Examples

```
name="Sachin"
age=36
print("My name is {1} and my age is {0}".format(age,name))
```

- **Output:**

My name is Sachin and my age is 36

PYTHON

LECTURE 16

Today's Agenda

- **Decision Control Statements**
  - The if Statement
  - Concept of Indentation
  - The if-else Statement
  - The if-elif-else Statement
  - What about ternary operator ?

Decision Control Statements

- **Decision Control Statements** in **Python** are those statements which decide the execution flow of our program.

- In other words , they allow us to decide whether a **particular part of our program** should **run** or **not** based upon certain condition.

- The 4 **decision control statements** in **Python** are:
  - **if**
  - **if….else**
  - **if…elif…else**
  - **nested if**

The if Statement

- The **if** the statement in **Python** is similar to other languages like in **Java**, **C**, **C++**, etc.
- It is used to **decide** whether a **certain statement** or **block of statements** will be executed or not .
- If a certain condition is **true** then a **block of statement** is **executed** otherwise **not**.

The if Statement

- **Syntax:**

```
if (expression):
        statement1
        statement2
        .
        .
        statement..n
```

- **Some Important Points:**
  - Python does not use { } to define the body of a code block , rather it uses indentation.
  - A code block starts with indentation and ends with the first unindented line.
  - The amount of indentation is up to the programmer, but he/she must be consistent throughout that block.
  - The colon after if( ) condition is important and is a part of the syntax. However parenthesis with condition is optional

Exercise

- WAP to accept an integer from the user and check whether it is an even or odd number.

Solution

Solution 1:

```
a=eval(input("Enter a number:"))
if(a%2==0):
    print("No is even")
if(a%2!=0):
    print("No is odd")
```

If the body of if( ) statement contains only one statement , then we can write it just after if( ) statement also

Solution 2:

```
if(a%2==0):print("No is even")
if(a%2!=0):print("No is odd")
```

What About Multiple Lines ?

- If there are **multiple lines** in the body of **if( )** , then :
  - **Either we can write them inside if( ) by properly indenting them**

OR

  - **If we write them just after if ( ) , then we must use semicolon as a separator**

What About Multiple Lines ?

Solution 1:

```
if(a%2==0):
    print("No is even")
    print("Hello")
if(a%2!=0):
    print("No is odd")
    print("Hi")
```

Solution 2:

```
if(a%2==0): print("No is even");print("Hello")
if(a%2!=0):  print("No is odd");print("Hi")
```

The if –else Statement

- The **if..else** statement evaluates **test expression** and will execute body of **if** only when **test condition** is **True**.
- If the **condition** is **False**, body of **else** is executed.
- **Indentation** is used to **separate** the blocks.

The if-else Statement

- **Syntax:**

```
if (expression):
        statement 1
        statement 2
else:
        statement 3
        statement 4
```

Indentation and **colon** are important for **else** also

Example

```
a=eval(input("Enter a number:"))
if(a%2==0):
    print("No is even")
else:
    print("No is odd")
```

Exercise

- WAP to accept a character from the user and check whether it is a capital letter or small letter. Assume user will input only alphabets

Solution

Solution 1:

```
ch=input("Enter a character:")
if  "A"<=ch<="Z":
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

We also can use the logical and operator and make the conditions separate

Solution 2:

```
ch=input("Enter a character:")
if ch>="A" and ch<="Z":
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

Guess The Output

Code:

```
ch=input("Enter a character:")
if  65<=ch<=90:
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

Suppose the input given is A

Output:

TypeError: <= not supported between int and str

Why Did The Exception Occur ?

- Recall that, in **Python** we don't have **character data type** and even single letter data is a **string**.
- So the input **"A"** , is **not converted** to **UNICODE** automatically because it is still treated as a **string** value.
- Thus , the comparison failed between **string** and **integer**.

Solution

- The solution is to convert the **"A"** to it's corresponding **UNICODE** value .
- **Can you think , how can we do it ?**
- The answer is , using the function **ord( )**.
- Recall that , this function accepts a **single letter string** and returns it's **UNICODE** value

Solution

```
ch=input("Enter a character:")
if  65<=ord(ch)<=90:
    print("You entered a capital letter")
else:
    print("You entered a small letter")
```

The if –elif-else Statement

- The **elif** is short for **else if**. It allows us to check for multiple expressions.
- If the condition for **if** is **False**, it checks the condition of the next **elif** block and so on.
- If all the conditions are **False**, body of **else** is executed.

The if –elif-else Statement

- **Syntax:**

```
if (expression):
        statement 1
        statement 2
elif (expression):
        statement 3
        statement 4
else:
        statement 5
        statement 6
```

Although it is not visible in the syntax , but we can have multiple **elif** blocks with a single **if** block

Exercise

- WAP to accept a character from the user and check whether it is a capital letter or small letter or a digit or some special symbol

Solution

```
ch=input("Enter a character:")
if  "A" <=ch <="Z":
    print("You entered a capital letter")
elif  "a" <=ch <="z":
    print("You entered a small letter")
elif  "0" <=ch <="9":
    print("You entered a digit")
else:
    print("You entered some symbol")
```

The nested if Statement

- We can have a **if…elif…else** statement inside another **if…elif…else** statement.
- This is called **nesting** in computer programming.
- Any number of these statements can be nested inside one another.
- **Indentation is the only way to figure out the level of nesting**

The nested if Statement

- **Syntax:**

```
if (expression):
        if (expression):
                statement 1
                statement 2
        else:
                statement 3
                statement 4
        statement 5
        statement 6
```

Exercise

- WAP to accept 3 integers from the user and without using any logical operator or cascading of relational operators , find out the greatest number amongst them

Solution

```
a,b,c=input("Enter 3 int").split()
a=int(a)
b=int(b)
c=int(c)
if a>b:
    if a>c:
            print("{o} is greatest".format(a))
    else:
            print("{o} is greatest".format(c))
else:
    if b>c:
            print("{o} is greatest".format(b))
    else:
            print("{o} is greatest".format(c))
```

Exercise

- WAP to accept an year from the user and check whether it is a leap year or not.

An year is a leap year if:
It is exactly divisible by 4 AND at the same time
not divisible by 100
OR
it is divisible by 400

For example:
2017 is not a leap year
2012 is a leap year
1900 is a not leap year
2000 is a leap year

Ternary Operator In Python

- Many **programming languages** have an **operator** called **ternary operator**, which is denoted by **? :** .
- It allows us to write complete **if – else** statement in just **one line.**
- For example , **C language** provides the following form of **ternary operator**:
  - **<condition> ? <expression1> : <expression2>**

Ternary Operator In Python

- But in **Python** we don't have a **ternary operator** officially .
- However **Python** provides us a **single line if – else** to work just like **ternary operator**
- **Syntax:**
  - **<expression1> if <condition> else <expression2>**
- It first evaluates the **condition**; if it returns **True** then **expression1** will be evaluated to give the result, otherwise it will evaluate **expression2**.

Example

Example 1:

```
age=12
msg='Kid' if age<13 else 'Teenager'
print(msg)
```

Output:

Kid

Example 2:

```
age=16
msg='Kid' if age<13 else 'Teenager'
print(msg)
```

Output:

Teenager

These codes internally become:

```
if age<13:
   msg='Kid'
else:
   msg='Teenager'
```

Example

Checking Even Odd Using Single Line if-else

```
a=eval(input("Enter a number:"))
msg= 'Even no' if a%2==0 else 'Odd No'
print(msg)
```

Output:

```
Enter a number:4
Even no
```

```
Enter a number:9
Odd No
```

Handling Multiple Conditions

- We can handle **multiple conditions** also **using single line if-else**
- **Syntax:**
  - **<expression1> if <condition1> else <expression2> if <condition2> else <expression3>**
- It first evaluates the **condition1**; if it returns **True** then **expression1** will be executed to give the result, otherwise it will evaluate the **condition2**; if it returns **True** then **expression2** will be executed otherwise it will execute **expression3.**

Example

Example 1:

```
age=16
msg='Kid' if age<13 else 'Teenager' if age<20  else 'Adult'
print(msg)
```

Output:

Teenager

```
if age<13:
   msg='Kid'
else:
  if age<20:
      msg='Teenager'
  else:
      msg='Adult'
```

Example 2:

```
age=21
msg='Kid' if age<13 else 'Teenager' if age<20  else 'Adult'
print(msg)
```

Output:

Adult

PYTHON

LECTURE 17

Today's Agenda

- **Iterative Statements**
  - Types of loop supported by Python
  - The while loop
  - The while-else loop
  - The break , continue and pass Statement

Iterative Statements

- There may be a situation when we need to **execute** a block of code **several number of times**.
- For such situations , **Python** provides the concept of **loop**
- A **loop** statement allows us to **execute a statement** or **group of statements multiple times**

Iterative Statements

- The **2 popular loops** provided by **Python** are:
  - **The while Loop**
  - **The for Loop**
- Recall that **Python** doesn't provide any **do..while** loop unlike **C**,**C++** and **Java**.

The **while** Loop

- **Syntax:**

```
while condition:
        <indented statement 1>
        <indented statement 2>
        …
        <indented statement n>
<non-indented statement 1>
<non-indented statement 2>
```

- **Some Important Points:**
  - First the condition is evaluated. If the condition is **true** then statements in the **while** block is **executed.**
  - After executing statements in the **while** block the condition is checked again and if it is still **true,** then the statements inside the while block is **executed again**.
  - The statements inside the **while** block will keep executing until the condition is **true**.
  - Each execution of the loop body is known as **iteration**.
  - When the condition becomes **false** loop terminates and program control comes **out of the while loop** to begin the **execution** of statement following it.

Examples

- **Example 1:**

```
i=1
while i<=10:
        print(i)
        i=i+1
print("done!")
```

- **Output:**

```
1
2
3
4
5
6
7
8
9
10
done!
```

- **Example 2:**

```
i=1
total=0
while i<=10:
        print(i)
        total+=i
        i=i+1
print("sum is
        {0}".format(total))
```

- **Output:**

```
1
2
3
4
5
6
7
8
9
10
sum is 55
```

Guess The Output

```
i=1
while i<=10:
        print(i)
i=i+1
print("done!")
```

This is an infinite loop

- **Output:**

```
1
1
1
1
1
1
1
1
1
```

```
i=1
while i<=10:
        print(i)
        total+=i
        i=i+1
print("sum is
        {0}".format(total))
```

- **Output:**

```
1
Traceback (most recent call last):
  File "loopdemo2.py", line 5, in <module>
    total=total+i
NameError: name 'total' is not defined
```

Another Form Of "while" Loop

- In **Python** , just like we have an **else** with **if** , similarly we also can have an **else** part with the **while** loop.

- The **statements** in the **else** part are **executed**, when the **condition is not fulfilled** anymore.

Another Form Of "while" Loop

Syntax:

```
while condition:
        <indented statement 1>
        <indented statement 2>
        ...
        <indented statement n>
else:
        <indented statement 1>
        <indented statement 2>
```

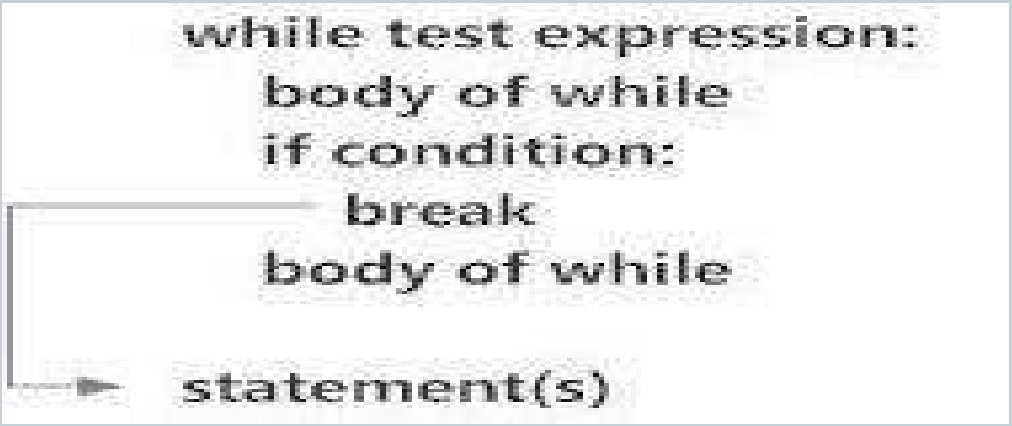
Some Important Points:

- Many programmer's have a doubt that If the statements of the additional **else** part were placed **right after the while loop without an else**, they would **have been executed anyway**, wouldn't they.
- **Then what is the use of else**
- To understand this , we need to understand the **break** statement,

The "break" Statement

- Normally a **while** loop ends only when the **test condition** in the loop becomes **false**.

- However , with the help of a **break** statement a **while** loop can be left prematurely,



Now comes the crucial point:

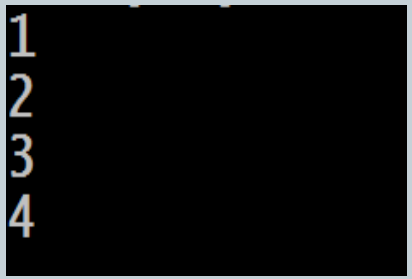
If a loop is left by break,
the else part is not executed.

Example

- **Example 1:**

```
i=1
while i<=10:
        if(i==5):
                break
        print(i)
        i=i+1
else:
        print("bye")
```

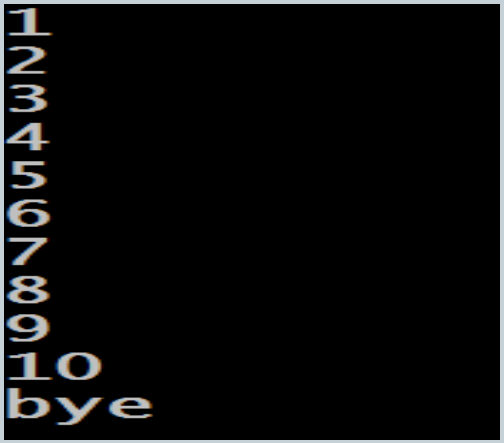
Output:



- **Example 2:**

```
i=1
while i<=10:
        print(i)
        i=i+1
else:
        print("bye")
```

Output:



Exercise

- WAP to accept a string from the user and check whether it contains any vowel or not.

Sample Output:

```
Type a string:sachin
sachin contains vowel
```

```
Type a string:rhythm
rhythm doesn't contain any vowel
```

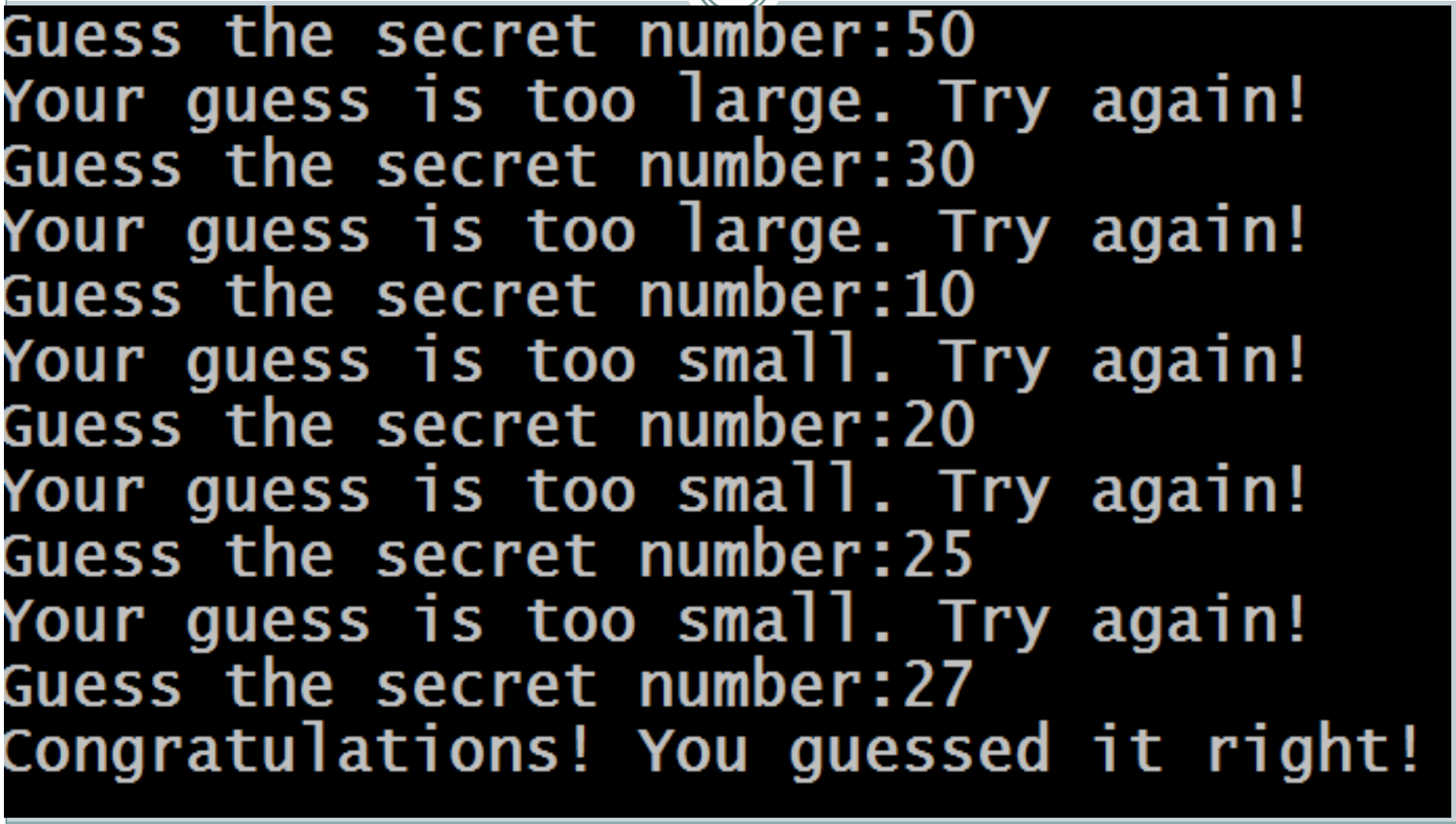
Exercise

- You have to develop a **number guessing game**. The program will generate a **random integer secretly**. Now it will ask the user to guess that number . If the user guessed it correctly then the program prints "**Congratulations! You guessed it right**" .

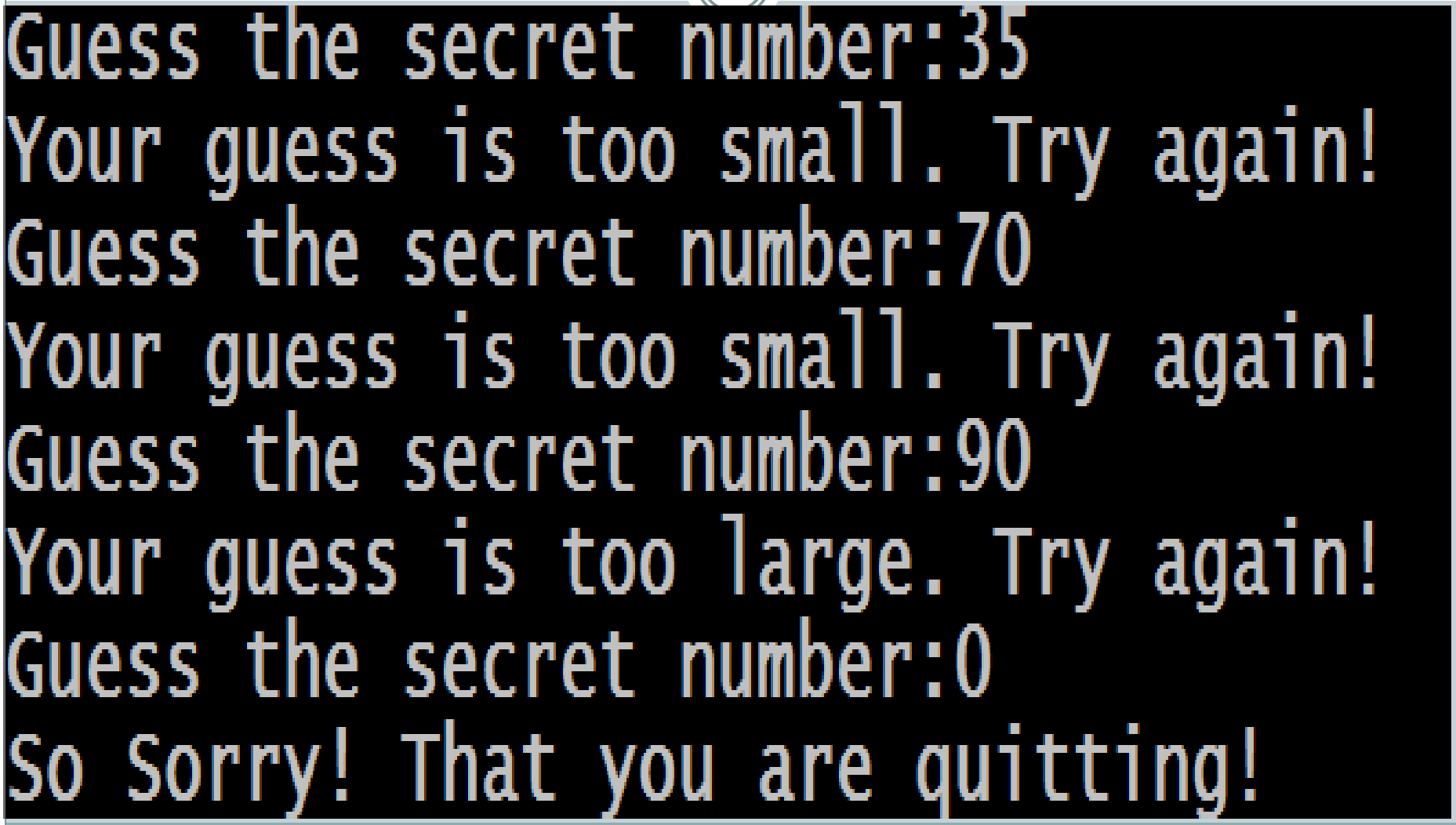
- If the **number guessed by the user** is **larger** than the **secret number** then program should print "**Number too large**" and , if the **number guessed by the user** is **smaller** than the **secret number** then program should print "**Number too small**" .

- This should continue until the **user guesses the number correctly** or **quits** . If the **user wants to quit** in between he will have to type **0** or **negative number**

Output


Guess the secret number:50
Your guess is too large. Try again!
Guess the secret number:30
Your guess is too large. Try again!
Guess the secret number:10
Your guess is too small. Try again!
Guess the secret number:20
Your guess is too small. Try again!
Guess the secret number:25
Your guess is too small. Try again!
Guess the secret number:27
Congratulations! You guessed it right!

Output


Guess the secret number:35
Your guess is too small. Try again!
Guess the secret number:70
Your guess is too small. Try again!
Guess the secret number:90
Your guess is too large. Try again!
Guess the secret number:0
So Sorry! That you are quitting!

How To Generate Random Number

- In **Python** , we have a module named **random** .

- This module contains a function called **randint()** , which accepts **2 arguments** and **returns** a **random number** between them **(both included).**

```
import random
a=random.randint(1,20)
print("Random number is",a)
```

Output:

```
Random number is 16
```

The "continue" Statement

- The **continue** statement in **Python** **returns the control** to the **beginning of the while loop**.

- It **rejects all the remaining statements** in the **current iteration** of the **loop** and **moves the control back** to the **top of the loop**.

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

Example

```
i=0
while i<10:
    i=i+1
    if(i%2!=0):
            continue
    print(i)
```

Output:

```
2
4
6
8
10
```

Exercise

- **Write a program** to **accept** a **string** from the **user** and **display it vertically** but **don't display the vowels** in it.

- **Sample Output:**

```
Type a string:Sachin
S
c
h
n
```

Exercise

- **Write a program** to **continuously accept integers** from the **user** until the user types **0** and **as soon as 0 is entered** display **sum of all the nos** entered before **0**

- **Sample Output:**

```
Enter an integer(press 0 to stop):5
Enter an integer(press 0 to stop):2
Enter an integer(press 0 to stop):11
Enter an integer(press 0 to stop):6
Enter an integer(press 0 to stop):0
Sum is 24
```

Exercise

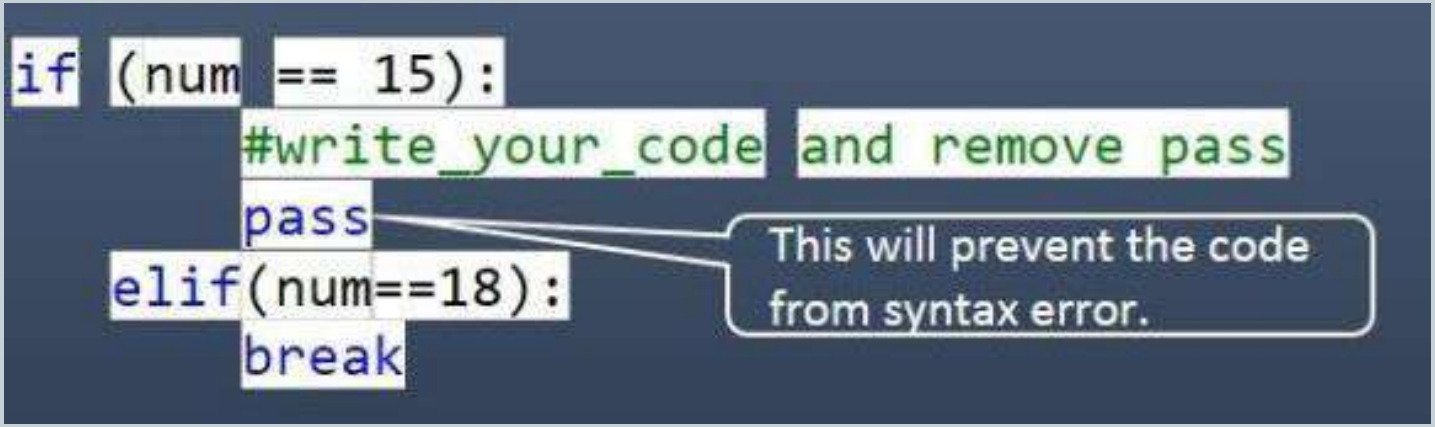
- **Modify** the **previous code** so that if the **user** inputs **negative integer** , your program should **ignore it** .

- **Sample Output:**

```
Enter an integer(press 0 to stop):5
Enter an integer(press 0 to stop):2
Enter an integer(press 0 to stop):-6
Enter an integer(press 0 to stop):11
Enter an integer(press 0 to stop):0
Sum is 18
```

The "pass" Statement

- In **Python**, the **pass** statement is a **no operation statement**.
- That is , **nothing happens** when **pass** statement is **executed.**
- **Example:**



Example

```
i=0
while i<10:
    i=i+1
    if(i%2!=0):
            pass
    else:
            print(i)
```

Output:

```
2
4
6
8
10
```

PYTHON

LECTURE 18

Today's Agenda

- The for Loop
  - The for Loop In Python
  - Differences with other languages
  - The range( ) Function
  - Using for with range( )

The for Loop

- Like the **while** loop the **for** loop also is a **programming language statement**, i.e. an **iteration statement**, which **allows a code block** to be **executed multiple number of times**.

- There are **hardly programming languages** without **for** loops.

- However the **for** loop **exists in many different flavours**, i.e. both the **syntax** and the **behaviour differs from language to language**

The for Loop

- **Different Flavors Of "for" Loop:**
- **Count-controlled for loop (Three-expression for loop):**
  - This is **by far** the **most common** type. This statement is the one used by **C** , **C++** and **Java** . Generally it has the form:
    for (i=1; i <= 10; i++)
    This kind of for loop is not implemented in Python!
- **Numeric Ranges**
  - This kind of **for loop** is a **simplification of the previous** kind. Starting with a **start value** and **counting up** to an **end value**, like
  - **for i = 1 to 100**
    Python doesn't use this either.

The for Loop

- **Iterator-based for loop**
  - Finally, we come to the one used by **Python**. This kind of a **for** loop **iterates** over a **collection** of **items**.
  - In **each iteration** step a **loop variable** is set to a **value** in a **sequence** or other **data collection.**
  - This kind of **for loop** is known in most **Unix** and **Linux** shells and it is the one which is implemented in **Python**.

Syntax Of for Loop In Python

- **Syntax:**

```
for some_var in some_collection:
        # loop body
        <indented statement 1>
        <indented statement 2>
         ...
        <indented statement n>
<non-indented statement 1>
<non-indented statement 2>
```

- **Some Important Points:**
  - The for loop in Python can iterate over string , list, tuple , set,frozenset, bytes,bytearray and dictionary
  - The **first item** in the **collection** is assigned to the loop variable.
  - Then the block is **executed.**
  - Then again the **next item** of **collection** is assigned to the loop variable, and the statement(s) block is **executed**
  - This goes on until the entire **collection** is exhausted.

Examples

- **Example 1:**

```
word="Sachin"
for ch in word:
    print(ch)
```

Output:

```
S
a
c
h
i
n
```

- **Example 2:**

```
fruits=["Apple","Banana","Guava","Orange"]
for fruit in fruits:
    print(fruit)
```

- **Output:**

```
Apple
Banana
Guava
Orange
```

Exercise

- **Write a program** using **for** loop to **accept** a **string** from the **user** and **display it vertically** but **don't display the vowels** in it.

- **Sample Output:**

```
Type a string:Sachin
S
c
h
n
```

QUIZ- Test Your Skills

1. **What is the output ?**

```
word="sachin"
if(ch in ["a","e","i","o","u"]):
    continue
print(ch,end=" ")
```

A. s c h n
B. Error
C. s a c h i n
D. Exception

Correct Answer: B

QUIZ- Test Your Skills

2. **What is the output?**

```
i=0
while i<4:
        i=i+1
        if(i%2!=0):
          pass
          print("hi",end=" ")
        else:
          print(i,end=" ")
```

A. hi 2 hi 4
B. Syntax Error
C. 2 4
D. Infinite loop

Correct Answer: A

QUIZ- Test Your Skills

3. **What is the output?**

```
i=0
while i<4:
        i=i+1
        if(i%2!=0):
          continue
          print("hi",end=" ")
        else:
          print(i,end=" ")
```

A. hi 2 hi 4
B. Syntax Error
C. 2 4
D. Infinite loop

Correct Answer: C

QUIZ- Test Your Skills

4. **What is the output?**

```
i=0
while i<4:
        i=i+1
        if(i%2!=0):
          break
          print("hi",end=" ")
        else:
          print(i,end=" ")
```

A. hi 2 hi 4
B. Syntax Error
C. 2 4
D. No output

Correct Answer: D

QUIZ- Test Your Skills

5. What is the output ?

```
x = 123
for i in x:
   print(i)
```

A. 123
B. 1
   2
   3
C. TypeError
D. Infinite loop

Correct Answer: C

QUIZ- Test Your Skills

6. **What is the output ?**

```
i = 1
while True:
        if i%3 == 0:
                break
        print(i,end=" ")
        i + = 1
```

A. Syntax Error
B. 1 2
C. 1 2 3
D. Blank Screen(No Output)

Correct Answer: A

QUIZ- Test Your Skills

7. **What is the output ?**

```
i = 1
while True:
        if i%2 == 0:
                break
        print(i,end=" ")
        i += 2
```

A. 1
B. 1 2
C. Infinite loop
D. Syntax Error

Correct Answer: C

QUIZ- Test Your Skills

8. **What is the output ?**

```
x = "abcdef"
i = "i"
while i in x:
        print(i, end=" ")
```

A. a b c d e f
B. i i i i i i
C. Error
D. No output

Correct Answer: D

QUIZ- Test Your Skills

9. What is the output ?

```
x = "abcdef"
i = "a"
while i in x:
        print(i, end=" ")
```

A. a b c d e f
B. Infinite loop
C. Error
D. No output

Correct Answer: B

QUIZ- Test Your Skills

10. What is the output ?

```
x = "abcdef"
i = "a"
while i in x:
   x = x[1:]
   print(i, end = " ")
```

A. a a a a a a
B. a
C. Error
D. No output

Correct Answer: B

QUIZ- Test Your Skills

11. What is the output ?

```
x = 'abcd'
for i in x:
    print(i,end=" ")
    x.upper()
```

A. a B C D
B. A B C D
C. a b c d
D. Syntax Error

Correct Answer: C

QUIZ- Test Your Skills

12. What is the output ?

```
x = 'abcd'
for i in x:
    print(i.upper())
```

A. a B C D
B. A B C D
C. a b c d
D. Syntax Error

Correct Answer: B

QUIZ- Test Your Skills

13. What is the output ?

```
text = "my name is sachin"
for i in text:
   print (i, end=", ")
```

A. my,name,is,sachin,
B. m,y, ,n,a,m,e, ,i,s, ,s,a,c,h,i,n,
C. Syntax Error
D. No output

Correct Answer: B

QUIZ- Test Your Skills

14. What is the output ?

```
text = "my name is sachin"
for i in text.split():
   print (i, end=", ")
```

A. my,name,is,sachin,
B. m,y,n,a,m,e,i,s,s,a,c,h,i,n
C. Syntax Error
D. No output

Correct Answer: A

QUIZ- Test Your Skills

15. What is the output ?

```
text = "my name is sachin"
for i not in text:
   print (i, end=", ")
```

A. my,name,is,sachin,
B. m,y,n,a,m,e,i,s,s,a,c,h,i,n
C. Syntax Error
D. No output

Correct Answer: C

QUIZ- Test Your Skills

16. What is the output?

```
True = False
while True:
        print(True)
        break
```

A. True
B. False
C. No output(Blank Screen)
D. None of the above

Correct Answer: D

QUIZ- Test Your Skills

17. What is the output?

```
i = 2
while True:
        if i%3 == 0:
          break
        print(i,end=" ")
        i += 2
```

A. Infinite loop
B. 2 4
C. 2 3
D. None of the above

Correct Answer: B

The range Function

- The range() function is a built-in function in Python, and it returns a range object.
- This function is very useful to generate a sequence of numbers in the form of a List.
- The range( ) function takes 1 to 3 arguments

The range Function With One Parameter

- **Syntax:**
  - range(n)

- For an argument **n**, the function returns a **range** object containing integer values from **0** to **n-1**.

Example:

```
a=range(10)
print(a)
```

Output:

```
range(0, 10)
```

As we can see that when we display the variable **a** , we get to see the description of the **range** object and not the values.

To see the values , we must convert **range** object to **list**

The range Function With One Parameter

Example:

```
a=range(10)
b=list(a)
print(b)
```

The function list( ) accepts a range object and converts it into a list of values .
These values are the numbers from 0 to n-1 where n is the argument passed to the function range()

Output:

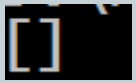
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

What If We Pass Negative Number ?

Guess:

```
a=range(-10)
b=list(a)
print(b)
```

Output:



The output is an **empty list** denoted by **[ ]** and it tells us that the function **range( )** is coded in such a way that it always moves towards **right side** of the **start value** which here is **0**.

But since **-10** doesn't come **towards right** of **0**, so the output is an **empty list**

The range Function With Two Parameter

- **Syntax:**
  - range(m,n)

- For an argument **m,n** , the function returns a **range** object containing **integer values** from **m** to **n-1**.

Example:

```
a=range(1,10)
print(a)
```

Output:

```
range(1, 10)
```

Here again when we display the variable **a** , we get to see the description of the **range** object and not the values. So we must use the function **list( )** to get the values

The range Function With Two Parameter

Example:

```
a=range(1,10)
b=list(a)
print(b)
```

Output:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The output is **list** of numbers from **1** to **9** because **10** falls **towards right** of **1**

What If We Pass First Number Greater?

Guess:

```
a=range(10,1)
b=list(a)
print(b)
```

Output:

```
[]
```

The output is an **empty list** because **as mentioned earlier** it **traverses towards right** of start value and **1** doesn't come to the right of **10**

Passing Negative Values

- We can pass **negative start** or/and **negative stop value** to **range( )** when we call it with **2 arguments** .

Example:

```
a=range(-10,3)
b=list(a)
print(b)
```

Since **3** falls on **right of -10** ,
so we are **getting range of numbers** from **-10** to **3**

Output:

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2]
```

Guess The Output

```
a=range(-10,-3)
b=list(a)
print(b)
```

- **Output:**

```
[-10, -9, -8, -7, -6, -5, -4]
```

```
a=range(-3,-3)
b=list(a)
print(b)
```

Output:

```
[]
```

```
a=range(-3,-10)
b=list(a)
print(b)
```

- **Output:**

```
[]
```

The range Function With Three Parameter

- **Syntax:**
  - range(m,n,s)
- Finally, the **range()** function can also take the **third parameter** . This is for the **step value**.

Example:

```
a=range(1,10,2)
b=list(a)
print(b)
```

Since **step value** is **2** , so we got nos from **1** to **9** with a **difference** of **2**

Output:

```
[1, 3, 5, 7, 9]
```

Guess The Output

```
a=range(7,1,-2)
b=list(a)
print(b)
```

- **Output:**

```
[7, 5, 3]
```

Pay close attention , that we are having start value greater than end value , but since step value is negative , so it is allowed

```
a=range(5,10,20)
b=list(a)
print(b)
```

Output:

```
[5]
```

Here, note that the first integer, 5, is always returned, even though the interval 20 sends it beyond 10

Guess The Output

```
a=range(2,14,1.5)
b=list(a)
print(b)
```

Note that all three arguments must be integers only.

- **Output:**

```
TypeError: 'float' object cannot be interpreted as an integer
```

```
a=range(5,10,0)
b=list(a)
print(b)
```

Output:

```
ValueError: range() arg 3 must not be zero
```

It raised a ValueError because the interval cannot be zero if we need to go from one number to another.

Guess The Output

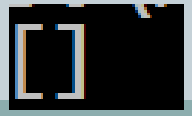
```
a=range(2,12)
b=list(a)
print(b)
```

- **Output:**

The default value of step is 1 , so the output is from 2 to 11

```
a=range(12,2)
b=list(a)
print(b)
```

Output:



As usual , since the start value is greater than end value so we get an empty list

Using range( ) With for Loop

- We can use **range()** and **for** together for **iterating** through a **list** of **numeric values**

- **Syntax:**
  - for <var_name> in range(end):
    indented statement 1
    indented statement 2
    .
    .
    indented statement n

Example

Code:

```
for i in range(11):
    print(i)
```

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Using 2 Parameter range( ) With for Loop

- We can use 2 argument **range()** with **for** also for iterating through a list of **numeric values** between a **given range**

- **Syntax:**
  - for <var_name> in range(start,end)

```
    indented statement 1
    indented statement 2
    .
    .
    indented statement n
```

Example

Code:

```
for i in range(1,11):
    print(i)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Exercise

- **Write a program** to **accept** an **integer** from the **user** and display the **sum of all the numbers** from **1** to that **number**.

- **Sample Output:**

```
Enter an int:5
sum of nos from 1 to 5 is 15
```

Solution

```
num=int(input("Enter an int:"))
total=0
for i in range(1,num+1):
    total=total+i
print("sum of nos from 1 to {} is {}".format(num,total))
```

Exercise

- **Write a program** to **accept** an **integer** from the **user** and **calculate it's factorial**

- **Sample Output:**

```
Enter an int:6
Factorial of 6 is 720
```

Using 3 Parameter range( ) With for Loop

- **Syntax:**
  - for <var_name> in range(start,end,step)

    indented statement 1

    indented statement 2

    .

    .

    indented statement n

Example

Code:

```
for i in range(1,11,2):
    print(i)
```

Output:

```
1
3
5
7
9
```

Example

Code:

```
for i in range(100,0,-10):
  print(i)
```

Output:

```
100
90
80
70
60
50
40
30
20
10
```

Using for With else

- Just like **while** , the **for** loop can also have an **else** part , which **executes** if no **break** statements executes in the **for** loop

- **Syntax:**

```
for <var_name> in some_seq:
                indented statement 1
                if test_cond:
                   break
 else:
                indented statement 3
                indented statement 4
```

Example

Code:

```
for i in range(10):
  print(i)
else:
  print("Loop complete")
```

Output:

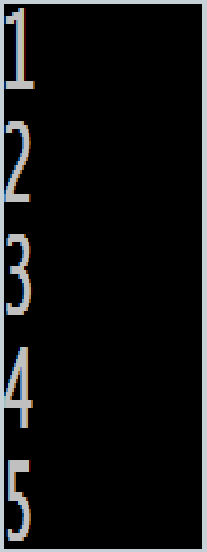
```
0
1
2
3
4
5
6
7
8
9
Loop complete
```

Example

Code:

```
for i in range(1,10):
  print(i)
  if i%5==0:
        break
else:
  print("Loop complete")
```

Output:



Using Nested Loop

- **Loops** can be **nested** in **Python**, as they can with **other programming languages.**

- A **nested loop** is a **loop** that **occurs within another loop**, and are constructed like so:

- **Syntax:**

```
for <var_name> in some_seq:
        for <var_name> in some_seq:
                indented statement 1
                indented statement 2
```

Example

Code:

```
numbers = [1, 2, 3]
alpha = ['a', 'b', 'c']
for n in numbers:
    print(n)
    for ch in alpha:
            print(ch)
```

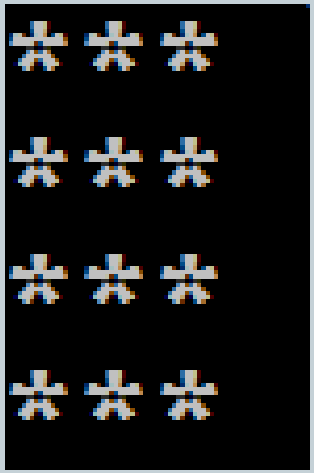
Output:

```
1
a
b
c
2
a
b
c
3
a
b
c
```

Exercise

- Write a program to print the following pattern

Sample Output:



Solution

Code:

```
for i in range(1,5):
    for j in range(1,4):
            print("*",end="")
    print()
```

Output:

Solution

Can you write the same code using only single loop ?

Code:

```
for i in range(1,5):
    print("*"*3)
```

Output:

Exercise

- Write a program to print the following pattern

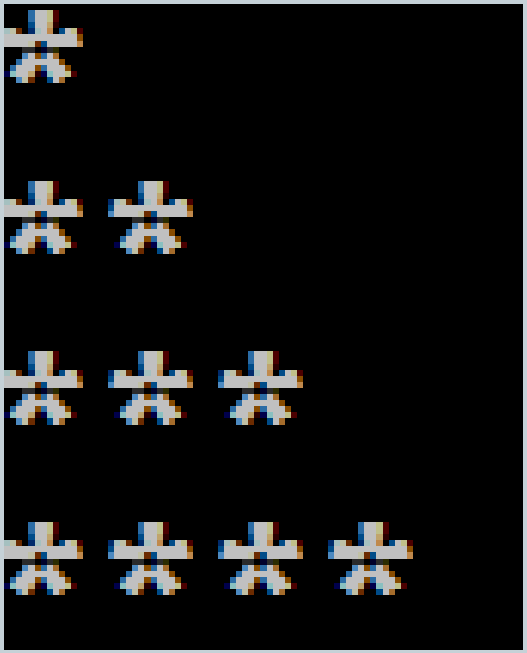
Sample Output:

Solution

Code:

```
for i in range(1,5):
    for j in range(1,i+1):
            print("*",end="")
    print()
```

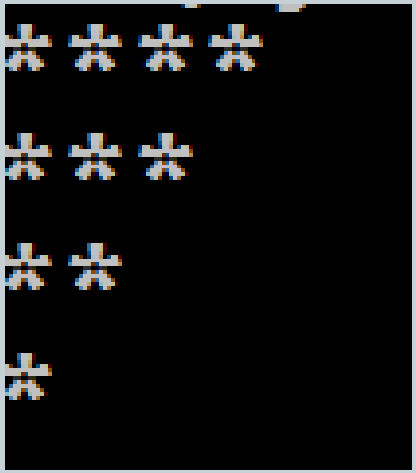
Output:



Exercise

- Write a program to print the following pattern

Sample Output:



Solution

Code:

```
for i in range(4,0,-1):
    for j in range(1,i+1):
            print("*",end="")
    print()
```

Output:

Exercise

- **Write a program** to **accept** an **integer** from the **user** and display **all the numbers** from **1** to **that number**. **Repeat** the **process** until the user enters **0**.
- **Sample Output:**

```
Enter a number: 3
1
2
3
Enter a number: 9
1
2
3
4
5
6
7
8
9
Enter a number: 0
```

Solution

Code:

```
x = int(input('Enter a number: '))
while x != 0:
    for y in range (1, x+1):
        print (y)
    x = int(input('Enter a number: '))
```

Output:

```
Enter a number: 3
1
2
3
Enter a number: 9
1
2
3
4
5
6
7
8
9
Enter a number: 0
```

PYTHON

LECTURE 19

Today's Agenda

- **User Defined Functions**
  - What Is A Function ?
  - Function V/s Method
  - Steps Required For Developing User Defined Function
  - Calling A Function
  - Returning Values From Function

What Is A Function ?

- A **function** in **Python** is a **collection of statements** having a **particular name** followed by **parenthesis** .
- To **run** a function , we have to **call** it and when we call a function **all the statements** inside the **function** are **executed**.
- So we don't have to write the code again and again
- This is called **code re-usability**.

Function V/s Method

- **Functions** are block of codes defined **individually** .
- But if a function is **defined inside a class** , it becomes a **method**
- So , **methods** and **functions** are same except their placement in the program.
- Also we can call a **function** directly using it's **name** but when we call a **method** we have to use either **object name** or **class name** before it

Function V/s Method

For example:

- **print(“hello”)**

- Here **print( )** is a function as we are calling it directly

  - **message=“Good Morning”**
  - **print(message.lower())**

- Here **lower( )** is a method which belongs to the class **str** and so it is called using the object **message**

Steps Required For Function

- To create and use a function we have to take **2 steps**:

- **Function Definition**: Creating or writing the body of a function is called defining it. It contains the set of statements we want to run , when the function execute.

- **Function Call**: A function never runs automatically . So to execute it's statements we must call it

Syntax Of Function Definition

```
def function_name(param 1,param 2, . . .):
        statement(s)
```

- Keyword **def** marks the start of **function header**.
- It is followed by a **function name** to uniquely identify it.
- **Parameters** (arguments) through which we pass values to a function. They are **optional**.
- A **colon** (:) to mark the end of **function header**.
- One or more valid **python statements** that make up the function body . All the statements must have same indentation level

Example Of Function Definition

```
def add(a,b):
    print("Values are",a,"and",b)
    c=a+b
    print("Their sum is",c)
```

How To Call A Function ?

- Once we have **defined** a function, we can **call** it from another **function, program** or even the **Python prompt**.

- To call a function we simply type the function name with appropriate parameters.

- **Syntax:**

  - function_name(arguments)

Complete Example

```
def add(a,b):
    print("Values are",a,"and",b)
    c=a+b
    print("Their sum is",c)
add(5,10)
add(2.5,5.4)
```

Output:

```
Values are 5 and 10
Their sum is 15
Values are 2.5 and 5.4
Their sum is 7.9
```

Returning Values From Function

- To return a value or values from a function we have to write the keyword **return** at the end of the function body along with the value(s) to be returned

- **Syntax:**

  - return <expression>

Complete Example

```
def add(a,b):
    c=a+b
    return c
x=add(5,10)
print("Sum of 5 and 10 is",x)
y=add(2.5,5.4)
print("Sum of 2.5 and 5.4 is",y)
```

Output:

```
Sum of 5 and 10 is 15
Sum of 2.5 and 5.4 is 7.9
```

Exercise

- **Write a function called absolute( ) to accept an integer as argument and return it's absolute value. Finally call it to get the absolute value of -7 and 9**

- **Sample Output:**

```
absolute of -7 is 7
absolute of 9 is 9
```

Solution

```
def absolute(n):
    if n>0:
        return n
    else:
        return -n

x=absolute(-7)
print("absolute of -7 is",x)
y=absolute(9)
print("absolute of 9 is",y)
```

Exercise

- **Write a function called factorial( ) which accepts a number as argument and returns it's factorial. Finally call the function to calculate and return the factorial of the number given by the user.**

```
Enter an int:4
Factorial of 4 is 24
```

Solution

```
def factorial(n):
    f=1
    while n>1:
        f=f*n
        n=n-1
    return f

x=int(input("Enter an int:"))
y=factorial(x)
print("Factorial of",x,"is",y)
```

Guess The Output

```
def greet(name):
    print("Hello",name)

greet("sachin")
greet()
```

Output:

```
Hello sachin
Traceback (most recent call last):
  File "func5.py", line 5, in <module>
    greet()
TypeError: greet() missing 1 required positional argument: 'name'
```

Guess The Output

```
def greet(name):
    print("Hello",name)

greet("sachin", "amit")
```

Output:

```
TypeError: greet() takes 1 positional argument but 2 were given
```

Guess The Output

```
def greet(name):
    print("Hello",name)
    return
    print("bye")

greet("sachin")
```

Output:

```
Hello sachin
```

Guess The Output

```
def greet(name):
    print("Hello",name)


x=greet("sachin")
print("value in x is",x)
```

Output:

```
Hello sachin
value in x is None
```

Returning Multiple Values From Function

- In languages like **C** or **Java** , a function can return only one value . However in **Python** , a function can return **multiple values** using the following syntax:

- **Syntax:**
  - return <value 1, value 2, value 3, . . . >

- **For example:**
  - return a,b,c

- When we do this , **Python** returns these values as a **tuple**, which just like a **list** is a collection of multiple values.

Receiving Multiple Values

- To receive multiple values returned from a function , we have **2 options:**
- **Syntax 1:**
  - var 1,var 2,var 3=<function_name>()
- **Syntax 2:**
  - var=<function_name>()
- In the **first case** we are receiving the values in **individual variables** . Their **data types** will be set according to the **types of values** being **returned**
- In the **second case** we are receiving it in a **single variable** and **Python** will automatically make the data type of this variable as **tuple**

Complete Example

```
def calculate(a,b):
    c=a+b
    d=a-b
    return c,d
x,y=calculate(5,3)
print("Sum is",x,"and difference is",y)
z=calculate(15,23)
print("Sum is",z[0],"and difference is",z[1])
```

Here Python will automatically set x and y to be of int type and z to be of tuple type

Output:

```
Sum is 8 and difference is 2
Sum is 38 and difference is -8
```

PYTHON

LECTURE 20

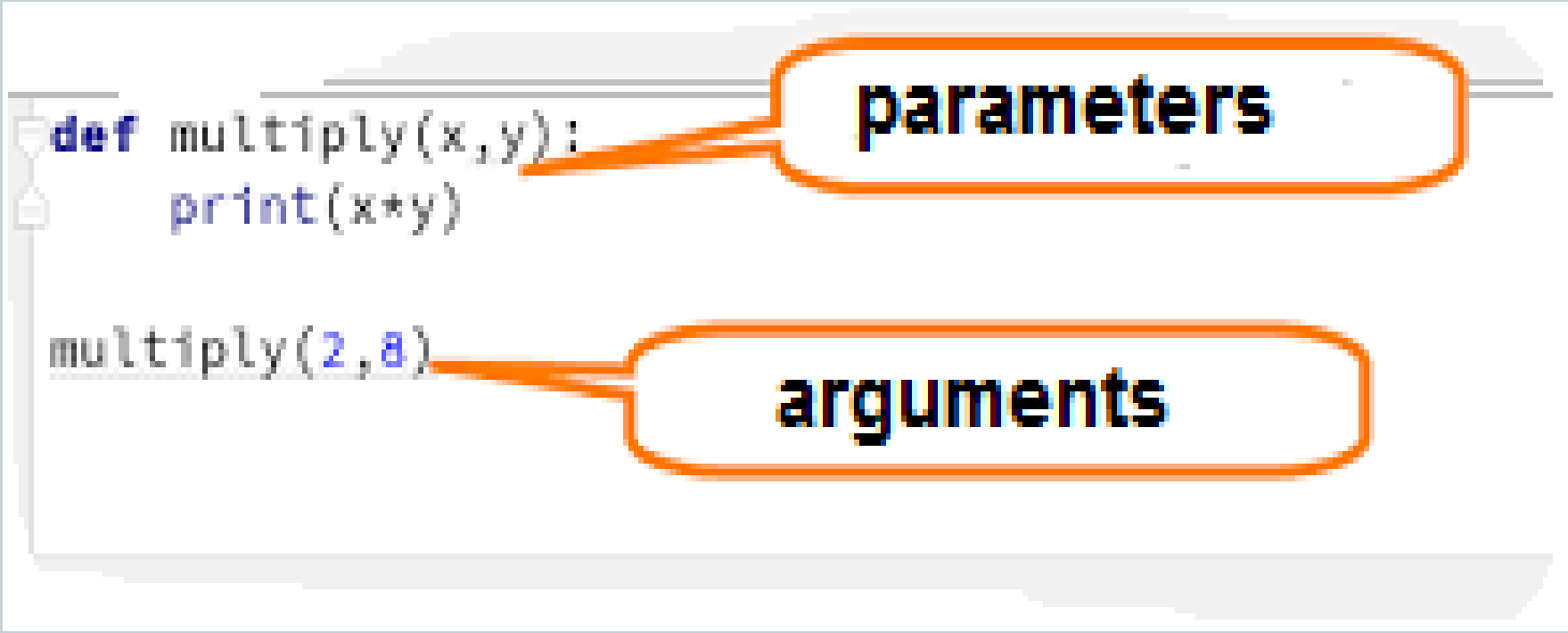
Today's Agenda

- **User Defined Functions-II**
  - Arguments V/s Parameters
  - Types Of Arguments

Parameters V/s Arguments?

- A lot of people—mix up **parameters** and **arguments**, although they are slightly different.
- A **parameter** is a variable in a **method definition**.
- When a method is called, the **arguments** are the data we pass into the method's **parameters**.

Parameters V/s Arguments?



Types Of Arguments

- In **Python** , a function can have **4** types of arguments:
  - **Positional Argument**
  - **Keyword Argument**
  - **Default Argument**
  - **Variable Length Argument**

Positional Arguments

- These are the arguments passed to a function in correct **positional order**.

- Here the number of **arguments** in the call must exactly match with number of **parameters** in the function definition

Positional Arguments

```
def attach(s1,s2):
    s3=s1+s2
    print("Joined String is:",s3)

attach("Good","Evening")
```

These are called

POSITIONAL ARGUMENTS

Output:

```
Joined String is: GoodEvening
```

Positional Arguments

- If the **number of arguments** in call do not match with the **number of parameters** in function then we get **TypeError**:

```
def attach(s1,s2):
    s3=s1+s2
    print("Joined String is:",s3)

attach("Good")
```

Output:

```
TypeError: attach() missing 1 required positional argument: 's2'
```

Guess The Output

```
def grocery(name,price):
    print("Item is",name,"It's price is",price)

grocery("Bread",20)
grocery(150,"Butter")
```

Output:

```
Item is Bread It's price is 20
Item is 150 It's price is Butter
```

The Problem With Positional Arguments

- The problem with **positional arguments** is that they always **bind** to the **position** of parameters.
- So **1st argument** will be copied to **1st parameter** , **2nd argument** will be copied to **2nd parameter** and so on.
- Due to this in the previous example the value **150** was copied to **name** and "**Butter**" was copied to **price**
- To solve this problem , **Python** provides us the concept of **keyword arguments**

Keyword Arguments

- **Keyword arguments** are arguments that identify parameters with their names

- With **keyword arguments** in **Python**, we can change the order of passing the arguments without any consequences

- **Syntax:**

function_name(paramname1=value,paramname2=value)

Complete Example

```
def grocery(name,price):
    print("Item is",name,"It's price is",price)

grocery(name="Bread",price=20)
grocery(price=150,name="Butter")
```

Output:

```
Item is Bread It's price is 20
Item is Butter It's price is 150
```

Point To Remember!

- A **positional argument** can never follow a **keyword argument** i.e. the **keyword argument** should always appear after **positional argument**

- **For example:**
  - **def display(num1,num2):**
    - **# some code**

Now if we call the above function as:

display(10,num2=15)

Then it will be **correct**. But if we call it as:

display(num1=10,15)

Then it will be a **Syntax Error**

Default Arguments

- For some functions, we may want to make some parameters ***optional*** and use **default values** in case the user does not want to provide values for them.

- This is done with the help of **default argument** values.

- We can specify **default argument** values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the **default value**.

- **Syntax:**

```
def function_name(paramname1=value,paramname2=value):
    # function body
```

Complete Example

```
def greet(name,msg="Good Morning"):
    print("Hello",name,msg)

greet("Sachin")
greet("Amit","How are you?")
```

Output:

```
Hello Sachin Good Morning
Hello Amit How are you?
```

Point To Remember!

- A function can have **any number of default arguments** but once we have a **default argument**, all the arguments to **it's right must also have default values**.

- This means to say, **non-default arguments** cannot follow **default arguments**.

Point To Remember!

- **For example:** if we had defined the function header above as:

def greet(msg = "Good morning!", name):

- **Then we would have got the following SyntaxError**

```
    def greet(msg="Good Morning",name):
              ^
SyntaxError: non-default argument follows default argument
```

Point To Remember!

- If a function has **default arguments** , set then while calling it if we are **skipping** an argument then **we must skip all the arguments after it also**.
- **For example:**

```
def show(a=10,b=20,c=30):
    print(a,b,c)
```

- Now , if we call the above function as :

```
show(5)
```

- It will work and output will be **5 20 30**

- If we call it as :

```
show(5,7)
```

- Still it will work and output will be **5 7 30**
- But if we call it as

```
show(5, ,7)
```

- Then it will be an error

The solution to this problem is to use **default argument** as **keyword argument** :

```
show(5,c=7)
```

This will give the output as **5 20 7**

Exercise

- Write a function called **cal_area( )** using **default argument** concept which accepts **radius** and **pi** as arguments and calculates and displays area of the Circle. The value of **pi** should be used as **default argument** and value of **radius** should be **accepted from the user**

Solution

```
def cal_area(radius,pi=3.14):
    area=pi*radius**2
    print("Area of circle with radius",radius,"is",area)

rad=int(input("Enter radius:"))
cal_area(rad)
```

```
Enter radius:4
Area of circle with radius 4 is 50.24
```

Guess The Output ?

```
def addnos(a,b):
    c=a+b
    return c

def addnos(a,b,c):
    d=a+b+c
    return d
print(addnos(10,20))
print(addnos(10,20,30))
```

Output:

```
    print(addnos(10,20))
TypeError: addnos() missing 1 required positional argument: 'c'
```

Why Did The Error Occur ?

- The error occurred because **Python** does not support **Function** or **Method Overloading**
- **Moreover Python understands the latest definition of a function addnos( ) which takes 3 arguments**
- Now since we passed **2 arguments** only , the call generated **error** because Python tried to call the method with **3 arguments**

Solution

- The solution to this problem is a technique called **variable length arguments**

- In this technique , we define the function in such a way that it can accept any number of arguments from **0** to **infinite**

Syntax Of Variable Length Arguments

def <function_name>(* <arg_name>):

Function body

- As we can observe , to create a function with **variable length arguments** we simply prefix the argument name with an **asterisk**.
- **For example:**

def addnos(*a):

- The function **addnos()** can now be called with as many **number of arguments** as we want and all the arguments will be stored inside the argument **a** which will be internally treated as **tuple**

Complete Example

```
def addnos(*a):
    sum =0
    for x in a:
        sum=sum+x
    return sum

print(addnos(10,20))
print(addnos(10,20,30))
```

Output:

```
30
60
```

Exercise

- Write a function called **findlargest( )** which accepts multiple strings as argument and returns the length of the largest string as well as the string itself

Solution

```
def findlargest(*names):
    max=0
    for s in names:
        if len(s)>max:
            max=len(s)
    return max
print(findlargest("January","February","March"))
```

Output:

```
8
```

Exercise

- Modify the previous example so that the function **findlargest( )** now returns the largest string itself and not it's length

Solution

```
def findlargest(*names):
    max=0
    largest=""
    for s in names:
        if len(s)>max:
            max=len(s)
            largest=s
    return largest
print(findlargest("January","February","March"))
```

Output:

```
February
```

Point To Remember!

- A function cannot have 2 variable length arguments. So the following is wrong:

```
def addnos(*a,*b):
```

Point To Remember!

- If we have any other argument along with **variable length argument** , then it should be set **before** the **variable length argument**

```
def addnos(n,*a):
    sum =n
    for x in a:
        sum=sum+x
    return sum
print(addnos(10,20,30))
print(addnos(10,20,30,40))
```

Point To Remember!

- If we set the other argument used with **variable length argument , after** the **variable length argument** then:
  - While calling it we must pass it as **keyword argument** OR
  - Either it should be set as **default argument**

```
def addnos(*a,n):
    sum =n
    for x in a:
        sum=sum+x
    return sum
print(addnos(20,30,n=0))
print(addnos(20,30,40,n=0))
```

```
def addnos(*a,n=0):
        sum =n
        for x in a:
            sum=sum+x
        return sum

print(addnos(20,30))
print(addnos(20,30,40))
```

Guess The Output

```
def addnos(*a,n):
    sum =n
    for x in a:
        sum=sum+x
    return sum
print(addnos(20,n=10,30))
```

Output:

SyntaxError: Positional argument follows keyword argument

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show(10,20)
```

Output:

10 20 3 4

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show(10,20,30,40)
```

Output:

10 20 30 40

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show(d=10,a=20,b=30)
```

Output:

20 30 3 10

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show()
```

Output:

TypeError

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show(c=30,d=40,10,20)
```

Output:

SyntaxError

Guess The Output

```
def show(a,b,c=3,d=4):
    print(a,b,c,d)

show(30,40,b=15)
```

Output:

TypeError : got multiple values for argument ‘b’

PYTHON

LECTURE 21

Today's Agenda

- **User Defined Functions-III**
  - Variable Scope
  - Local Scope
  - Global Scope
  - Argument Passing

Variable Scopes

- The ***scope*** of a variable refers to the places from where we can see or access a variable.

- In Python , there are 4 types of scopes:
  - **Local : Inside a function body**
  - **Enclosing: Inside an outer function's body . We will discuss it later**
  - **Global: At the module level**
  - **Built In: At the interpreter level**

- In short we pronounce it as **LEGB**

Global Variable

- **GLOBAL VARIABLE**
  - A variable which is defined in the **main body** of a **file** is called a ***global*** variable.
  - It will be visible **throughout** the **file**

Local Variable

- **LOCAL VARIABLE**
  - A variable which is defined **inside a function** is **local** to that function.
  - It is accessible **from the point** at which it is defined until the **end of the function**.
  - It exists for as long as the function is executing.
  - Even the **parameter** in the function definition behave like **local variables**
  - **When we use the assignment operator (=) inside a function, it's default behaviour is to create a new local variable – unless a variable with the same name is already defined in the local scope.**

Example

```
s = "I love Python"
def f():
    print(s)
f()
```

Since the variable **s** is **global** , we can access it from anywhere in our code

Output:

I love Python

Example

```
s = "I love Python"
def f():
    print(s)
```

Output:

Since we have not called the function f( ) , so the statement print(s) will never get a chance to run

Example

```
def f():
    print(s)
s = "I love Python"
f()
```

Output:

I love Python

Even though the variable **s** has been declared after the function **f( )** , still it is considered to be **global** and can be accessed from anywhere in our code

Example

```
def f():
   print(s)
f()
s="I love Python"
```

Output:

NameError !

Since we have called the function f( ) , before declaring variable s , so we get NameError!

Example

```
def f():
    s="I love Python"
    print(s)
f()
```

Output:

I love Python

The variable s now becomes a local variable and a function can easily access all the local variables inside it's definition

Example

```
def f():
    s="I love Python"
    print(s)
f()
print(s)
```

Output:

I love Python

NameError!

The variable s is local and cannot be accessed from outside it's function's definition

Example

```
s="I love Python"
def f():
    s="I love C"
    print(s)
f()
print(s)
```

Output:

```
I love C
I love Python
```

If a variable with same name is defined inside the scope of function as well then Python creates a new variable in local scope of the function and uses it

Example

What if we want to use the same global variable inside the function also ?

```
s="I love Python"
def f():
    global s
    s="I love C"
    print(s)
f()
print(s)
```

Output:

```
I love C
I love C
```

To do this , we need a special keyword in Python called global. This keyword tells Python , not to create any new variable , rather use the variable from global scope

Guess The Output ?

```
s="I love Python"
def f():
    print(s)
    s="I love C"
    print(s)
f()
print(s)
```

Now , this is a special case! . In Python any variable which is changed or created inside of a function is local, if it hasn't been declared as a global variable. To tell Python, that we want to use the global variable, we have to explicitly state this by using the keyword "global"

Output:

UnboundLocalError!:

Local variable s referenced before assignment

Guess The Output ?

```
s="I love Python"
def f():
    global s
    print(s)
    s="I love C"
    print(s)
f()
print(s)
```

Output:

```
I love Python
I love C
I love C
```

Guess The Output ?

```
a=1
def f():
    print ('Inside f() : ', a)
def g():
    a = 2
    print ('Inside g() : ',a)
def h():
    global a
    a = 3
    print ('Inside h() : ',a)

print ('global : ',a)
f()
print ('global : ',a)
g()
print ('global : ',a)
h()
print ('global : ',a)
```

Output:

```
global : 1
inside f( ):1
global: 1
inside g( ): 2
global : 1
inside h( ): 3
global : 3
```

Guess The Output ?

```
a=0
if a == 0:
   b = 1
def my_function(c):
   d = 3
   print(c)
    print(d)
my_function(7)
print(a)
print(b)
print(c)
print(d)
```

Output:

7
3
0
1
NameError!

Guess The Output ?

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a, b, x, y = 1, 15, 3,4
foo(17, 4)
print(a, b, x, y)
```

Output:

42 17 4 17
42 15 3 4

Argument Passing

- There are **two** ways to pass arguments/parameters to function calls in **C programming**:
  - **Call by value**
  - **Call by reference.**

Call By Value

- In **Call by value**, **original value is not modified**.
- In **Call by value**, the value being passed to the function is locally stored by the function parameter as **formal argument**
- So , if we change the value of **formal argument**, it is changed for the **current function** only.
- These changes are not reflected in the **actual argument's** value

Call By Reference

- In **Call by reference** , the location (address) of **actual argument** is passed to **formal arguments**, hence any change made to formal arguments will also reflect in actual arguments.

- In **Call by reference**, **original value is modified** because we pass reference (address).

What About Python ?

- When asked whether **Python** function calling model is "**call-by-value**" or "**call-by-reference**", the correct answer is: **neither**.

- What Python uses , is actually called "**call-by-object-reference**"

A Quick Recap Of Variables

- We know that everything in **Python** is an **object**.
- All **numbers** , **strings** , **lists** , **tuples** etc in **Python** are objects.
- Now , recall , what happens when we write the following statement in **Python**:

  x=10

- An **object** is created in **heap** , storing the value **10** and **x** becomes the reference to that **object**.

A Quick Recap Of Variables

- Also we must recall that in **Python** we have 2 types of data : **mutable** and **immutable**.

- **Immutable types** are those which do not allow modification in object's data and examples are **int** , **float** , **string** ,**tuple** etc

- **Mutable types** are those which allow us to modify object's data and examples are **list** and **dictionary**

What Is Call By Object Reference ?

- Now , when we pass **immutable** arguments like **integers**, **strings** or **tuples** to a function, the passing acts like **call-by-value.**

- The ***object reference is passed*** to the function parameters.

- They can't be changed within the function, because they can't be changed at all, i.e. they are **immutable**.

What Is Call By Object Reference ?

- It's different, if we pass **mutable arguments**.
- They are also **passed by object reference**, but they can be **changed in place** in the function.
- If we pass a **list** to a function, elements of that **list** can be changed in place, i.e. the **list** will be changed even in the caller's scope.

Guess The Output ?

```
def show(a):
    print("Inside show , a is",a," It's id is",id(a))

a=10
print("Outside show, a i
show(a)
```

Since Python uses Pass by object reference , so when we passed a , Python passed the address of the object pointed by a and this address was received by the formal variable a in the function's argument list. So both the references are pointing to the same object

Output:

```
Outside show, a is 10  It's id is 8791162737984
Inside show , a is 10  It's id is 8791162737984
```

Guess The Output ?

```
def increment(a):
    a=a+1
a=10
increment(a)
print(a)
```

Output:

10

When we pass **a** to **increment(a)**, the function has the local variable **a** referring to the same object. Since integer is **immutable**, so Python is not able to **modify** the object's value to **11** in place and thus it created a new object. But the original variable **a** is still referring to the **same object** with the value **10**

Guess The Output ?

```
def show(mynumbers):
    print("Inside show , mynumbers is",mynumbers)
    mynumbers.append(40)
    print("Inside show , mynumbe
mynumbers=[10,20,30]
print("Before calling show, m
show(mynumbers)
print("After calling show, my
```

Since list is a mutable type , so any change made in the formal reference mynumbers does not create a new object in memory . Rather it changes the data stored in original list

Output:

```
Before calling show, mynumbers is [10, 20, 30]
Inside show , mynumbers is [10, 20, 30]
Inside show , mynumbers is [10, 20, 30, 40]
After calling show, mynumbers is [10, 20, 30, 40]
```

Guess The Output ?

```
def show(mynumbers):
    mynumbers=[50,60,70]
    print("Inside show , mynumbers is",mynumbers)
mynumbers=[10,20,30]
print("Before calling show, mynumbers is",mynumbers)
show(mynumbers)
print("After calling show, mynumbers is",mynumbers)
```

If we create a new object inside the function , then Python will make the formal reference mynumbers refer to that new object but the actual argument mynumbers , will still be refering to the actual object

Output:

```
Before calling show, mynumbers is [10, 20, 30]
Inside show , mynumbers is [50, 60, 70]
After calling show, mynumbers is [10, 20, 30]
```

Guess The Output ?

```
def foo(x):
    x.append (3)
    x = [8]
    return x
x=[1, 5]
y= foo(x)
print(x)
print(y)
```

Output:

[1,5,3]

[8]

Guess The Output ?

```
def swap(a,b):
    a,b=b,a
a=10
b=20
swap(a,b)
print(a)
print(b)
```

Output:

```
10
20
```

Guess The Output ?

```
def changetoupper(s):
    s=s.upper()
s="bhopal"
changetoupper(s)
print(s)
```

Output:

bhopal

Guess The Output ?

```
def changetoupper(s):
    s=s.upper()
    return s
s="bhopal"
s=changetoupper(s)
print(s)
```

Output:

BHOPAL

PYTHON

LECTURE 22

Today's Agenda

- **User Defined Functions-IV**
  - Anonymous Functions OR Lambda Function

What Are Anonymous Functions ?

- An **anonymous** function is a function that is ***defined without a name***.

- While **normal functions** are defined using the **def** keyword, we define **anonymous functions** using the **lambda** keyword.

- Hence, **anonymous functions** are also called **lambda** functions.

Syntax Of Lambda Functions

- **Syntax:**

  lambda [arg1,arg2,..]:[expression]

  - ***lambda*** is a keyword/operator and can have any number of arguments.
  - But it can have only one **expression**.
  - Python evaluates the **expression** and returns the result automatically.

What Is An Expression ?

- An **expression** here is anything that can return some value.
- The following items qualify as expressions.
  - **Arithmetic operations** like a+b and a**b
  - **Function calls** like sum(a,b)
  - **A print statement** like print(“Hello”)

So, What Can Be Written In Lambda Expression ?

- **Assignment statements cannot be used in lambda** , because they don't return anything, not even **None** (null).
- Simple things such as **mathematical operations**, **string operations** etc. are OK in a lambda.
- **Function calls** are expressions, so it's OK to put a function call in a lambda, and to pass arguments to that function.
- Even **functions** that return **None**, like the ***print*** function in Python 3, can be used in a lambda.
- **Single line if – else** is also allowed as it also evaluates the condition and returns the result of **true** or **false** expression

How To Create Lambda Functions ?

- Suppose, we want to make a **function** which will ***calculate sum of two numbers.***

- In **normal approach** we will do as shown below:

```
def add(a,b):
        return a+b
```

- In case of **lambda function** we will write it as:

```
lambda a,b: a+b
```

Why To Create Lambda Functions ?

- A very common doubt is that when we can define our functions using **def** keyword , then **why we require lambda functions ?**
- The most common use for **lambda functions** is in code that requires **a simple one-line function**, where it would be an overkill to write a complete **normal function**.
- We will explore it in more detail when we will discuss **two** very important functions in **Python** called **map( )** and **filter( )**

How To Use Lambda Functions ?

- There are 2 ways to use a **Lambda Function**.
  - **Using it anonymously in inline mode**
  - **Using it by assigning it to a variable**

How To Use Lambda Functions ?

- Using it as **anonymous function**

```
print((lambda a,b: a+b)(2,3))
```

Output:

```
5
```

How To Use Lambda Functions ?

- Using it by assigning it to a variable

```
sum=lambda a,b: a+b

print(sum(2,3))
print(sum(5,9))
```

Output:

```
5
14
```

What is happening in this code ?

The statement **lambda a,b:a+b** , is creating a **FUNCTION OBJECT** and returning that object . The variable **sum** is referring to that object. Now when we write **sum(2,3)**, it behaves like function call

Guess The Output ?

```
sum=lambda a,b: a+b

print(type(sum))
print(sum)
```

Since functions also are objects in Python , so they have their a unique memory address as well as their corresponding class as function

Output:

```
<class 'function'>
<function <lambda> at 0x0000000000050C1E0>
```

Example

```
squareit=lambda a: a*a

print(squareit(25))
print(squareit(10))
```

Output:

```
625
100
```

Example

```
import math
sqrt=lambda a: math.sqrt(a)

print(sqrt(25))
print(sqrt(10))
```

Output:

```
5.0
3.1622776601683795
```

Exercise

- **Write a lambda function that returns the first character of the string passed to it as argument**

Solution:

```
firstchar=lambda str: str[0]

print("First character of Bhopal :",firstchar("Bhopal"))
print("First character of Sachin :",firstchar("Sachin"))
```

Output:

```
First character of Bhopal : B
First character of Sachin : S
```

Exercise

- **Write a lambda function that returns the last character of the string passed to it as argument**

Solution:

```
lastchar=lambda str: str[-1]

print("Last character of Bhopal :",lastchar("Bhopal"))
print("Last character of Sachin :",lastchar("Sachin"))
```

Output:

```
Last character of Bhopal : l
Last character of Sachin : n
```

Exercise

- **Write a lambda function that returns True or False depending on whether the number passed to it as argument is even or odd**

Solution:

```
iseven=lambda n: n%2==0
print("10 is even :",iseven(10))
print("7 is even:",iseven(7))
```

Output:

```
10 is even : True
7 is even: False
```

Exercise

- **Write a lambda function that accepts 2 arguments and returns the greater amongst them**

Solution:

```
maxnum=lambda a,b: a if a>b else b
print("max amongst 10 and 20 :",maxnum(10,20))
print("max amongst 15 and 5 :",maxnum(15,5))
```

Output:

```
max amongst 10 and 20 : 20
max amongst 15 and 5 : 15
```

PYTHON

LECTURE 23

Today's Agenda

- **User Defined Functions V**
  - The map( ) Function
  - The filter( ) Function
  - Using map( ) and filter( ) with Lambda Expressions

What Is map( ) Function?

- As we have mentioned earlier, the advantage of the lambda operator can be seen when it is used in combination with the **map()** function.
- **map()** is a function which takes two arguments:
  - **r = map(func, iterable)**
- The first argument ***func*** is the **name of a function** and the second argument , ***iterable*** ,should be a **sequence** (e.g. a list , tuple ,string etc) or anything that can be used with ***for*** loop.
- ***map()*** applies the function ***func*** to all the elements of the sequence ***iterable***

What Is map( ) Function?

- To understand this , let's solve a problem.

- Suppose we want to define a function called **square( )** that can accept a number as argument and returns it's square.

- The definition of this function would be :

```
def square(num):
    return num**2
```

What Is map( ) Function?

- Now suppose we want to call this function for the following list of numbers:
  - **mynums=[1,2,3,4,5]**
- One way to do this , will be to use a **for** loop

```
mynums=[1,2,3,4,5]
for x in mynums:
   print(square(x))
```

Complete Code

```
def square(num):
    return num**2

mynums=[1,2,3,4,5]
for x in mynums:
    print(square(x))
```

Output:

```
1
4
9
16
25
```

Using map( ) Function

- Another way to solve the previous problem is to use the **map( )** function .
- The **map( )** function will accept 2 arguments from us.
  - The **first** argument will be the **name of the function square**
  - The **second** argument will be **the list mynums.**
- It will then apply the function **square** on every element of **mynum** and return the corresponding result as **map** object

Previous Code Using map( )

```
def square(num):
    return num**2

mynums=[1,2,3,4,5]
result=map(square,mynums)
print(result)
```

Output:

```
<map object at 0x00000000029030F0>
```

- As we can observe , the return value of **map( )** function is a **map object**
- To convert it into actual numbers we can pass it to the **function list( )**

Previous Code Using map( )

```
def square(num):
    return num**2

mynums=[1,2,3,4,5]
result=map(square,mynums)
sqrnum=list(result)
print(squrnum)
```

Output:

```
[1, 4, 9, 16, 25]
```

```
def square(num):
        return num**2

mynums=[1,2,3,4,5]
# we can club the 2 lines in 1 line

sqrnum=list(map(square,mynums))

print(sqrnum)
```

Previous Code Using map( )

To make it even shorter we can directly pass the **list( )** function to the function **print()**

```
def square(num):
    return num**2

mynums=[1,2,3,4,5]
print(list(map(square,mynums)))
```

Output:

```
[1, 4, 9, 16, 25]
```

Previous Code Using map( )

In case we want to **iterate** over this **list** , then we can use **for loop**

```
def square(num):
    return num**2

mynums=[1,2,3,4,5]
for x in map(square,mynums):
    print(x)
```

Output:

```
1
4
9
16
25
```

Exercise

- Write a function called **inspect( )** that accepts a string as argument and returns the word **EVEN** if the string is of **even length** and returns it's **first character** if the string is of **odd length**

Now call this function for first 3 month names

Solution

```
def inspect(mystring):
    if len(mystring)%2==0:
        return "EVEN"
    else:
        return mystring[0]

months=["January","February","March"]
print(list(map(inspect,months)))
```

Output:

```
['J', 'EVEN', 'M']
```

What Is filter( ) Function?

- Like **map( )** , **filter( )** is also a function that is very commonly used in **Python** .
- The function **filter ( )** takes 2 arguments:

  filter(function, sequence)

  - The **first argument** should be a **function** which must return a **boolean value**
  - The **second argument** should be a **sequence** of **items**.

- Now the function **filter( )** applies the function passed as argument to every **item** of the **sequence** passed as second argument.
- If the function returned **True** for that item , **filter( )** returns that **item** as part of it's return value otherwise the **item** is **not returned**.

What Is filter( ) Function?

- To understand this , let's solve a problem.

- Suppose we want to define a function called **check_even( )** that can accept a **number** as argument and return **True** if it is even , otherwise it should return **False**

- The definition of this function would be :

```
def check_even(num):
    return num%2==0
```

What Is filter( ) Function?

- Now suppose we have a list of numbers and we want to extract only even numbers from this list
  - **mynums=[1,2,3,4,5,6]**
- One way to do this , will be to use a **for** loop

```
mynums=[1,2,3,4,5,6]
for x in mynums:
   if check_even(x):
         print(x)
```

Complete Code

```
def check_even(num):
    return num%2==0

mynums=[1,2,3,4,5,6]
for x in mynums:
    if check_even(x):
        print(x)
```

Output:

```
2
4
6
```

Using filter( ) Function

- Another way to solve the previous problem is to use the **filter( )** function .
- The **flter( )** function will accept 2 arguments from us.
  - The **first** argument will be the **name of the function check_even**
  - The **second** argument will be **the list mynums.**
- It will then apply the function **check_even** on every element of **mynum** and if **check_even** returned **True** for that element then **filter( )** will return that element as a part of it's return value otherwise that element will not be returned

Previous Code Using filter( )

```
def check_even(num):
    return num%2==0

mynums=[1,2,3,4,5,6]
print(filter(check_even,mynums))
```

Output:

```
<filter object at 0x00000000029F3F60>
```

- As we can observe , the return value of **filter( )** function is a **filter object**
- To convert it into actual numbers we can pass it to the **function list( )**

Previous Code Using filter( )

```
def check_even(num):
    return num%2==0

mynums=[1,2,3,4,5,6]
print(list(filter(check_even,mynums)))
```

Output:

```
[2, 4, 6]
```

Previous Code Using filter( )

In case we want to **iterate** over this **list** , then we can use **for loop** as shown below:

```
def check_even(num):
    return num%2==0

mynums=[1,2,3,4,5,6]
for x in filter(check_even,mynums):
    print(x)
```

Output:

```
2
4
6
```

Guess The Output

```
def f1(num):
    return num*num

mynums=[1,2,3,4,5]
print(list(filter(f1,mynums)))
```

Output:

[1,2,3,4,5]

Ideally , the function passed to filter( ) should return a boolean value. But if it doesn't return boolean value , then whatever value it returns Python converts it to boolean . In our case for each value in mynums the return value will be it's square which is a non-zero value and thus assumed to be True. So all the elements are returned by filter()

Guess The Output

```
def f1(num):
    return num%2

mynums=[1,2,3,4,5]
print(list(filter(f1,mynums)))
```

Output:

[1,3,5]

For every even number the return value of the function f1( ) will be 0 which is assumed to be False and for every odd number the return value will be 1 which is assumed to be True . Thus filter( ) returns only those numbers for which f1( ) has returned 1.

Guess The Output

```
def f1(num):
    print("Hello")

mynums=[1,2,3,4,5]
print(list(filter(f1,mynums)))
```

Output:

Hello
Hello
Hello
Hello
Hello
[ ]

Hello is displayed 5 times because the filter( ) function has called f1( ) function 5 times. Now for each value in mynums , since f1( ) has not returned any value , by default it's return value is assumed to be None which is a representation of False. Thus filter( ) returned an empty list.

Guess The Output

```
def f1(num):
    pass

mynums=[1,2,3,4,5]
print(list(filter(f1,mynums)))
```

Output:

[ ]

For each value in **mynums** , since **f1( )** has not returned any value , by default it's return value is assumed to be **None** which is a representation of **False**. Thus **filter( )** returned an empty list.

Guess The Output

```
def f1():
    pass

mynums=[1,2,3,4,5]
print(list(filter(f1,mynums)))
```

The function filter() is trying to call f1( ) for every value in the list mynums. But since f1( ) is a non-parametrized function , this call generates TypeError

Output:

```
TypeError: f1() takes 0 positional arguments but 1 was given
```

Guess The Output

```
def f1():
    pass

mynums=[]
print(list(filter(f1,mynums)))
```

Output:

[ ]

Guess The Output

```
def f1(num):
    return num%2

mynums=[1,2,3,4,5]
print(list(map(f1,mynums)))
```

Output:

[1,0,1,0,1]

For every even number the return value of the function f1( ) will be 0 and for every odd number the return value will be 1. Thus map( ) has returned a list containing 1 and 0 for each number in mynums based upon even and odd.

Guess The Output

```
def f1(num):
    pass

mynums=[1,2,3,4,5]
print(list(map(f1,mynums)))
```

Since f1( ) is not returning anything , so it's return value by default is assumed to be None and because map( ) has internally called f1 () 5 times , so the list returned contains None 5 times

Output:

[None,None,None,None ,None]

Guess The Output

```
def f1():
    pass

mynums=[]
print(list(map(f1,mynums)))
```

Output:

[ ]

Using Lambda Expression With map( ) And filter( )

- The best use of **Lambda Expression** is to use it with **map( )** and **filter( )** functions

- Recall that the keyword **lambda** creates an **anonymous function** and returns it's **address**.

Using Lambda Expression With map( ) And filter( )

- So , we can pass this **lambda expression** as first argument to **map( )** and **filter()** functions , since their first argument is the a **function object reference**

- In this way , we wouldn't be required to specially create a separate function using the keyword **def**

Using Lambdas With map( )

```
def square(num):
        return num**2

mynums=[1,2,3,4,5]
sqrnum=list(map(square,mynums)
print(squrnum)
```

To convert the above code using **lambda** ,we have to do 2 changes:

1. Remove the function **square( )**
1. Rewrite this function as **lambda** in place of **first argument** while calling the function **map( )**

Following will be the resultant code:

```
mynums=[1,2,3,4,5]
sqrnum=list(map(lambda num: num*num,mynums))
print(sqrnum)
```

Exercise

- Write a **lambda expression** that accepts a string as argument and returns it's **first character**

Now use this lambda expression in **map( )** function to work on for first 3 month names

Solution

```
months=["January","February","March"]
print(list(map(lambda mystring: mystring[0],months)))
```

Output:

```
['J', 'F', 'M']
```

Exercise

- Write a lambda expression that accepts a string as argument and returns the word EVEN if the string is of even length and returns it's first character if the string is of odd length

Now use this lambda expression in map( ) function to work on for first 3 month names

Solution

```
months=["January","February","March"]
print(list(map(lambda mystring: "EVEN" if len(mystring)%2==0 else mystring[0],months)))
```

Output:

```
['J', 'EVEN', 'M']
```

Using Lambdas With filter( )

```
def check_even(num):
        return num%2==0

mynums=[1,2,3,4,5,6]
print(list(filter(check_even,mynums)))
```

To convert the above code using **lambda** ,we have to same 2 steps as before.

Following will be the resultant code:

```
mynums=[1,2,3,4,5,6]
print(list(filter(lambda num:num%2==0 ,mynums)))
```

Exercise

- Write a lambda expression that accepts a character as argument and returns True if it is a vowel otherwise False

Now ask the user to input his/her name and display only the vowels in the name . In case the name does not contain any vowel display the message No vowels in your name

Solution

```
name=input("Enter your name:")
vowels=list(filter(lambda ch: ch in "aeiou" ,name))
if len(vowels)==0:
    print("No vowels in your name")
else:
    print("Vowels in your name are:",vowels)
```

Output:

```
Enter your name:sachin
Vowels in your name are: ['a', 'i']
```