.

Tuple Assignment with Unpacking

Python has a very powerful tuple assignment feature that allows a tuple of variable names on the left of an assignment statement to be assigned values from a tuple on the right of the assignment. Another way to think of this is that the tuple of values is unpacked into the variable names.

tup_var=("Jim","Parsons",2020,"Big Bang Theory")

name, Surname, year, Show=tup_var

```
1  tup_var=("Jim","Parsons",2020,"Big Bang Theory")
2
3  name, Surname, year, Show=tup_var
4  print(name)
5  print(Surname)
6  print(year)
7  print(Show)
```

```
Jim
Parsons
2020
Big Bang Theory
```

Pretty clean and easy but we need to remember, the number of variables on the left and the number of values on the right have to be the same. Or else we might end up with an error.

```
tup_var=("Jim","Parsons",2020,"Big Bang Theory")

name, Surname, year,Show, new=tup_var
print(name)
print(Surname)
print(year)
print(Show)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-309-951237dbcf8b> in <module>
      1 tup_var=("Jim","Parsons",2020,"Big Bang Theory")
      2
----> 3 name, Surname, year,Show, new=tup_var
      4 print(name)
      5 print(Surname)

ValueError: not enough values to unpack (expected 5, got 4)
```

Swapping Values between Variables

This feature is used to enable swapping the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b

```
a = 1
b = 2
temp = a
a = b
b = temp
print(a, b, temp)

a = 1
b = 2
(a, b) = (b, a)
print(a, b)
```

```
2 1 1
2 1
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Unpacking Into Iterator Variables

```
1 founder=[('python','Guido'), ('java','James') , ('C++','Danis')]
2
3 for lang,found in founder:
4     print("the founder of {} is {}".format(lang,found))
```

```
the founder of python is Guido
the founder of java is James
the founder of C++ is Danis
```

Multiple assignment with unpacking is particularly useful when you iterate through a list of tuples. You can unpack each tuple into several loop variables.

- In for loop, the lang and found are 2 iterators which will take elements from founder. We could have written in a different way for better understanding taking 2 more variables

```
1 founder=[('python','Guido'), ('java','James') , ('C++','Danis')]
2
3 for _ in founder:
4     k=_[0]
5     v=_[1]
6     print("the founder of {} is {}".format(k,v))
```

```
the founder of python is Guido
the founder of java is James
the founder of C++ is Danis
```

Let's try to unpack using dictionary:

```
pokemon = {'Rattata': 19, 'Machop': 66, 'Seel': 86, 'Volbeat': 86, 'Solrock': 126}
p_names=[]
p_number=[]
for i in pokemon.items():
    p_names.append(i[0])
    p_number.append(i[1])
print(p_names)
print(p_number)
```

```
['Rattata', 'Machop', 'Seel', 'Volbeat', 'Solrock']
[19, 66, 86, 86, 126]
```

Tuples as Return Values

Functions can return tuples as return values. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some ecological modeling we may want to know the number of rabbits and the number of wolves on an island at a given time. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

```
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

circumference, area= circleInfo(10)
print("the circumference of the given cirle is ",circumference)
print("the area of the given cirle is ",area)
```

```
the circumference of the given cirle is  62.8318
the area of the given cirle is  314.159
```

Unpacking Tuples as Arguments to Function Calls

Python even provides a way to pass a single tuple to a function and have it be unpacked for assignment to the named parameters

```
1 def add(x, y):
2     """function to add 2 numbers"""
3     return x + y
4 
5 print(add(3, 4)) #here we are sending 2 separate actual arguments
6 z = (5, 4)
7 print(add(z)) # this line causes an error
8 
```

```
7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-327-b7d067264c05> in <module>
      5 print(add(3, 4)) #here we are sending 2 separate actual arguments
      6 z = (5, 4)
----> 7 print(add(z)) # this line causes an error

TypeError: add() missing 1 required positional argument: 'y'
```

It caused an error but we thought we could pass the tuple and python should be smart enough to understand how to unpack and assign the values to the variables but it failed :O

because the function add is expecting two parameters, but you're only passing one parameter (a tuple).

You might be thinking shouldn't there there be a way to tell python to unpack that tuple and assign the values to the element accordingly??
To add to your excitement, yes we have a way, cause common it's python

```python
def add(x, y):
    """function to add 2 numbers"""
    return x + y

print(add(3, 4)) #here we are sending 2 separate actual arguments
z = (5, 4)
print(add(*z)) # we added a * and this helped pythong to understand
               #what we want

```

```
7
9
```

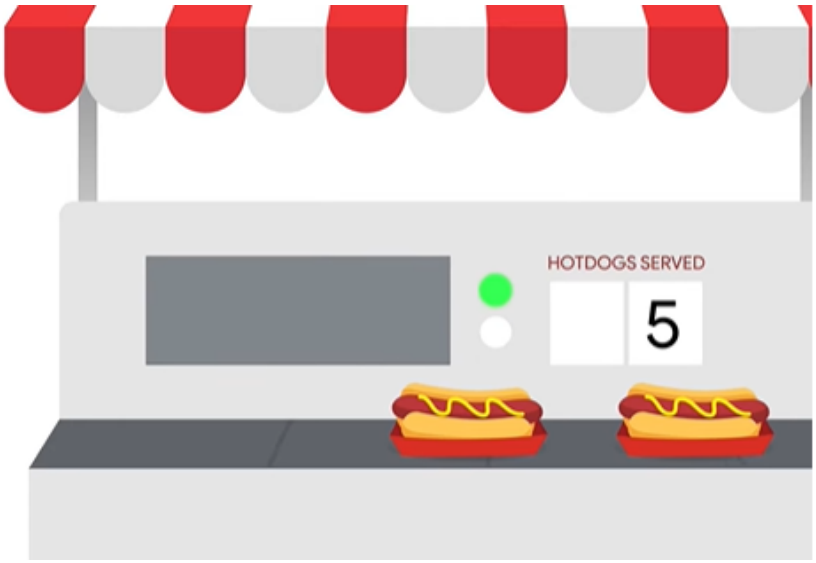
Isn't that amazing, just adding a * can let you pass a tuple which will unpack during faction call and assign all the values sequentially.

the more you learn , the more you will keep falling in love with python.

The while statement

well, we are back with loops/iterations but wait, why do we need another loop when we have already learned **FOR loop** and it did just fine for all our problems?

Let's see:



Suppose, you own a automated hotdog delivery shop, and you see 5 customers standing outside the shop in queue. Simple , isn't it? you write a code with for loop and feed the program with count as 5 to deliver 5 hotdogs.

```
for x in range(5):
  serveHotDog()
```

Easy!!! For is a definite iterations loop cause you know the limit.

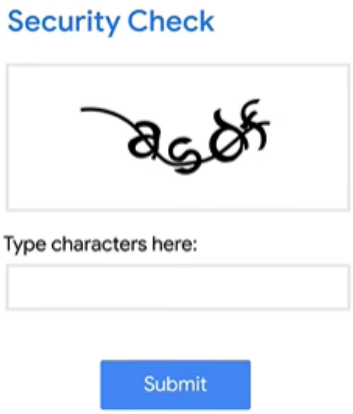
But what if the queue increases too big and you cannot count the number of customers standing in queue? SOS

Now the conditions have changed, and you can serve the customers only until the limit of hotdogs isn't exhausted. And you also don't know the exact quantity of hotdogs you have, now the approach will be to have a check on if the number of hotdogs is greater than 0 -> keep delivering

```
while numHotDogs > 0:
  serveHotDog()
  numHotDogs = numHotDogs - 1
```

Amazing... this is called indefinite iteration cause we are not sure for how many times will our loop continue to iterate.

Before we start, let's try to go through the algorithm of how the very famous security check "enter captcha" works :
Captcha is something which most of the webpages uses to differentiate between a robot and a human, every captcha has just 1 correct answer but it is depicted in not a really clear way

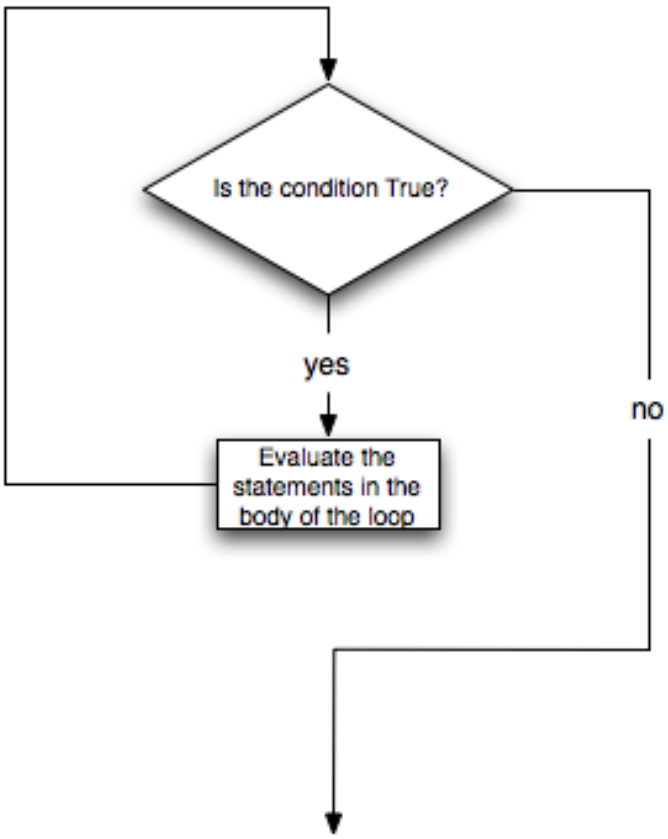


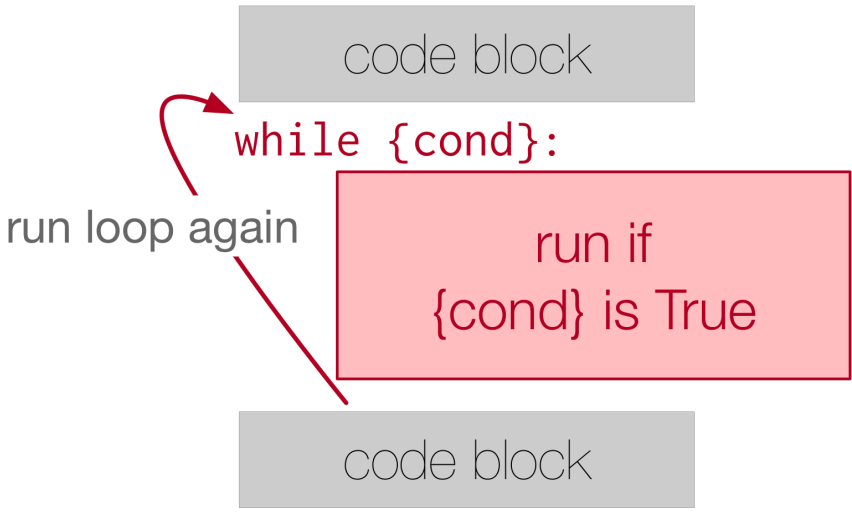
Sample program algorithm for taking captcha in capital or small letters from the user and comparing if the user has inserted the right captcha.

```
import random, string

def getcaptchaans():
    valid_input=False
    return valid_input

def showNewCaptcha():
    x = ''.join(random.choice(string.ascii_uppercase + string.ascii_lowercase + string.digits)
                for _ in range(4))
    print(x)
    return x


def input_check(valid_input):
    while not valid_input:
        validAnswer=showNewCaptcha()
        answer=input("take the input")
        #answer=answer.lower()
        #validAnswer= validAnswer.lower()
        if answer == validAnswer:
            valid_input = True
        else:
            print('please try again')
    return answer


input_check(getcaptchaans())

```

```
hwTh
take the inputHWTH
please try again
qtEQ
take the inputqtEQ

'qtEQ'
```

The **while statement** provides a **much more general mechanism** for **iterating**. Similar to the **if statement**, it uses a boolean expression to control the flow of execution. The body of while will be repeated as long as the controlling boolean expression evaluates to True.





We can use the while loop to create any type of iteration we wish, including anything that we have previously done with a for loop.

```
1  def sum_return(limit):
2      the_sum=0
3      number=0
4
5      while number<=limit:
6          the_sum=the_sum + number
7          number = number+1
8      return the_sum
9
10 print(sum_return(5))
```

15

Let's try to break while loop formally, the flow of execution for a while statement looks like ;

1.Evaluate the condition, yielding False or True.

2.If the condition is False, exit the while statement and continue execution at the next statement.

3.If the condition is True, execute each of the statements in the body and then go back to step 1.

Question: Write a while loop that is initialized at 0 and stops at 15. If the counter is an even number, append the counter to a list called eve_nums.

solution

```
eve_nums=[]
itr=0
while itr <15:
    if itr%2 == 0:
        eve_nums.append(itr)

    itr = itr+1
```

Question 2: convert below for loop into while

```
list1 = [8, 3, 4, 5, 6, 7, 9]

tot = 0
for elem in list1:
    tot = tot + elem
```

Question: Write a function called stop_at_four that iterates through a list of numbers. Using a while loop, append each number to a new list until the number 4 appears. The function should return the new list.

Solution:

```
def stop_at_four(list1):
    list2=[]
    count=0
    while count< len(list1) and list1[count] != 4:
        list2=list2 + [list1[count]]
        count= count + 1
    return list2

print(stop_at_four([0,9,4.5,7.4]))
```

The Listener Loop

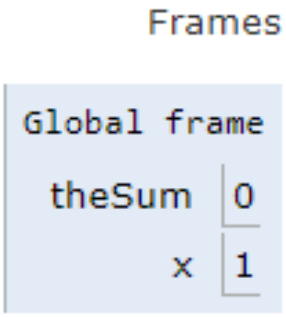
What will you do when you don't know how many times the user needs to provide an input? You surely cannot use for loop, while loop comes to our rescue but even while loop is used in different ways, out of which one **is LIstener Loop**

If in a while loop there is a function call to get user input. The loop will run indefinitely, until a particular input is received.

```
theSum = 0
x = -1
while (x != 0):
    x = int(input("next number to add up (enter 0 if no more numbers): "))
    theSum = theSum + x

print(theSum)
```

```
next number to add up (enter 0 if no more numbers): 1
next number to add up (enter 0 if no more numbers): 2
next number to add up (enter 0 if no more numbers): 3
next number to add up (enter 0 if no more numbers): 4
next number to add up (enter 0 if no more numbers): 5
next number to add up (enter 0 if no more numbers): 6
next number to add up (enter 0 if no more numbers): 7
next number to add up (enter 0 if no more numbers): 8
next number to add up (enter 0 if no more numbers): 9
next number to add up (enter 0 if no more numbers): 0
45
```

let's try to see what is happening in the above program:



- the interpreter assigns the global values
- Later enters the while loop and checks the condition of x!=0
- finds it true enters the block and gives the user to feed a number
- adds it to the sum and after not finding anymore commands to execute as per indentation , gives back control to while condition and again checks the value and goes on until the user feeds 0 as the input.
- Here the interpreter is listening to the while condition and going in an indefinite loop.

Other uses of while

lets see some real life examples where we are using indefinite loops :

- When the baggage crew unloads a plane, they don't know in advance how many suitcases there are. They just keep unloading while there are bags left in the cargo hold. (Why your suitcase is always the last one is an entirely different problem.)

- When you go through the checkout line at the grocery, the clerks don't know in advance how many items there are. They just keep ringing up items as long as there are more on the conveyor belt.

will try to write a program for grocery billing system.

Step 1: Algorithm

- while moreitems
  ask for price
  add price to total
  add one to product_count

```
def bill():
    count=0
    total=0
    more_items=True
    while more_items:
        price=input("the price of the product(if no more product press 0)= ")
        if int(price) != 0:
            total=total + int(price)
            count = count + 1
        else:
            more_items=False
    return total

print(bill())
```

```
the price of the product(if no more product press 0)= 123
the price of the product(if no more product press 0)= 12
the price of the product(if no more product press 0)= 1
the price of the product(if no more product press 0)= 412
the price of the product(if no more product press 0)= 0
548
```

There are still a few problems with this program.

- If you enter a negative number, it will be added to the total and count. Let's try to fix it

```
def bill():
    count=0
    total=0
    more_items=True
    while more_items:
        price=input("the price of the product(if no more product press 0)= ")
        if int(price) < 0:
            print('You gave wrong input, check the prices again n retry')
        elif int(price) >0:
            total=total + int(price)
            count = count + 1
        else:
            more_items=False
    return total

print(bill())
```

```
the price of the product(if no more product press 0)= -132
You gave wrong input, check the prices again n retry
the price of the product(if no more product press 0)= 11
the price of the product(if no more product press 0)= 1
the price of the product(if no more product press 0)= 2
the price of the product(if no more product press 0)= 3
the price of the product(if no more product press 0)= 4
the price of the product(if no more product press 0)= 5
the price of the product(if no more product press 0)= 0
26
```

Validating Input

You can also use a while loop when you want to validate input, let's try to understand with a program

```
def feedback():
    valid_input=False
    while not valid_input:
        response=input('Please give us the feedback of how you liked the food')
        if response =='Y' or response == 'N':
            return response
        else:
            valid_input=False

response=feedback()
if response == 'Y':
    print('Thank you for your feedback, hope will serve you again')
else:
    print("""We are sorry that you didnt like the food,
    we hope you will give us chance to serve you better again""")

```

```
Please give us the feedback of how you liked the foody
Please give us the feedback of how you liked the foodY
Thank you for your feedback, hope will serve you again
```

Randomly Walking Turtles

Will take 1 scenario how can we use while + turtle together, you can create your own ideas and try to solve them. Let your imagination explore.
Consider a situation which behaves in the following way:

- The turtle begins in the center of the screen.
- Flips a coin. If it’s heads then turns left 90 degrees. If it’s tails then turns right 90 degrees & takes 50 steps forward.
- If the turtle has moved outside the screen then stop, otherwise go back to step 2 and repeat.

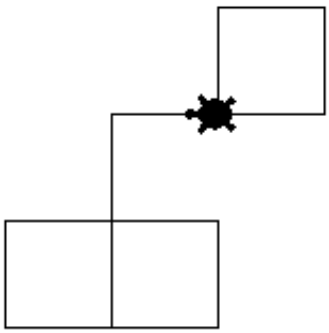
Easy?

Algorithm

- create a window
  create a turtle
  while the turtle is still in window:
      generate a random number between 0 and 1
      if number == 0(heads):
          turn left
      else:
          turn right
      move forward 50

Now we have the Idea, let's try to write it down in python

```
import turtle
import random

def turt_in_screen(w,t):
    if random.random() > 0.1:
        return True
    else:
        return False

def turtle_while():
    wn=turtle.Screen()
    turt=turtle.Turtle()

    turt.shape('turtle')
    while turt_in_screen(wn,turt):
        if random.randrange(0,2) == 0:
            turt.left(90)
        else:
            turt.right(90)
        turt.forward(50)
    wn.exitonclick()

turtle_while()
```



the first question that might arise in your mind after looking at the program is how turt_in_screen where we are using a random function to derive value between 0 & 1 could ever decide if the turtle in inside the screen? :D

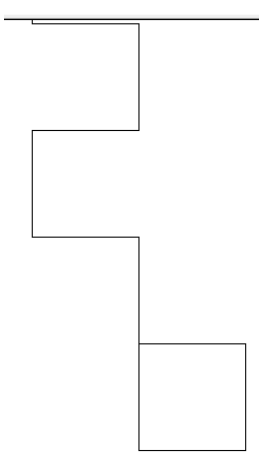
1 good thing about programming is we can delay the tough stuff and get something in our program working right away. The way we are going to do this is to delegate the work of deciding whether the turtle is still in the screen or not to a boolean function.

We called the boolean function turt_ins_screen, it always return True or false deciding randomly. Function returning only True would not be a good idea so we wrote our version to decide randomly. Considering that there is a 90% probability the turtle is still in the window and 10% that the turtle has escaped. we took the condition random.random() > 0.1 .

Let's try to handle the complex part now

- We can **find** out the width and the height of the screen using the **window_width** and **window_height methods** of the screen object. However, remember that the **turtle starts** at **position 0,0** in the middle of the screen.
- So we **never want** the turtle to go **farther right than width/2** or **farther left than negative width/2.** We never want the turtle to go **further up than height/2** or **further down than negative height/2**.
- Once we know what the boundaries are we can use some conditionals to check the turtle position against the boundaries and return False if the turtle is outside or True if the turtle is inside.

```python
import turtle
import random

def turt_in_screen(t,lb,rb,tb,bb):
    stillIn=True
    if t.xcor() > rb or t.xcor() < lb:
        stillIn=False
    elif t.ycor() > tb or t.ycor() < bb:
        stillIn=False
    return stillIn

def turtle_while():
    wn=turtle.Screen()
    turt=turtle.Turtle()
    left_bound=-(wn.window_width()/2)
    right_bound=(wn.window_width()/2)
    top_bound=(wn.window_height()/2)
    bottom_bound=-(wn.window_height()/2)

    turt.shape('turtle')
    print('calling function')
    while turt_in_screen(turt,left_bound,right_bound,top_bound,bottom_bound):
        if random.randrange(0,2) == 0:
            turt.left(90)
        else:
            turt.right(90)
        turt.forward(100)
    wn.exitonclick()

turtle_while()
```

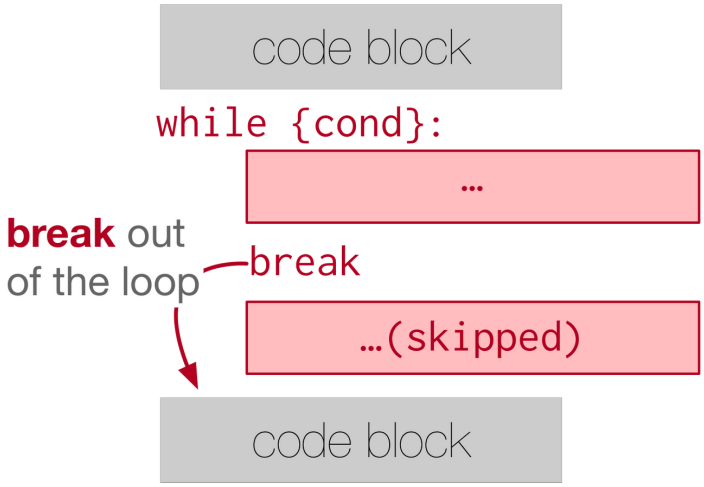


Break and Continue

When we write programs, we need total control of how things would flow from one part to another, which is where Python provides ways for us to control the flow of iteration with two keywords:

- break
- continue.

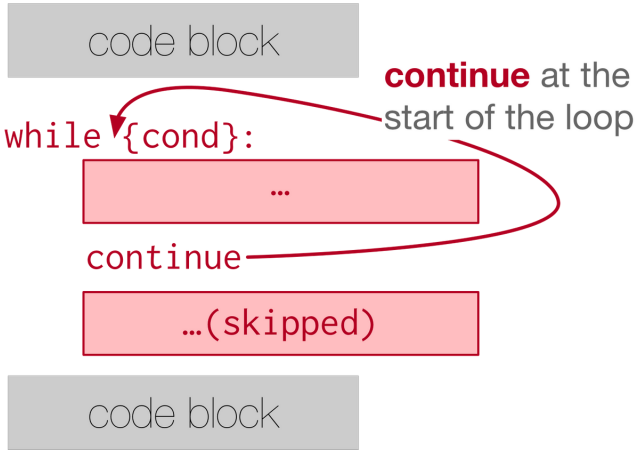
break allows the program to immediately **'break out' of the loop, regardless** of the **loop's conditional structure.** This means that the program will then skip the rest of the iteration, **without rechecking the condition**, and just goes on to the next outdented code that exists after the whole while loop. Same works with for loop too.




```
1  while True:
2      print("this phrase will always print")
3      break
4      print("Does this phrase print?")
5
6  print("We are done with the while loop.")
7
```

```
this phrase will always print
We are done with the while loop.
```

continue is the other keyword that can control the flow of iteration. Using continue allows the program to **immediately "continue" with the next iteration.** The program will **skip the rest of the iteration, recheck the condition**, and maybe does another iteration depending on the condition set for the while loop.




```
1  x = 0
2  while x < 10:
3      print("we are incrementing x")
4      if x % 2 == 0:
5          x += 3
6          continue
7      if x % 3 == 0:
8          x += 5
9      x += 1
10 print("Done with our loop! X has the value: " + str(x))
11
```

```
we are incrementing x
we are incrementing x
we are incrementing x
Done with our loop! X has the value: 15
```

Infinite Loops

While loop has helped us in many ways until now but as we read in very start that while loop is also called as indefinite loop that would mean we need to handle it with care or it might go bad and actually lead to infinite loops

Let's try to see in which all scenarios it might happen.

First off all, if the variable that you are using to determine while loop is unknowingly getting reset inside the while loop, then your code would turn into an infinite loop. **(Unless of course you use break to break out of the loop.)**

- this is an infinite loop

```
while True:
   print("Will this stop?")
print("We have escaped.")
```

Another case where an infinite loop is likely to occur is when you have reassigned the value of the variable used in the while statement in a way that prevents the loop from completing.

```
b = 15
while b < 60:
    b = 5
    print("Bugs")
   b = b + 7
```

This would go on forever

Questions

Write a function called beginning that takes a list as input and contains a while loop that only stops once the element of the list is the string ‘bye’. What is returned is a list that contains up to the first 10 strings, regardless of where the loop stops. (i.e., if it stops on the 32nd element, the first 10 are returned. If “bye” is the 5th element, the first 4 are returned.) If you want to make this even more of a challenge, do this without slicing

answer:

```
def beginning(lst):
    lst1=[]
    count=0
    while count < len(lst):
        if lst[count] == 'bye':
            break
        elif count < 10:
            lst1.append(lst[count])
        count = count+1

   return lst1
```

Question 2:Write a function called stop_at_z that iterates through a list of strings. Using a while loop, append each string to a new list until the string that appears is “z”. The function should return the new list.

Answer:

```
def stop_at_z(lst_num):
    var=0
    lst1=[]
    while var < len(lst_num):
        if lst_num[var] == 'z':
            break
        else:
            lst1.append(lst_num[var])
        var=var + 1
   return lst1
```

Question 3:

Using a while loop, create a list numbers that contains the numbers 0 through 35. Your while loop should initialize a counter variable to 0. On each iteration, the loop should append the current value of the counter to the list and the counter should increase by 1. The while loop should stop when the counter is greater than 35.

Answer:

```
count=0
numbers=[]
while count<36:
    numbers.append(count)
   count=count+1
```

Question 4:

Modify the walking turtle program so that rather than a 90 degree left or right turn the angle of the turn is determined randomly at each step.

Answer:

```
import turtle
import random
def turt_in_screen(t,lb,rb,tb,bb):
    stillIn=True
    if t.xcor() > rb or t.xcor() < lb:
        stillIn=False
    elif t.ycor() > tb or t.ycor() < bb:
        stillIn=False
    return stillIn
def turtle_while():
    wn=turtle.Screen()
    turt=turtle.Turtle()
    left_bound=-(wn.window_width()/2)
    right_bound=(wn.window_width()/2)
    top_bound=(wn.window_height()/2)
    bottom_bound=-(wn.window_height()/2)

    turt.shape('turtle')
    while turt_in_screen(turt,left_bound,right_bound,top_bound,bottom_bound):
        if random.randrange(0,2) == 0:
            turt.left(random.randrange(0,361))
        else:
            turt.left(random.randrange(0,361))
        turt.forward(50)
    wn.exitonclick()

turtle_while()
```

Question 5:

Modify the turtle walk program so that you have two turtles each with a random starting location. Keep the turtles moving until one of them leaves the screen.

```
import turtle
import random

def turt_in_screen(t,lb,rb,tb,bb):
    stillIn=True
    if t.xcor() > rb or t.xcor() < lb:
        stillIn=False
    elif t.ycor() > tb or t.ycor() < bb:
        stillIn=False
    return stillIn

def turtle_while():
    wn=turtle.Screen()
    burt=turtle.Turtle()
    turt=turtle.Turtle()
    left_bound=-(wn.window_width()/2)
    right_bound=(wn.window_width()/2)
    top_bound=(wn.window_height()/2)
    bottom_bound=-(wn.window_height()/2)

    turt.shape('turtle')
    burt.shape('turtle')
    #turt.speed(0)
    #burt.speed(0)
    turt.up()
    turt.goto(random.randrange(left_bound, right_bound),
        random.randrange(bottom_bound, top_bound))
    turt.setheading(random.randrange(0, 360))
    turt.down()
    print('calling function')
    while turt_in_screen(turt,left_bound,right_bound,top_bound,bottom_bound) and
turt_in_screen(burt,left_bound,right_bound,top_bound,bottom_bound):
        if random.randrange(0,2) == 0:
            turt.left(random.randrange(0,361))
            burt.left(random.randrange(0,361))
        else:
            turt.right(random.randrange(0,361))
            burt.right(random.randrange(0,361))
        turt.forward(50)
        burt.forward(50)
    wn.exitonclick()

turtle_while()
```

Optional Parameters

So far we have used functions with zero or more formal parameters and each function invocation provides exactly that many values.

But sometimes it is **convenient to have optional parameters** that can be specified or omitted. When an optional parameter is omitted from a function invocation, the **formal parameter is bound to a default value**

Consider, for example, the **int function**, which we have used so many times until now .

- Its **first parameter**, which is required, **specifies the object** that you **wish to convert** to an integer.
    - For example, if you call in on a string, int("100"), the return value will be the integer 100.

The **int function** provides an **optional parameter for the base**. When it is not specified, the number is converted to an integer assuming the original number was in base 10. We say that **10 is the default value**. So **int("100")** is the **same** as **invoking int("100", 10).**

We can override the default of 10 by supplying a different value.

```
1 print(int("100"))
2 print(int("100", 10))   # same thing, 10 is the default value for the base
3 print(int("100", 8))     # now the base is 8, so the result is 1*64 = 64
4
```

```
100
100
64
```

```
initial=7
def optional_param(x,y=3,z=initial):
    print('x, y, z values are {} {} {}'.format(x,y,z))
optional_param(2)
optional_param(2,5)
optional_param(2,5,10)
```

```
x, y, z values are 2 3 7
x, y, z values are 2 5 7
x, y, z values are 2 5 10
```

- Notice the different bindings of x, y, and z on the three invocations of f. The first time, y and z have their default values, 3 and 7.
- The second time, y gets the value 5 that is passed in, but z still gets the default value of 7.
- The last time, z gets the value 10 that is passed in.If you want to provide a non-default value for the third parameter (z), you also need to provide a value for the second item (y).

*****KEY POINT:**

In a stand-alone assignment statement, not part of a function definition, x=3 assigns 3 to the variable x. As part of specifying the parameters in a function definition, x=3 says that 3 is the default value for x, used only when no value is provided during the function invocation.

```
initial=7
def optional_param(x,y=3,z=initial):
    print('x, y, z values are {} {} {}'.format(x,y,z))

initial=12
optional_param(2)
optional_param(2,5)
optional_param(2,5,10)
```

```
x, y, z values are 2 3 7
x, y, z values are 2 5 7
x, y, z values are 2 5 10
```

in above program we see a tricky thing that would have confused you :

- That the **default value is determined at the time** that the **function is defined**, **not** at the **time** that it is **invoked**. So in the example above, if we wanted to invoke the function optional_param with a value of 10 for 12, we cannot simply set initial = 10 right before invoking optional_param.
- Because the value of initial has already been defined and set as default 7, so now only the actual parameters being passed to the function can only override the default value

```
def mutable_list(x, lst=[]):
    lst.append(x)
    return lst

print(mutable_list(1))
print(mutable_list(2))
print(mutable_list(3))
print(mutable_list(4))
print(mutable_list(5))
print(mutable_list(6,['Hello']))
print(mutable_list(7,['Bye']))
```

```
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
['Hello', 6]
['Bye', 7]
```

Another tricky thing in python is with if the default value is set to a mutable objects, such as list or dictionary, that object will be shared in all invocations of the function.

As you see above, until 5th function call lst kept appending all the int, but as soon as the actual parameter was passed, it overrode the earlier list and started pointing to a new list, need to be really carefully while dealing with such situations

Keyword Parameters

by now you know how to define default values for formal parameters. you'll see one more way to invoke functions with optional parameters, with keyword-based parameter passing

The basic idea of passing arguments by keyword is very simple. When invoking a function, inside the parentheses there are always 0 or more values, separated by commas. With keyword arguments, some of the values can be of the form **paramname = <expr>** instead of just <expr>.

Note that when you have **paramname = <expr> in a function definition**, it is **defining the default value** for a **parameter** when no value is provided in the invocation; **when you have paramname = <expr> in the invocation**, it is **supplying a value**, **overriding** the default for that paramname.

Let's try to understand in details with an example, each time the function is invoked, make a prediction about what each of the four parameter values will be during execution of lines 2-5. Then only execute one call and see what they actually are returning.

******KEY POINT:***
keyword parameters should always be mentioned after the formal arguments i.e variable without default values should always come first , followed by keyword parameters or it will give you an error

```python
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't " + action,)
    print("if you put " + str(voltage) + " volts through it.")
    print("-- Lovely plumage, the " +  type)
    print("-- It's " + state + "!")

parrot(1000)                                         # 1 positional argument
parrot(voltage=1000)                                 # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')            # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)            # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')        # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

```
-- This parrot wouldn't voom
if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff!
-- This parrot wouldn't voom
if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff!
-- This parrot wouldn't VOOOOOM
if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff!
-- This parrot wouldn't VOOOOOM
if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff!
-- This parrot wouldn't jump
if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life!
-- This parrot wouldn't voom
if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies!
```

Keyword Parameters with .format

We have already learned about format in part3 of our series and you should know very well how to use the format method for strings, which allows you to structure strings like fill-in-the-blank sentences.
Now that you've learned about optional and keyword parameters, we can introduce a new way to use the format method.

```
results = [("Amit",[67,77,87]),("Vijay",[77,87,97]),("Vishal",[53,64,75])]
for name, scores in results:
    print("The scores {nm} got were: {s1},{s2},{s3}.".format(nm=name,s1=scores[0],s2=scores[1],s3=scores[2]))
    print("the scores {0} got were: {1}, {2}, {3}.".format(name,scores[0],scores[1],scores[2]))
    print("the scores {} got were: {}, {}, {}.".format(name,scores[0],scores[1],scores[2]))
```

```
The scores Amit got were: 67,77,87.
the scores Amit got were: 67, 77, 87.
the scores Amit got were: 67, 77, 87.
The scores Vijay got were: 77,87,97.
the scores Vijay got were: 77, 87, 97.
the scores Vijay got were: 77, 87, 97.
The scores Vishal got were: 53,64,75.
the scores Vishal got were: 53, 64, 75.
the scores Vishal got were: 53, 64, 75.
```

What value will be printed for x,y, z?

```
initial = 7
def f(x, y = 3, z = initial):
    print("x, y, z are:", x, y, z)
f(2, 5)
```

What value will be printed for x?

```
initial = 7
def f(x, y = 3, z = initial):
    print("x, y, z are:", x, y, z)
f(2, x=5)
```

Runtime error since two different values are provided for xCheck meCompare me

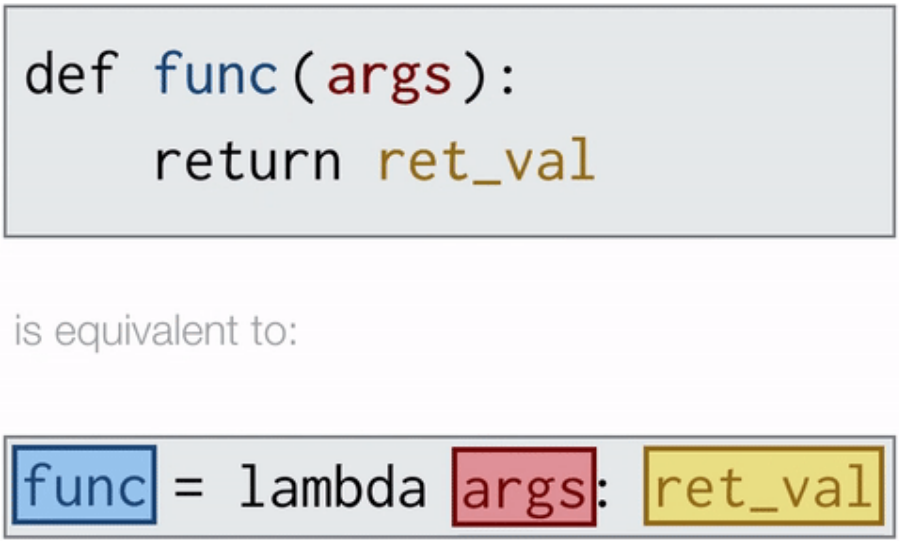
What value will be printed below?
names = ["Alexey", "Catalina", "Misuki", "Pablo"]
print("'{first}!' she yelled. 'Come here, {first}! {f_one}, {f_two}, and {f_three} are here!'".format(first = names[1], f_one = names[0], f_two = names[2], f_three = names[3]))

Anonymous functions with lambda expressions

To further drive home the idea that we are passing a function object as a **parameter** to the sorted object, let's see an **alternative notation** for **creating a function, a lambda expression.**

The **syntax** of a lambda expression is the word **"lambda"** followed by parameter names, separated by commas but not inside (parentheses), followed by a colon and then an expression.
lambda arguments: expression yields a function object. This unnamed object behaves just like the function object constructed below.

```
def fname(arguments):
    return expression
```



```
1  def f(x):
2      return x - 1
3
4  print(f)
5  print(type(f))
6  print(f(3))
7
8  print(lambda x: x-1)
9  print(type(lambda x: x -1))
10 print((lambda x: x -1)(6))
```

```
<function f at 0x000002720B270F70>
<class 'function'>
2
<function <lambda> at 0x000002720B270700>
<class 'function'>
5
```

- At line 4, **f is bound to a function object.** Its printed representation is “<function f>”.

- At line 8, **the lambda expression produces a function object**. **Because** it is **unnamed (anonymous),** its printed representation doesn’t include a name for it, “<function <lambda>>”. Both are of type ‘function’.

- A function, whether named or anonymous, can be called by placing parentheses () after it. In this case, because there is one parameter, there is one value in parentheses. This works the same way for the named function and the anonymous function produced by the lambda expression.

- The lambda expression had to go in parentheses just for the purposes of grouping all its contents together. Without the extra parentheses around it on line 10, the interpreter would group things differently and make a function of x that returns x - 2(6).

- You might find it confusing, solution is to practice more examples

******KEY POINT***

In a typical function, we have to use the keyword return to send back the value. In a lambda function, that is not necessary - whatever is placed after the colon is what will be returned.

Programming With Style

Try to consider below points as standards and make them your habit:

- *u*se 4 spaces for indentation
- imports should go at the top of the file
- separate function definitions with two blank lines
- keep function definitions together
- keep top level statements, including function calls, together at the bottom of the program

Write a function called checkingIfIn that takes three parameters. The first is a required parameter, which should be a string. The second is an optional parameter called direction with a default value of True. The third is an optional parameter called d that has a default value of {'apple': 2, 'pear': 1, 'fruit': 19, 'orange': 5, 'banana': 3, 'grapes': 2, 'watermelon': 7}. Write the function checkingIfIn so that when the second parameter is True, it checks to see if the first parameter is a key in the third parameter; if it is, return True, otherwise return False.But if the second paramter is False, then the function should check to see if the first parameter is not a key of the third. If it's not, the function should return True in this case, and if it is, it should return False.

```
def checkingIfIn(s,direction=True,d={'apple': 2, 'pear': 1, 'fruit': 19, 'orange': 5, 'banana': 3, 'grapes': 2,
'watermelon': 7}):
    if direction:
        if s in d.keys():
            return True
        else:
            return False
    else:
        if s not in d.keys():
            return True
        else:
         return False
```

Sorting with Sort and Sorted

Remember, though without its entire information we have used a method sort in some of our examples while dealing with lists?

if you haven't read the old parts, then i am afraid you would be able to recall , but no problem will go through it now and understand it in detail.

When we invoked sort method on a list, the order of items in the list was changed. If no optional parameters are specified, the items are arranged in whatever the natural ordering is for the item type.

For example, if the items are all integers, then smaller numbers go earlier in the list. If the items are all strings, they are arranged in alphabetic order.

```
1 lst=[1,34,15,-134,0,51]
2 lst1=['sort', 'me']
3 lst.sort()
4 lst1.sort()
5 print(lst)
6 print(lst1)
7 print(lst1.sort())
```

```
[-134, 0, 1, 15, 34, 51]
['me', 'sort']
None
```

KEY POINT:
the sort method does not return a sorted version of the list. In fact, it returns the value None. But the list itself gets modified. This kind of operation that works directly on the passed object and does not return anything can be quite confusing.

In order to avoid this confusion, we have another function named as sorted() . Because **sorted() is a function rather than a method** (we will see method in details while we start reading about OOPS in python), it is invoked on a list by passing the list as a parameter inside the parentheses (which have been doing since always for function), rather than putting the list before the period.

More importantly, sorted does not change the original list. Instead, it returns a new list.

```python
L2 = ["python", "java", "Linux"]

L3 = sorted(L2)
print(L3)
print(sorted(L2))
print(L2) # unchanged
print(sorted([1,3,5,1,6,78,23]))

print("----")

L2.sort()
print(L2)
print(L2.sort())  #return value is None
```

```
['Linux', 'java', 'python']
['Linux', 'java', 'python']
['python', 'java', 'Linux']
[1, 1, 3, 5, 6, 23, 78]
----
['Linux', 'java', 'python']
None
```

Optional reverse parameter

It is not necessary that we will always need the output in the default state as for number in ascending order and for words in alphabetic order.

So we do need some optional parameter to manipulate the output, which is where sorted function provides some optional parameters:

- the first optional parameter is a key function, will check it soon.

- The second optional parameter is a Boolean value which determines whether to sort the items in reverse order. By default, it is False, but if you set it to True, the list will be sorted in reverse order.

```
1 L2 = ["python", "java", "Linux"]
2
3 print(sorted(L2, reverse=True))
4
```

```
['python', 'java', 'Linux']
```

***KEY POINT:**
There is also a possibility that instead of giving keyword parameters as we used reverse= True in above program, It is possible to provide the value True for the reverse parameter without naming that parameter, but then we would have to provide a value for the first parameter as well, rather than allowing the default parameter value to be used. We would have had to write sorted(L2, None, True). That's a lot harder for us to read and understand so try to stick with keyword parameters

Optional key parameter

We saw how to manipulate the sorting ordering by using 2nd Optional parameter but what if you want to sort things in some order other than **the "natural" or its reverse**, you can provide an **additional parameter**, the **key parameter**.

For example, suppose you want to sort a list of numbers based on their absolute value or square of the elements present in the list , so that -4 comes after 3 if it absolute value if you consider from lowest to highest or square of -2 comes after square of 1?

Or suppose you have a dictionary with strings as the keys and numbers as the values. Instead of sorting them in alphabetic order based on the keys, you might like to sort them in order based on their values.

```
L1 = [1, 7, 4, -2, 3]

def absolute(x): #defining absolute function which will return
                 #the positive value of any integer you feed
    if x >= 0:
        return x
    else:
        return -x

def square(x): # defining a function named square which will
               #return square of any number provided
    return x * x

print(square(3))
print(square(-119))
print(absolute(3))
print(absolute(-119)

#below call will call absolute function for every element in the list
print(sorted(L1,key=absolute))
#below call will call square function for every element in the list
print(sorted(L1,key=square))


```

```
9
14161
3
119
[1, -2, 3, 4, 7]
[1, -2, 3, 4, 7]
```

There are lot of things to learn from above small block of code:

- We defined a function absolute that takes a number and returns its absolute value. (Actually, python provides a built-in function abs that does this, but we are going to define our own to understand the functioning of key parameter.)
- We also defined another function square which returns square. We already have Exponent Operator (**) to square any value or use math.pow() but again using just to understand key param in a better way

- but during sorted function call, if you have a close look , you will find something pretty strange. Before, all the values we have passed as parameters have been pretty easy to understand: numbers, strings, lists, Booleans, dictionaries. **But here we have passed a function object**: absolute/square looks variable names whose **value is the function**. When we pass that function object, it is not automatically invoked. Instead, it is just bound to the formal parameter key of the function sorted.

- If you will go searching around on official python website, you will find somewhere in **sorted function code a parameter named key with** a **default value of None**. & when a value is provided for that parameter in an **invocation of the function sorted,** it has to be a **function**.

- What is sorted function doing in backend?
    - It **calls that key function once for each element** in the **list** that's getting sorted. It associates the result returned by that function (the absolute and square function in our case) with the original value.
    - Think of those associated values as being little **post-it notes** that decorate the original values.
    - The value 4 has a post-it note that says 4 on it for absolute function and 16 for square function, and the value of -2 has a post-it note that says 2 for absolute function and 4 for square function.

- Lastly, the sorted function rearranges the original items in order of the values written on their associated post-it notes.

*****KEY POINT**

It might be a little confusing that we are reusing the word **key** so many times.

The name of the optional parameter is key.

We will usually pass a parameter value using the keyword parameter passing mechanism. When we write **key=some_function** in the function invocation, the word key is there because it is the name of the parameter, specified in the definition of the sort function, **not because we are using keyword-based parameter passing.**

Question:

You will be sorting the following list by each element's second letter, a to z. Create a function to use when sorting, called second_let. It will take a string as input and return the second letter of that string. Then sort the list, create a variable called sorted_by_second_let and assign the sorted list to it. You can try to use lambda.

ex_lst = ['hi', 'how are you', 'bye', 'apple', 'zebra', 'dance']

Answer:

```
ex_lst = ['hi', 'how are you', 'bye', 'apple', 'zebra', 'dance']

def second_let(s):
    print(s[-1])
    return s[-1]

sorted_by_second_let=sorted(ex_lst,key=second_let)
print(sorted_by_second_let)
sorted_by_second_let=sorted(ex_lst, key=lambda ex_lst: second_let(ex_lst))
print(sorted_by_second_let)
```

Question:

Once again, sort the list nums based on the last digit of each number from highest to lowest. However, now you should do so by writing a lambda function. Save the new list as nums_sorted_lambda.

nums = ['1450', '33', '871', '19', '14378', '32', '1005', '44', '8907', '16']

answer:

nums_sorted_lambda =sorted(nums, key=lambda nums: nums[-1] , reverse=True)

Sorting a Dictionary

We know dictionary's keys are not sorted in any particular order. In fact, you may get a different order of output than someone else running the same code. We can force the results to be displayed in some fixed ordering, by sorting the keys.

```
L = ['E', 'F', 'B', 'A', 'D', 'I', 'I', 'C', 'B', 'A', 'D', 'D', 'E', 'D']
d={}
for i in L:
    if i not in d:
        d[i]=0
    d[i] = d[i] + 1

# we want to sort first as per the keys
sort_keys=sorted(d.keys())
print(sort_keys)
#we are sorting the dict and storing the key values in a variable as per its values
sort_values=sorted(d,key=lambda x:d[x])
print(sort_values)
for i in sort_values:
    print("alphabet {} appears {} times".format(i,d[i]))

```

```
['A', 'B', 'C', 'D', 'E', 'F', 'I']
['F', 'C', 'E', 'B', 'A', 'I', 'D']
alphabet F appears 1 times
alphabet C appears 1 times
alphabet E appears 2 times
alphabet B appears 2 times
alphabet A appears 2 times
alphabet I appears 2 times
alphabet D appears 4 times
```

*****KEY POINT:**
Here things get a little confusing because we have two different meaning of the word “key”. One meaning is a key in a dictionary. The other meaning is the parameter name for the function that you pass into the sorted function.

Remember that the key function always takes as input one item from the sequence and returns a property of the item. In above example , we have tried to sort the dictionaries' output as per :

- keys
- values

items to be sorted are the dictionary’s keys, so each item is one key from the dictionary.

*****KEY POINT**
When we sort the keys, passing a function with ***key=lambda x: d[x]*** *does not specify to sort the keys of a dictionary. The lists of keys are passed as the first parameter value in the invocation of sort. The key parameter provides a function that says how to sort them, whatever output the key parameter will return after individually going through all the elements of the list/dictionary .*

The function sorted is invoked. Its **first parameter value is a dictionary,** which **really means the keys of the dictionary**. The s**econd parameter**, the key function, **decorates the dictionary key with a post-it note** containing that key’s value in dictionary d.

Questions:
Which of the following will sort the keys of d in ascending order of their values (i.e., from lowest to highest)?

```
L = [4, 5, 1, 0, 3, 8, 8, 2, 1, 0, 3, 3, 4, 3]
d = {}
for x in L:
   if x in d:
      d[x] = d[x] + 1
   else:
      d[x] = 1
def g(k, d):
   return d[k]
ks = d.keys()
```

Options:
A. sorted(ks, key=g)
B. sorted(ks, key=lambda x: g(x, d))
C. sorted(ks, key=lambda x: d[x])

Breaking Ties: Second level Sorting

What happens when two items are “tied” in the sort order? For example, suppose we sort a list of words by their lengths. Which five letter word will appear first?

The answer is that the python interpreter will sort the tied items in the same order they were in before the sorting.

What if we wanted to sort them by some other property, say alphabetically, when the words were the same length? Python allows us to specify multiple conditions when we perform a sort by returning a tuple from a key function.

```
tups = [('A', 3, 2),
        ('C', 1, 4),
        ('B', 3, 1),
        ('A', 2, 4),
        ('C', 1, 2)]
for tup in sorted(tups):
    print(tup)
```

```
('A', 2, 4)
('A', 3, 2)
('B', 3, 1)
('C', 1, 2)
('C', 1, 4)
```

In below code, we are going to sort a list of fruit words first by their length, smallest to largest, and then alphabetically to break ties among words of the same length. To do that, we have the key function return a tuple whose first element is the length of the fruit's name, and second element is the fruit name itself.

```
fruits = ['peach', 'kiwi', 'apple', 'blueberry', 'papaya', 'mango', 'pear']
new_order =sorted(fruits, key=lambda fruit: (len(fruit),fruit) , reverse=True)
for i in new_order:
    print(i)
```

```
blueberry
papaya
peach
mango
apple
pear
kiwi
```

What will you do, if i ask to sort the words from largest to smallest, but also if in case 2 words have same length is should sort the words in reverse alphabetical order

```python
fruits = ['peach', 'kiwi', 'apple', 'blueberry', 'papaya', 'mango', 'pear']
new_order =sorted(fruits, key=lambda fruit: (len(fruit),fruit) , reverse=True)
new_order1 =sorted(fruits, key=lambda fruit: (-len(fruit),fruit) , reverse=False)
for i in new_order:
    print(i)
for i in new_order1:
    print(i)
```

```
blueberry
papaya
peach
mango
apple
pear
kiwi
blueberry
papaya
apple
mango
peach
kiwi
pear
```

What will the sorted function sort by?

weather = {'Reykjavik': {'temp':60, 'condition': 'rainy'},
'Buenos Aires': {'temp': 55, 'condition': 'cloudy'},
'Cairo': {'temp': 96, 'condition': 'sunny'},
'Berlin': {'temp': 89, 'condition': 'sunny'},
'Caloocan': {'temp': 78, 'condition': 'sunny'}}

problem1:
sorted_weather = sorted(weather, key=lambda w: (w, weather[w]['temp']))

problem2:
sorted_weather = sorted(weather, key=lambda w: (w, -weather[w]['temp']), reverse=True)

Lambda Expressions: When to use them

Though you can often use a lambda expression or a named function interchangeably when sorting. it's generally best to use lambda expressions until the process is too complicated, and then a function should be used.

Suppose we want to sort the states in order by the length of the first city names:

states = {"Minnesota": ["St. Paul", "Minneapolis", "Saint Cloud", "Stillwater"],
"Michigan": ["Ann Arbor", "Traverse City", "Lansing", "Kalamazoo"],
"Washington": ["Seattle", "Tacoma", "Olympia", "Vancouver"]}

```
states = {"Minnesota": ["St. Paul", "Minneapolis", "Saint Cloud", "Stillwater"],
          "Michigan": ["Ann Arbor", "Traverse City", "Lansing", "Kalamazoo"],
          "Washington": ["Seattle", "Tacoma", "Olympia", "Vancouver"]}

print(sorted(states,key=lambda state: len(states[state][0]) ))
```

```
['Washington', 'Minnesota', 'Michigan']
```

For our second sort order question, the property we want to sort by is the number of cities that begin with the letter 'S'.

We would need to write a function defining this property is harder to express, requiring a filter and count accumulation pattern in lambda expression. So we are better off defining a separate, named function.

```
def s_cities_count(city_list):
    ct = 0
    for city in city_list:
        if city[0] == "S":
            ct += 1
    return ct

states = {"Minnesota": ["St. Paul", "Minneapolis", "Saint Cloud", "Stillwater"],
          "Michigan": ["Ann Arbor", "Traverse City", "Lansing", "Kalamazoo"],
          "Washington": ["Seattle", "Tacoma", "Olympia", "Vancouver"]}

print(sorted(states, key=lambda state: s_cities_count(states[state])))

```

```
['Michigan', 'Washington', 'Minnesota']
```

- we've chosen to make a lambda expression that looks up the value associated with the particular state and pass that value to the named function s_cities_count.

- We could have passed just the key, but then the function would have to look up the value, and it would be a little confusing, from the code, to figure out what dictionary the key is supposed to be looked up in.

- Here, we've done the lookup right in the lambda expression, which makes it a little bit clearer that we're just sorting the keys of the states dictionary based on a property of their values. It also makes it easier to reuse the counting function on other city lists, even if they aren't embedded in that particular states dictionary.

PART 8