# S206E057 -- Lecture 20, 5/1/2018, Python – condition and loops

This lecture is the beginning on operating logic in Python. There are three types of logical operations available in Python: conditional testing of **if-then-else** clause, loops of "**While**" and "**For**". These conditional statements are used to compare and evaluate variables or values, which are parts of human reasoning. In Python, conditions are handled by IF statements.

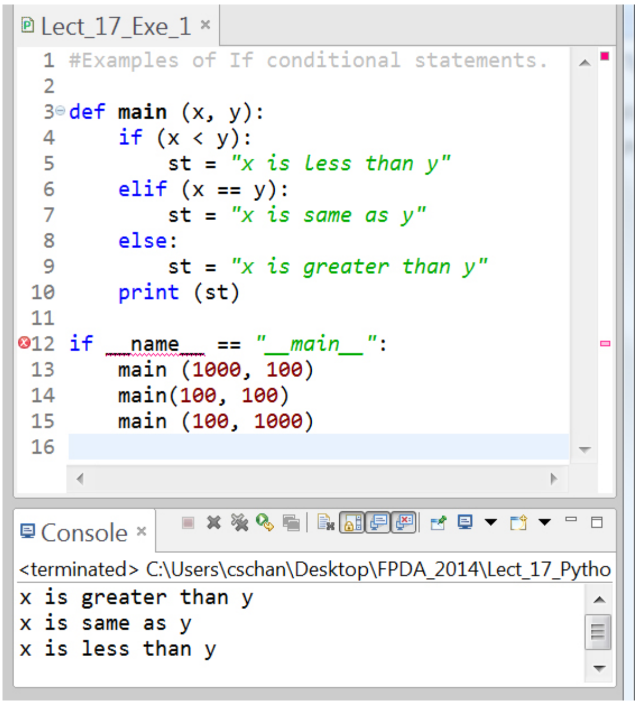
## Logical Operation: If-then-else

The if-else format is:

```
if (conditions):
    print ("True")
else:
    print ("False")
```

or

```
if (conditions):
    return ("True")
else:
    return ("False")
```



```
#Examples of If conditional statements.

def main (x, y):
    if (x < y):
        st = "x is less than y"
    elif (x == y):
        st = "x is same as y"
    else:
        st = "x is greater than y"
    print (st)

if __name__ == "__main__":
    main (1000, 100)
    main(100, 100)
    main (100, 1000)
```

<terminated> C:\Users\cschan\Desktop\FPDA_2014\Lect_17_Pytho

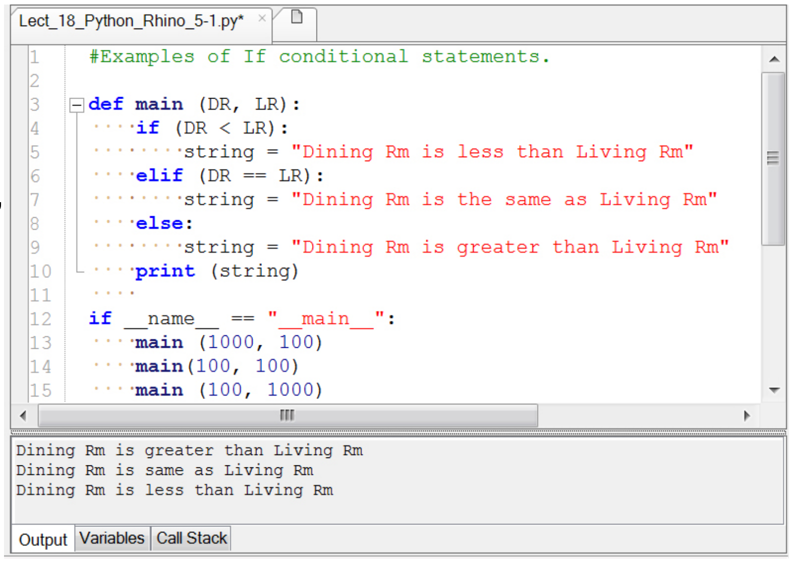
```
x is greater than y
x is same as y
x is less than y
```

The example given on the right provides some explanations. Usually "=" sign is used to assign values to a variable. Yet, in the example on the right, there is a == sign used, which is the evaluation sign to evaluate whether x is equivalent to y. If x is equivalent to y, then a true value is returned, or a false value otherwise. "!=" is the not equal sign (Note: "<>" has also been used as not equal sign.)

An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to "0" or a "false" value. The *else* statement is an optional statement and there could be at most only one **else** statement following **if.**

Alternatively, there is **elif** statement, which allows us to check multiple expressions for truth value and execute a block of code as soon as one of the conditions evaluates to be true. Like the **else**, the **elif** statement is optional.

Unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if** and before the **else. Else** will serve as the last statement or the stopping statement. There is another shorten version of **if-else**, see the following example.



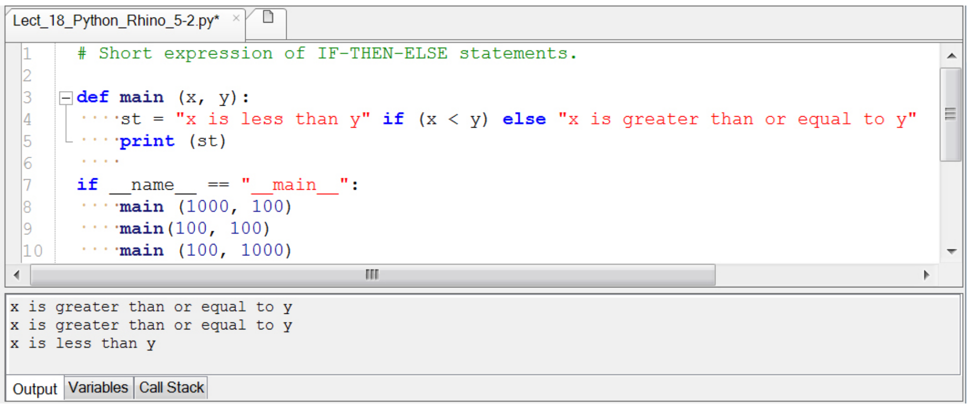
```
#Examples of If conditional statements.

def main (DR, LR):
    if (DR < LR):
        string = "Dining Rm is less than Living Rm"
    elif (DR == LR):
        string = "Dining Rm is the same as Living Rm"
    else:
        string = "Dining Rm is greater than Living Rm"
    print (string)

if __name__ == "__main__":
    main (1000, 100)
    main(100, 100)
    main (100, 1000)
```

```
Dining Rm is greater than Living Rm
Dining Rm is same as Living Rm
Dining Rm is less than Living Rm
```

The other concise way of writing a conditional IF-ELSE statement is to use the format of "a if C else b". Here, the conditions are limited to two cases of if something is true then do something, otherwise do something else. For example:

```
def main (x, y):
    st = "x is less than y" if (x < y) else "x is greater than or equal to y"
    print (st)

if __name__ == "__main__":
    main (1000, 100)
    main(100, 100)
    main (100, 1000)
```

This function call is the major call to implement the execution of the function main three times with different argument inputs.



Lect_18_Python_Rhino_5-2.py*

```
# Short expression of IF-THEN-ELSE statements.

def main (x, y):
    st = "x is less than y" if (x < y) else "x is greater than or equal to y"
    print (st)
    
if __name__ == "__main__":
    main (1000, 100)
    main(100, 100)
    main (100, 1000)
```

```
x is greater than or equal to y
x is greater than or equal to y
x is less than y
```

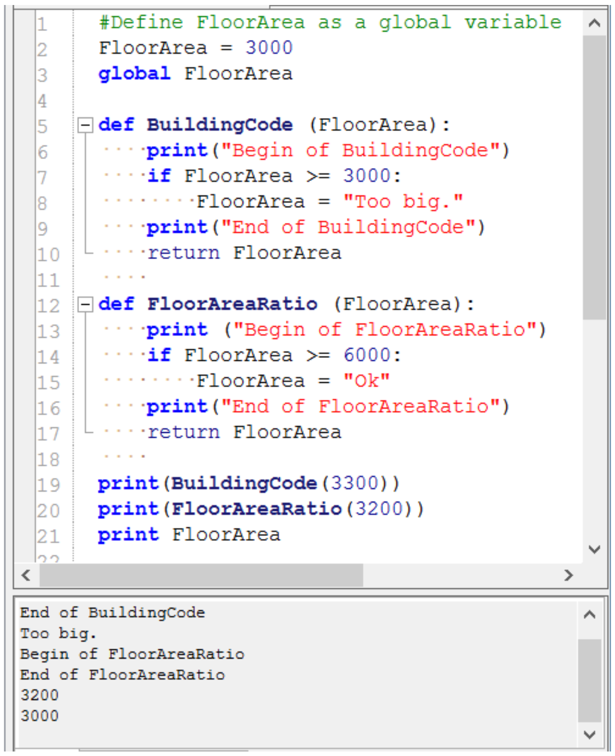
## Another global & local variable example and coding techniques:

In this example, I used the strings of “Begin of BuildingCode” and “Begin of FloorAreaRatio” as markers to indicate the beginning and end of each function. This is the technique for debugging.

The function of BuildingCode redefines the value of FloorArea from the integer of 3000 to a string of “Too Big”. When it completes the execution, the variable FloorArea will be returned and its value of “Too Big” will be passed out to the print command and shown in the display area.

For the FloorAreaRatio function, the variable FloorArea has been redefined to the string of “OK” which is printed as output. Then after it is completed, it is done with no value left. Thus, when we use the print command to call this function, it has noting provided from executing the function, thus the output is none.

After completing the two function calls, the FloorArea is back to its original value and print as 3000. In fact, it is treated as global variable for it is defined in the beginning without being modified outside the functions.



```
#Define FloorArea as a global variable
FloorArea = 3000
global FloorArea

def BuildingCode (FloorArea):
    print("Begin of BuildingCode")
    if FloorArea >= 3000:
        FloorArea = "Too big."
    print("End of BuildingCode")
    return FloorArea
    
def FloorAreaRatio (FloorArea):
    print ("Begin of FloorAreaRatio")
    if FloorArea >= 6000:
        FloorArea = "Ok"
    print("End of FloorAreaRatio")
    return FloorArea
    
print(BuildingCode(3300))
print(FloorAreaRatio(3200))
print FloorArea
```

```
End of BuildingCode
Too big.
Begin of FloorAreaRatio
End of FloorAreaRatio
3200
3000
```

## Loops

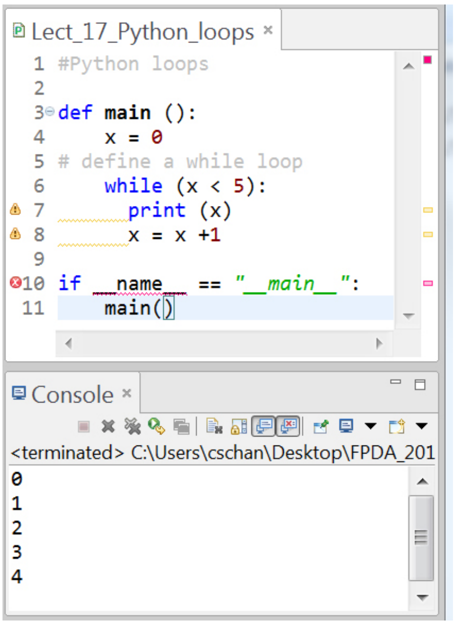
Repeating code over and over again via a construct known as a loop is also a fairly common technique in programming. Most programming language applies FOR, and WHILE loops. Examples of defining while and for loops with different input methods are shown here.

### While loop

Here is the example of setting up the x value to 0, and it runs five times and prints the value respectively.

*Example 1:*

This example determines the loop execution by the index number of 5.



Lect_17_Python_loops

```
#Python loops

def main ():
    x = 0
# define a while loop
    while (x < 5):
        print (x)
        x = x +1

if __name__ == "__main__":
    main()
```

Console

```
<terminated> C:\Users\cschan\Desktop\FPDA_201
0
1
2
3
4
```

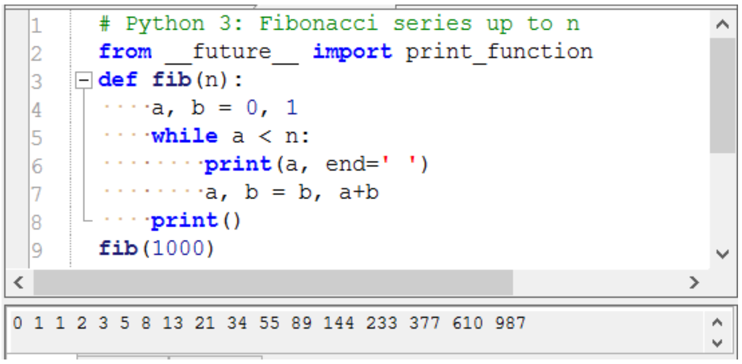
If x<5 is true, then it will print the x value and increment its value by one and keep comparing its new value to test whether it is less than five or not. We could also control the number of running the while loop by entering the number as input. Here is the second example.

*Example 2:*

```
#Python loops

def main (y):
   x = 0
# define a while loop
   while (x < y):
     print (x)
     x = x + 1

if __name__ == "__main__":
   main(5)
```

In this instance, the input value of 5 is used to control how many loops it will go through. So x will be used to compare whether it is less than 5, which serves as a control value. In the previous example 1, the while loop is controlled by the built-in index number of 5, which is pre-defined and not flexible. In the second example, the call function provides a data input of 5, which is the data decided by the function call. Users could define the number of repetitions for the while loop five times. Note: While loop is also a dangerous function if you have not controlled it well. A hint, if it ends up as an infinite execution without knowing when to stop or the window is frozen, then close the Python platform (sometimes the Rhino system) and restart it.

Example 3: Fibonacci series generation up to n:



```
# Python 3: Fibonacci series up to n
from __future__ import print_function
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
fib(1000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

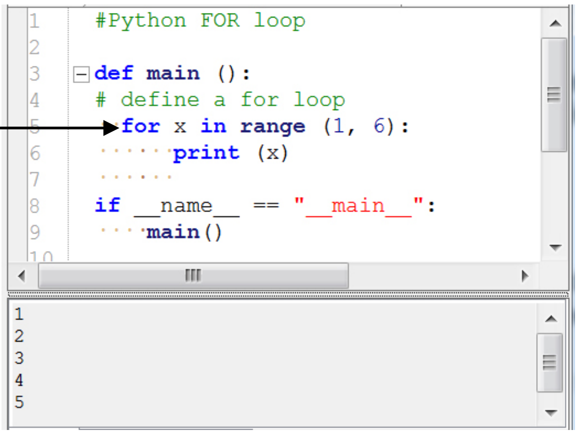
```
def fib(n):
   a, b = 0, 1
   while a < n:
       print(a)
       a, b = b, a+b
fib(1000)
```

## For loop

Python's FOR loops are called iterators. If we want to have x loop repeated over a range of numbers, we shall use the built-in function of range.

Example 1:

The first example of "FOR" loop is defined from range 1 to 6. Here, actions will be repeated from 1 to 5 and stop at 6.



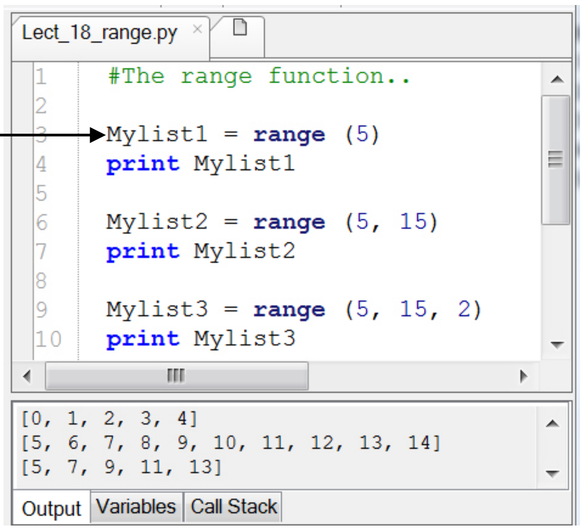
```
#Python FOR loop

def main ():
# define a for loop
    for x in range (1, 6):
        print (x)

if __name__ == "__main__":
    main()
```

```
1
2
3
4
5
```

The build in function of range has the following three activities.

1. Range (stop)
   It shows the end number. The number shall be integers.
2. Range(start, stop)
   It shows the beginning and the end number.
3. Range(start, stop, step)
   It shows the beginning, end, and the number of steps to pass over.

To better understand the function of range is to make a list and print the list as shown below.


Lect_18_range.py

```
#The range function..

Mylist1 = range (5)
print Mylist1

Mylist2 = range (5, 15)
print Mylist2

Mylist3 = range (5, 15, 2)
print Mylist3
```

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[5, 7, 9, 11, 13]
```

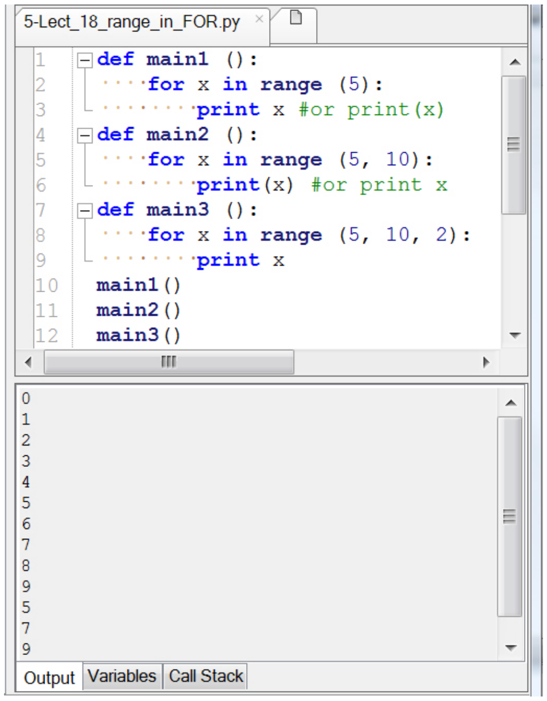
The first function indicates the end integer of 5 and stop at 5. But, the beginning of this range is 0 by default.

The second function shows the number from 5 to 10. 5 is the beginning and 10 is the end. So, it prints the list from 5 to 9 and stop at 10.

The third list is to start from 5, end at 15 with the interval of two steps. So, it prints from 5, 6…up to 14. It jumps over 14 and stops at 15. On the other hand, if the beginning number is 6, then it will provide a list of even numbers instead of odd numbers. In Python, the list is always displayed by a pair of angled parenthesis [].

Example 2: combining range with FOR loop.

In this example, the function of range is inside the FOR loop to generate a set of data. Coding provided on the right explains the three cases of range function implemented by the FOR function.


5-Lect_18_range_in_FOR.py

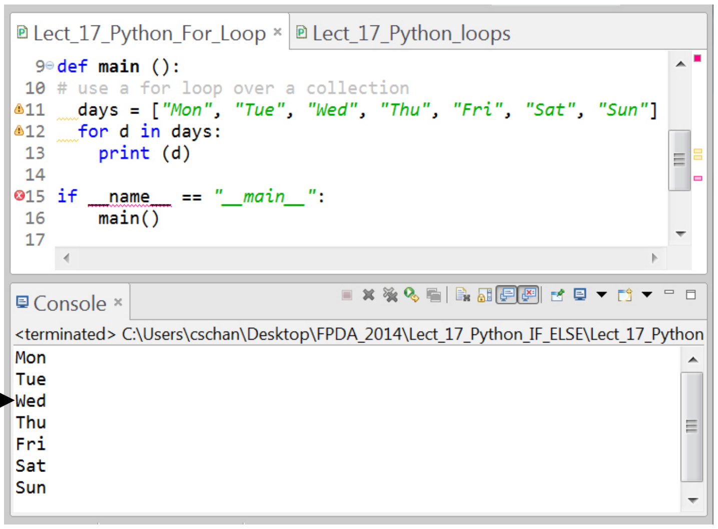
```
def main1 ():
    for x in range (5):
        print x #or print(x)
def main2 ():
    for x in range (5, 10):
        print(x) #or print x
def main3 ():
    for x in range (5, 10, 2):
        print x
main1()
main2()
main3()
```

```
0
1
2
3
4
5
6
7
8
9
5
7
9
```

Here the list shows the number from 0 to 4, in which, 0 is the beginning and 4 is the end in function main1. So it prints the data from 0 to 4 and stops. The second function of main2 is to print the data individually from 5 to 9. The third function is to make an interval of 2 steps. So, it prints from 5, 7 up to 9 and stops.

Example 3: Use FOR loop for a list argument.

```
def main ():
# use a for loop over a collection
  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  for d in days:
    print (d)
if __name__ == "__main__":
    main()
```



```
def main ():
# use a for loop over a collection
  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  for d in days:
    print (d)

if __name__ == "__main__":
    main()
```

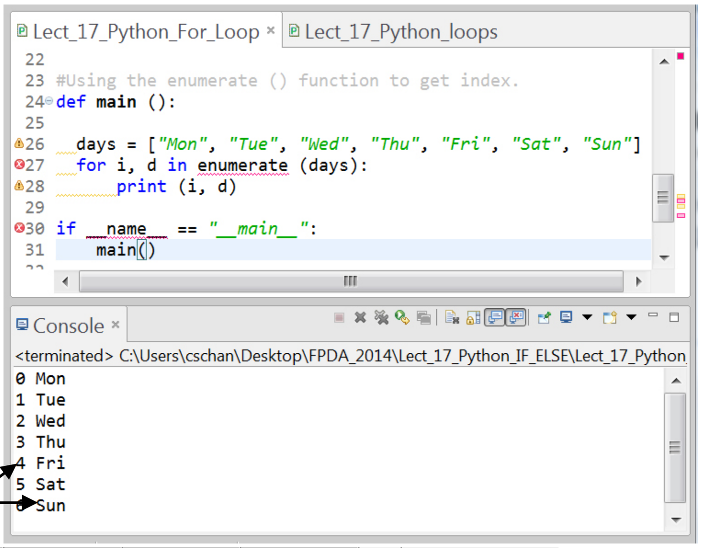
```
<terminated> C:\Users\cschan\Desktop\FPDA_2014\Lect_17_Python_IF_ELSE\Lect_17_Python
Mon
Tue
Wed
Thu
Fri
Sat
Sun
```

In this example, **days** is a list and while **d** is IN the list, then print each of them sequentially. There has no index counter involved to guide the progress. It just iterates over the members of a list.

We could also use the built-in Python function of enumerate() to get an object that is a list of tuples (immutable lists) consisting of a pair of count/index and value. Thus, we use **i** to indicate the first element of index number on the list. Then Python will go through the list and return its index number together with the items back. Here is the example:

```
#Using the enumerate () function to get index.
def main ():

 days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
 for i, d in enumerate (days):
   print (i, d)

if __name__ == "__main__":
  main()
```

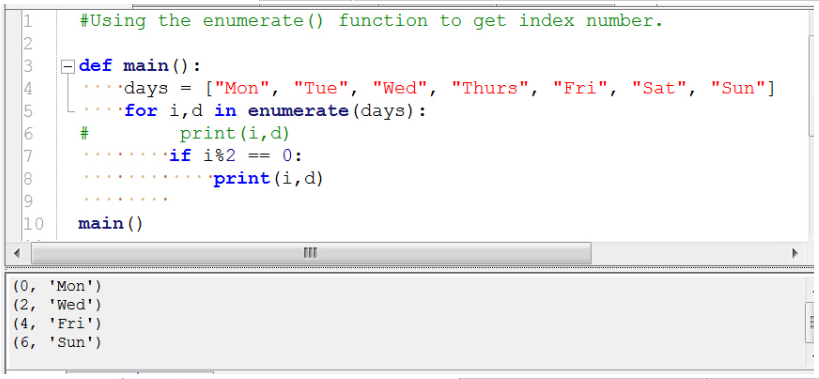
In result, the print will print the day and the index number together.



```
#Using the enumerate () function to get index.
def main ():

    days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
    for i, d in enumerate (days):
        print (i, d)

if __name__ == "__main__":
    main()
```

```
<terminated> C:\Users\cschan\Desktop\FPDA_2014\Lect_17_Python_IF_ELSE\Lect_17_Python
0 Mon
1 Tue
2 Wed
3 Thu
4 Fri
5 Sat
6 Sun
```

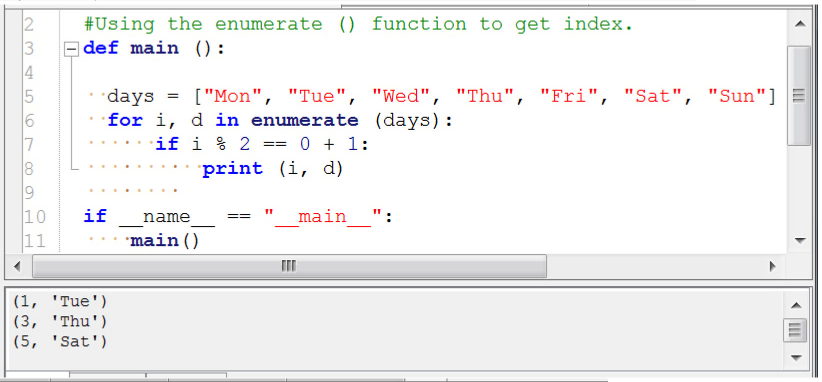
If we want to only show the result of **even number** of the index, then use modulo % sign to work it out. The % (modulo) operator yields the remainder from the division of the first argument by the second. Here, it is the remainder of the number of I divided by 2.



```
#Using the enumerate() function to get index number.

def main():
    days = ["Mon", "Tue", "Wed", "Thurs", "Fri", "Sat", "Sun"]
    for i,d in enumerate(days):
#           print(i,d)
        if i%2 == 0:
            print(i,d)

main()
```

```
(0, 'Mon')
(2, 'Wed')
(4, 'Fri')
(6, 'Sun')
```

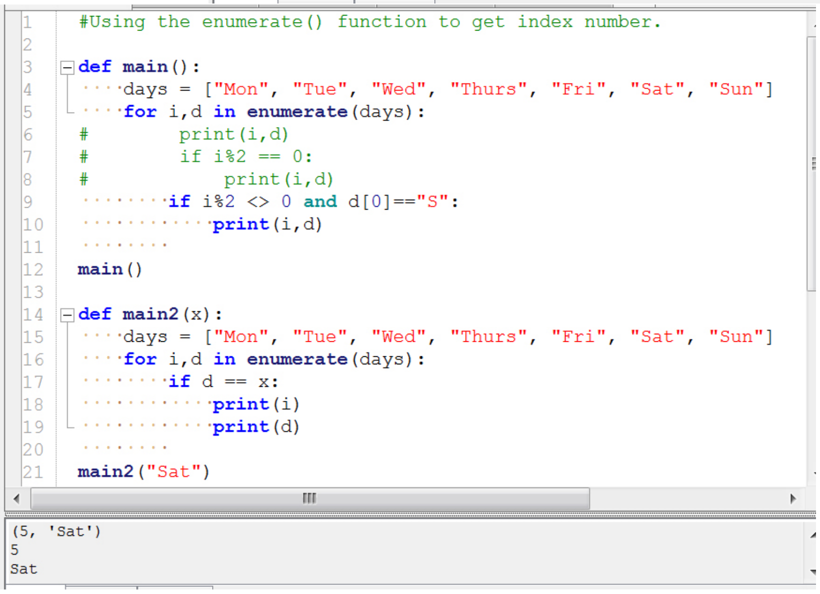
To show **odd number**:



```
#Using the enumerate () function to get index.
def main ():

  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  for i, d in enumerate (days):
      if i % 2 == 0 + 1:
          print (i, d)

if __name__ == "__main__":
   main()
```

```
(1, 'Tue')
(3, 'Thu')
(5, 'Sat')
```

To only show Saturday:

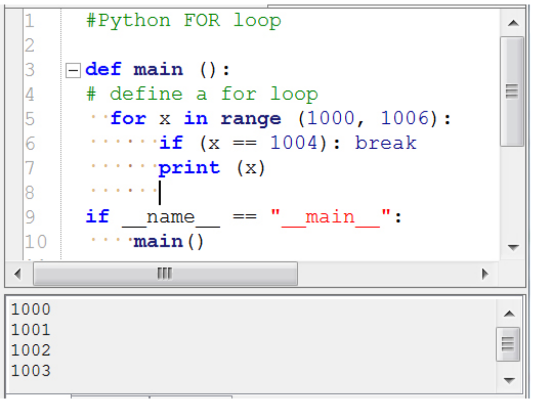


```
#Using the enumerate() function to get index number.

def main():
    days = ["Mon", "Tue", "Wed", "Thurs", "Fri", "Sat", "Sun"]
    for i,d in enumerate(days):
#           print(i,d)
#           if i%2 == 0:
#               print(i,d)
        if i%2 <> 0 and d[0]=="S":
            print(i,d)

main()

def main2(x):
    days = ["Mon", "Tue", "Wed", "Thurs", "Fri", "Sat", "Sun"]
    for i,d in enumerate(days):
        if d == x:
            print(i)
            print(d)

main2("Sat")
```

```
(5, 'Sat')
5
Sat
```

## Break and continue functions

The “For” loop execution, if necessary, could be stopped at some number by the break statement. For instance, when the break number is 1004 within the range of 1000 to 1006, it will not execute anything after 1003. So, the break statement is used to break the execution of the loop. If a condition is met, then the loop stops and it will go to the next execution line. In this case, it is the end of the if function. There is another function of “pass” to skip the remaining statements.

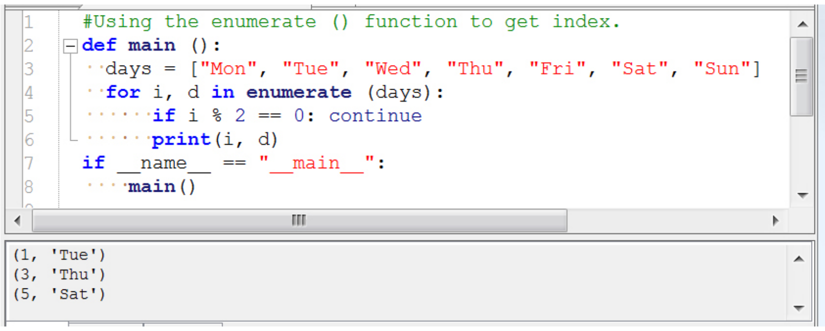


```
#Python FOR loop

def main ():
  # define a for loop
  for x in range (1000, 1006):
      if (x == 1004): break
      print (x)

if __name__ == "__main__":
   main()
```

```
1000
1001
1002
1003
```

Another function of continue in the FOR loop:

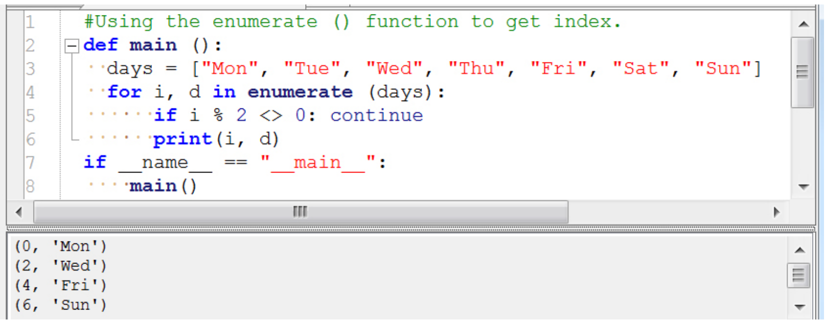
On the contrast to break, the function of “continue” will skip the action and jump to the next index. For instance, in the first coding example on the upper right, whenever the modulo is 0, it would skip the execution of the rest of the statements in the if component. In this case, if modulo equals 0, (in other words, take the index of i divide by 2 and if the left over value equals 0), then continue and skip the rest of the statements in the if function (here is the print statement). Therefore, it skips the even number and prints only odd index numbers.



```
#Using the enumerate () function to get index.
def main ():
  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  for i, d in enumerate (days):
      if i % 2 == 0: continue
      print(i, d)
if __name__ == "__main__":
   main()
```

```
(1, 'Tue')
(3, 'Thu')
(5, 'Sat')
```

The second example, on the contrary, will skip the odd number and just print even numbers. It of course is the other method of providing odd and even number outputs.



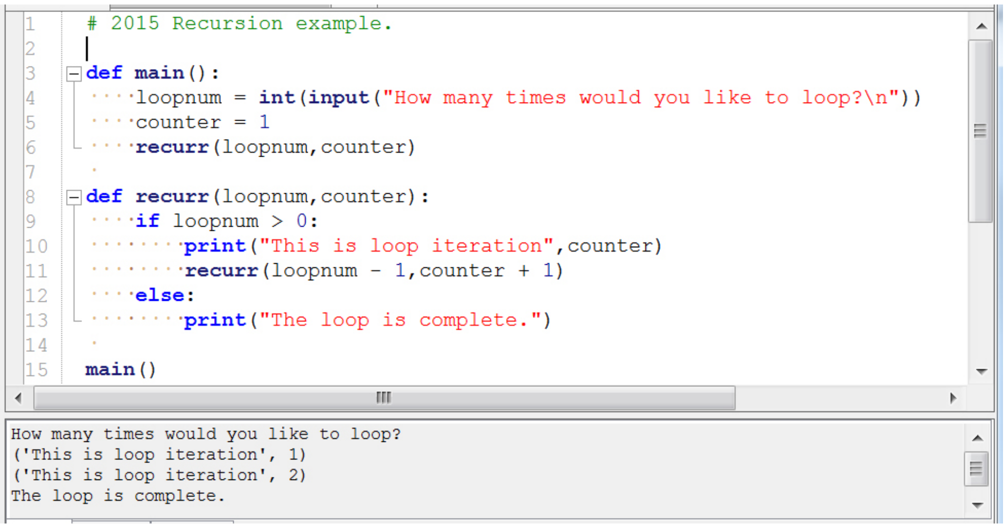
```
#Using the enumerate () function to get index.
def main ():
  days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
  for i, d in enumerate (days):
      if i % 2 <> 0: continue
      print(i, d)
if __name__ == "__main__":
   main()
```

```
(0, 'Mon')
(2, 'Wed')
(4, 'Fri')
(6, 'Sun')
```

## Concept of recursion:

In Python, we have functions call other functions. However, it is possible for a function to call itself. For example, counter function calls itself here.

```
num = 0

def main():
  counter(num)

def counter(num):
  print(num)
  num += 1
  counter(num)

main()
```



```
# 2015 Recursion example.

def main():
    loopnum = int(input("How many times would you like to loop?\n"))
    counter = 1
    recurr(loopnum,counter)

def recurr(loopnum,counter):
    if loopnum > 0:
        print("This is loop iteration",counter)
        recurr(loopnum - 1,counter + 1)
    else:
        print("The loop is complete.")

main()
```

```
How many times would you like to loop?
('This is loop iteration', 1)
('This is loop iteration', 2)
The loop is complete.
```

If you were to run this program, it would run forever. This is the example of an infinite recursion. The only way to stop the loop would be to interrupt the execution by pressing Ctrl+C on your keyboard or close the Python script window. But, recursion could be controlled, for instance, by arguments/parameters to determine the number of recursions.

```
def main():
    loopnum = int(input("How many times would you like to loop?\n"))
    counter = 1
    recurr(loopnum,counter)

def recurr(loopnum,counter):
    if loopnum > 0:
        print("This is loop iteration",counter)
        recurr(loopnum - 1,counter + 1)
    else:
        print("The loop is complete.")

main()
```

## Applications of Recursion

Often, recursion is studied at an advanced computer science level. Recursion is usually used to solve complex problems that can be broken down into smaller, identical problems. Recursion isn't required to solve a problem. Problems that can be solved with recursion, most likely can be solved with loops. Also, a loop may be more efficient than a recursive function. Recursive functions require more memory, and resources than loops, making them less efficient in a lot of cases. This usage requirement is sometimes referred to as *overhead*. You might be thinking, "Well why bother with recursion. I'll just use a loop. I already know how to loop and this is a lot more work." This thought is understandable, but not entirely ideal. When solving complex problems, a recursive function is sometimes easier, faster, and simpler to build and code.

Think of these two "rules":

- If I can solve the problem now, without recursion, the function simply returns a value.
- If I cannot solve the problem now without recursion, the function reduces the problem to something smaller and similar, and calls itself to solve the problem.

Let's apply this using a common mathematical concept, factorials. If you are unfamiliar with factorials in mathematics, please refer to the following reading.

The factorial of a number *n*, is denoted as *n!* Here are some basic rules of factorials.

If n = 0 then n! = 1 If n > 0 then n! = 1 x 2 x 3 x ... x n

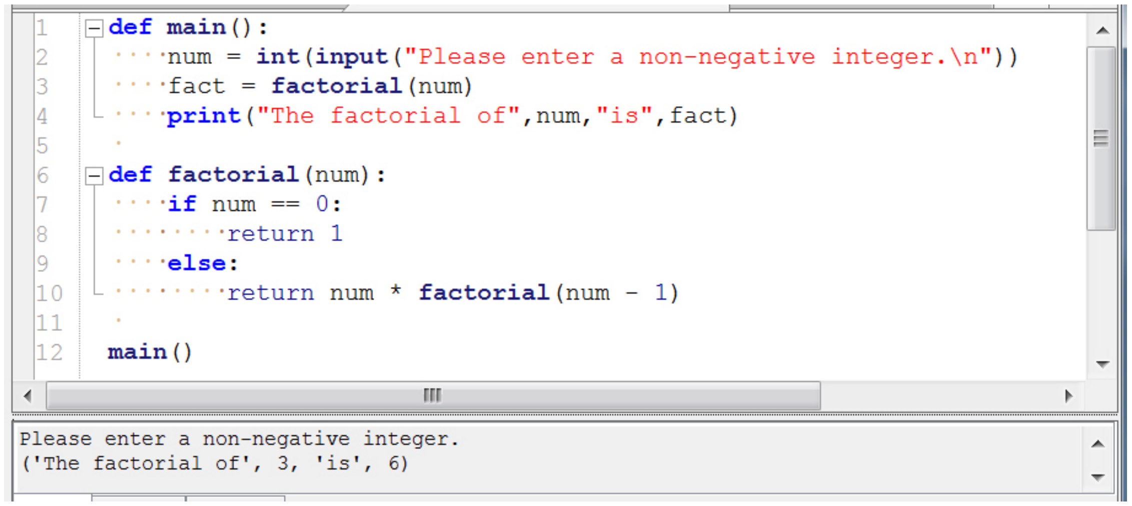
For example, let's look at the factorial of the number 9.

9! = 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9

Let's look at a program which calculates the factorial of any number, entered by the user, by method of recursion.

```
def main():
    num = int(input("Please enter a non-negative integer.\n"))
    fact = factorial(num)
    print("The factorial of",num,"is",fact)

def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)

main()
```

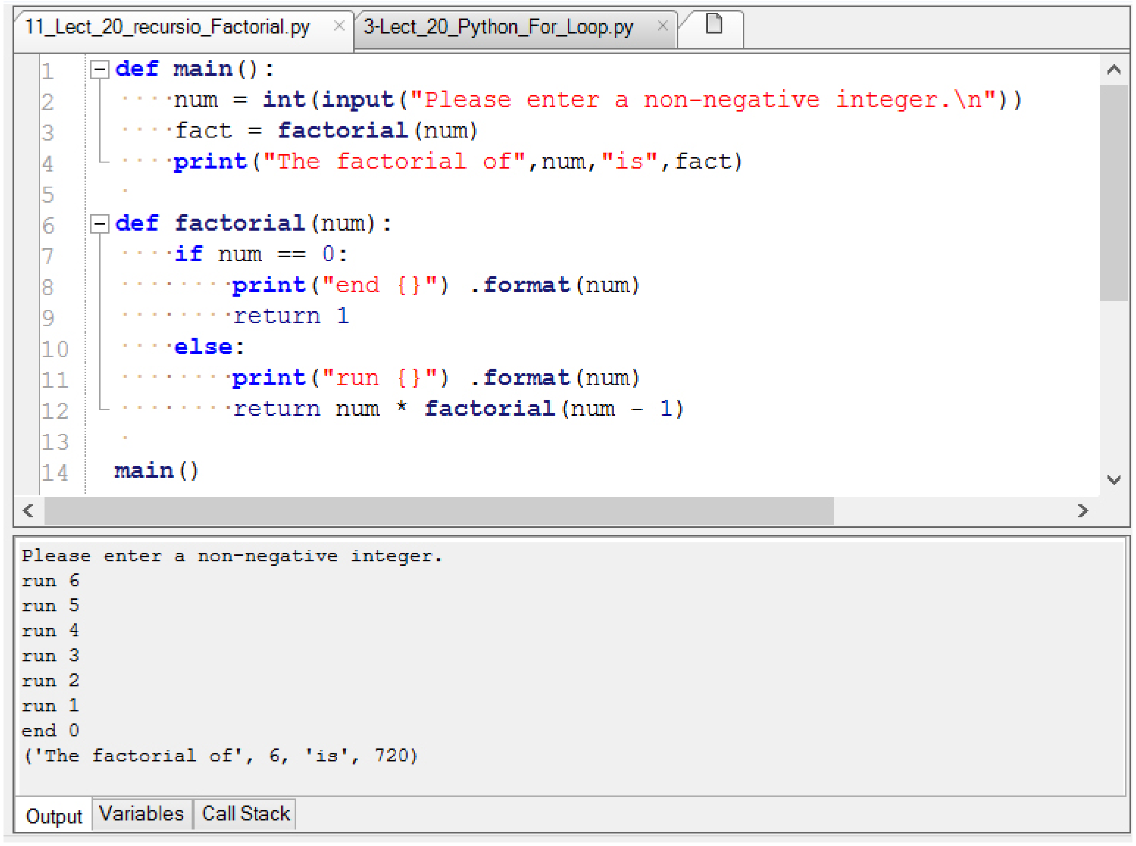
Execution results:



```python
def main():
    num = int(input("Please enter a non-negative integer.\n"))
    fact = factorial(num)
    print("The factorial of",num,"is",fact)

def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)

main()
```

```
Please enter a non-negative integer.
('The factorial of', 3, 'is', 6)
```

Use the print method to understand each run in recursion.



11_Lect_20_recursio_Factorial.py | 3-Lect_20_Python_For_Loop.py

```python
def main():
    num = int(input("Please enter a non-negative integer.\n"))
    fact = factorial(num)
    print("The factorial of",num,"is",fact)

def factorial(num):
    if num == 0:
        print("end {}") .format(num)
        return 1
    else:
        print("run {}") .format(num)
        return num * factorial(num - 1)

main()
```

```
Please enter a non-negative integer.
run 6
run 5
run 4
run 3
run 2
run 1
end 0
('The factorial of', 6, 'is', 720)
```