

Python



- General Introduction
- Language Syntax-
 - ✓ Identifiers
 - ✓ Naming rules
 - ✓ Reserved keywords
 - ✓ Comments
 - ✓ Python block Indentation
- Python Data Types
 - ✓ Numbers
 - ✓ Strings
 - ✓ List
 - ✓ Tuples
 - ✓ Dictionary
 - ✓ Bool



Input techniques

Inside every well-written large program is a well-written small program. -- Charles Antony Richard Hoare, computer scientist

INPUT TECHNIQUES

- Way to pass user defined input to a program.
- Remember the ***scanf*** function in C?
- That **scanf** evolved into **input** function in python.
- The input(string) method returns a line of user input as a string.
- The parameter is used as a prompt
- The string can be converted by using the conversion methods int(string), float(string), etc.



INPUT TECHNIQUES

Try and Learn

```
print("What's your name?")  
name = input("> ")  
print("What year were you born?")  
birthyear = int(input("> "))  
print("Hi %s! You are %d years old!"  
      % (name, 2015 - birthyear))
```

Raw input

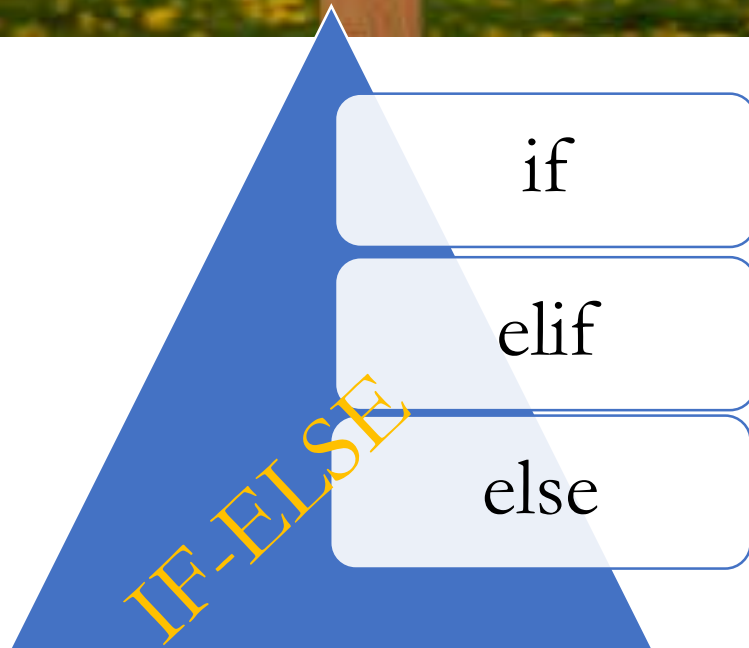
- Older version of the input function.



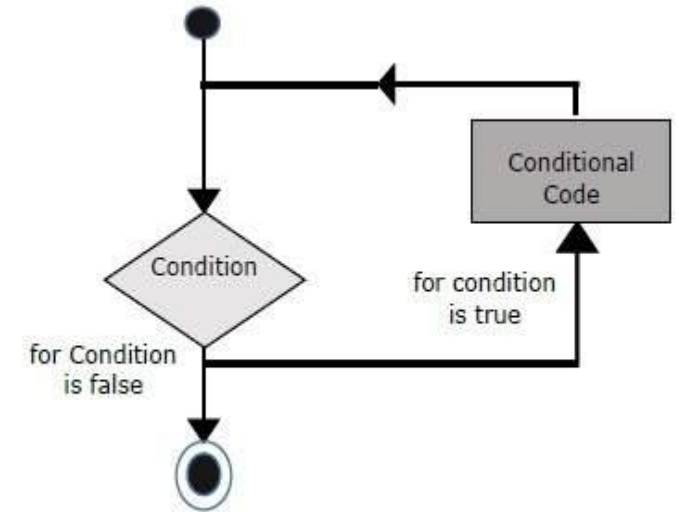
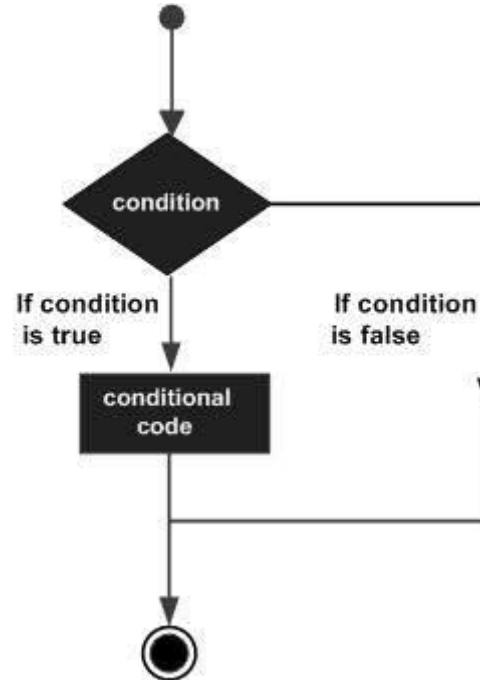
Decision Making

When a programming language is created that allows programmers to program in simple English, it will be discovered that programmers cannot speak English. -- unknown

Decision Making & Looping



- Decision making structures allow programmers to test one or more conditions and take actions accordingly.



Decision Making – IF-ELSE conditions

Try and Learn

```
var1 = int(input(">"))
```

```
if var1:
```

```
    print("True")
```

```
else:
```

```
    print("False")
```

- Input 3 Values : 10,0, -1 and check the Output.
- **Hint** : Python programming language assumes any non-zero and non-null values as TRUE, and any zero or null values as FALSE value.

Try and Learn

- Design and develop a IT calculation system for a company, such that when employee salary is entered, the Income Tax is calculated. Income tax slabs as follows:-

Lower Limit(INR)	Upper Limit(INR)	Income Tax %
100,000	300,000	0%
300000	500,000	5%
500,000	100,000,0	10%
>1000000		20% © DIPTARKO DAS SHARMA

Decision Making – IF-ELSE conditions

- At runtime, you need to enter atleast 4 employee salaries in all the four income range.

EmpName	Salary(INR)
A	10,000
B	100,000
C	3,99,000
D	20,00,000

How do you do range test within an IF-ELSE condition?

- Range() function is used to iterate through a loop , format being **range(start,stop,seq).**
- Range test can also be done as follows if($a \leq b \leq c$):
- Range test can also be done as follows : if($a \geq b$ and $a \leq c$)

Decision Making – WHILE & FOR LOOP

WHILE LOOP

- While loop, repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

```
>>> count = 0
```

```
>>> while (count < 9):
```

```
    print(count is:', count)
```

```
    count = 'The count + 1
```

FOR LOOP

- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

```
for letter in 'Python': # First Example
```

```
    print('Current Letter :', letter)
```

```
fruits = ['banana', 'apple', 'mango']
```

```
for fruit in fruits: # Second Example
```

```
    print('Current fruit :', fruit)
```

Decision Making –FOR LOOP

FOR LOOP(Contd)

- Before we check out this example, we will take up the function **range()**.
- **A lot of looping and decision making is done based on this function.**
- The built-in function **range()** is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.
- **range()** generates an iterator to progress integers starting with 0 upto n-1. To obtain a list object of the sequence, it is typecasted to **list()**. Now this list can be iterated using the for statement.

Example :-

```
>>> range(5)
      range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Try and learn

```
>>> fruits = ['banana', 'apple', 'mango']
>>> for i in range(len(fruits)):
    print(fruits[i])
```

Decision Making – MORE of LOOPS

Nested Loops:

- Python programming language allows the usage of one loop inside another loop. The following section shows a few examples to illustrate the concept.
- You can put any type of loop inside any other type of loop. For example a for loop can be inside a while loop or vice versa.

Try and learn:

Program to implement Tables from 1 to 10:-

#Trying a Nested Loop. Printing Multiplication tables from 1 to 10

i = 0

j = 0

for i in range(1,11):#First Loop to input the Table

 print("Starting Generating Table, for %d"%(i))

 for j in range(1,11): #Second Loop to take the Multiplicants

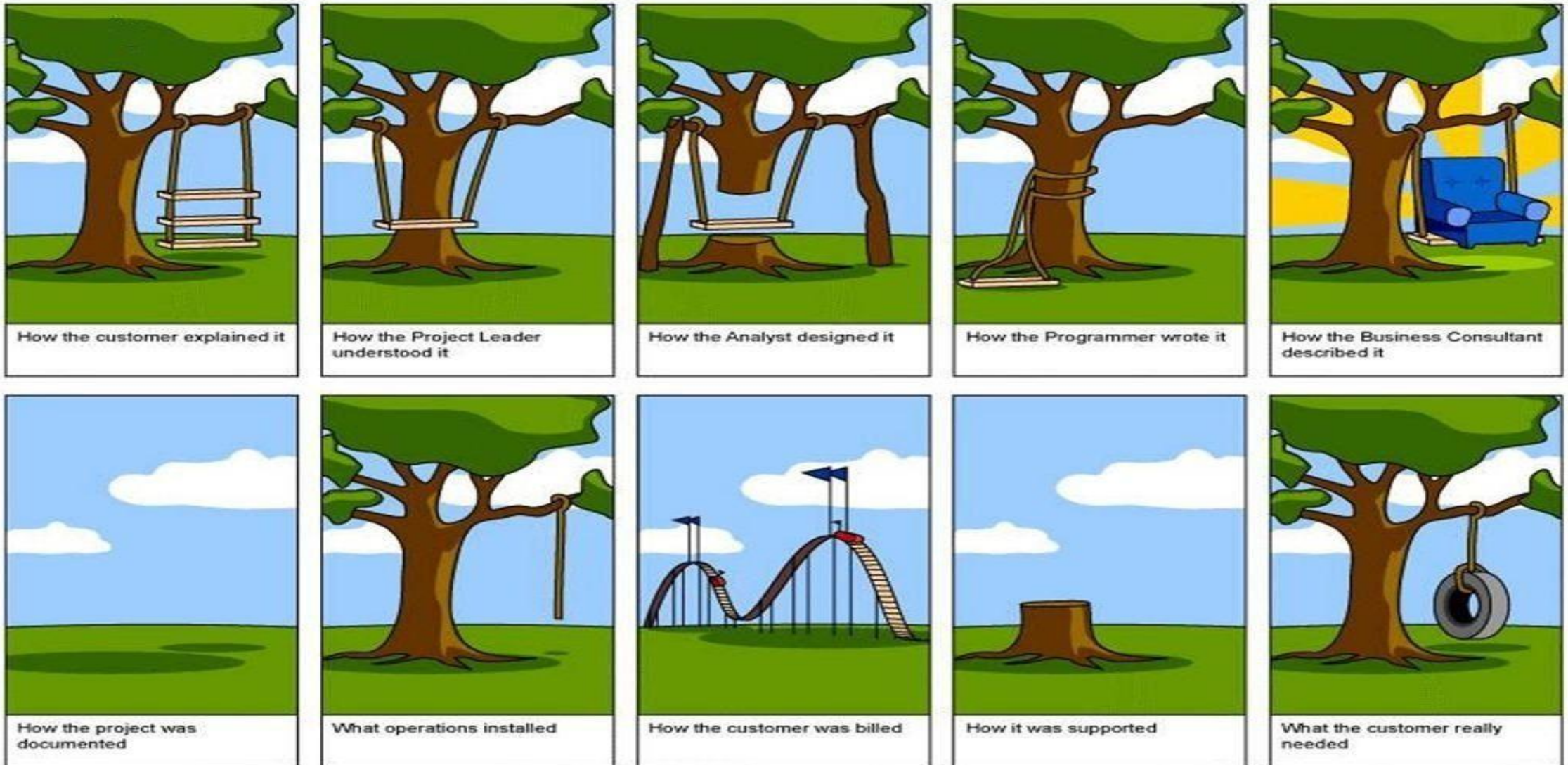
 print("%d X %d ="%(i,j),(i*j))



SNOOZE MODE

A Joke a Day, Keeps Boredom Away!!

Inside the World of Software Development



More Looping Techniques

Looping through Dictionaries

- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items()** method.

```
>>> fruits= {'apple': 'red', 'mango': 'yellow'}
```

```
>>> for k, v in fruits.items():
```

```
... print(k, v) ... apple red mango yellow
```

Looping through multiple sequences

- A quick look at the Zip function is needed to understand this section.
- **Zip** function, **Zips** together two Sequences into a sort of Key/Value pair.

Try and Learn

```
>>> names = ['ben', 'chen', 'yaqin']
```

```
>>> names
```

```
['ben', 'chen', 'yaqin']
```

```
>>> gender = [0, 0, 1]
```



More Looping Techniques

```
>>> name_age = list(zip(names,gender))  
>>> name_age  
[('ben', 0), ('chen', 0), ('yaqin', 1)]  
>>> name_age = tuple(zip(names,gender))  
>>> name_age  
(('ben', 0), ('chen', 0), ('yaqin', 1))  
>>> name_age = dict(zip(names,gender))  
>>> name_age  
{'ben': 0, 'chen': 0, 'yaqin': 1}
```



➤ Can you explain what happened in each of these cases?

More Looping Techniques

Now, how do we loop through such multiple sequences, obtained as part of zip?

Try and Learn

```
>>> name_age = zip(names,gender)
```

```
>>> name_age
```

```
<zip object at 0x0231DEE0>
```

```
>>> for i,j in name_age:
```

```
    print(i,j)
```

```
ben 0
```

```
chen 0
```

```
yaqin 1
```

Loop control Techniques

- The Loop control statements change the execution from its normal sequence. When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements:-
 - ✓ **Break**: Terminates the loop statement and transfers execution to the statement immediately following the loop.
 - ✓ **Continue** : Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
 - ✓ **Pass** : The pass statement is a null operation; nothing happens when it executes. The pass statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs.

Example of pass statement:

```
for letter in 'Python':  
    if letter == 'h':  
        pass  
        print("This is pass block")  
    print('Current Letter :', letter)
```

Loop control Techniques

Iterator :

- **Iterator** is an object which allows a programmer to traverse through all the elements of a collection, regardless of its specific implementation. In Python, an iterator object implements two methods, **iter()** and **next()**.
- String, List or Tuple objects can be used to create an Iterator.

```
list = [1,2,3,4]
```

```
it = iter(list) # this builds an iterator object
```

```
print (next(it)) #prints next available element in iterator
```

Iterator object can be traversed using regular **for** statement

```
for x in it:
```

```
    print (x, end=" ")
```

or using next() function

```
while True:
```

```
    try:
```

```
        print (next(it))
```

```
    except StopIteration:
```

```
        sys.exit() #you have to import sys module for this
```

Loop control Techniques

Generator :

- A **generator** is a function that produces or yields a sequence of values using yield method.
- When a generator function is called, it returns a generator object without even beginning execution of the function. When the next() method is called for the first time, the function starts executing until it reaches the yield statement, which returns the **yielded** value. The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.
- Saves a lot of memory in the process

Loop control Techniques

Generator :

Example

The following example defines a generator, which generates square numbers

```
>>> def square_numbers(nums):  
    for i in nums:  
        yield (i*i)  
  
>>> my_nums = square_numbers([1,2,3,4,5])  
>>> type(my_nums)  
<class 'generator'>  
>>> next(my_nums)  
1
```

PYTHON LOOPING CONCEPTS

Stimulants

1. What is the output of the following?

```
x = ['ab', 'cd']  
for i in x:  
    x.append(i.upper())  
    print(x)
```

- a) ['AB', 'CD'].
- b) ['ab', 'cd', 'AB', 'CD'].
- c) ['ab', 'cd'].
- d) none of the mentioned

2. What is the output of the following?

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

- a) 1 2
- b) 1 2 3
- c) error
- d) none of the mentioned

3. What is the output of the following?

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

- a) 2 4 6 8 10 ...
- b) 2 4
- c) 2 3
- d) Error

4. What is the output of the following?

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

- a) no output
- b) i i i i i ...
- c) a b c d e f
- d) abcdef

5. What does print(i) return in the following?

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

- a) a B C D
- b) a b c d
- c) A B C D
- d) error

6. What is the output of the following?

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

- a) a b c d
- b) 0 1 2 3
- c) error
- d) none of the mentioned

7. What will be the output of the below :

for i in range(float('inf')):

print (i)

a) 0.0 0.1 0.2 0.3 ...

b) 0 1 2 3 ...

c) 0.0 1.0 2.0 3.0 ...

d) none of the mentioned

8. What is the output of the following?

x = 'abcd'

for i in range(x):

print(i)

a) a b c d

b) 0 1 2 3

c) error

d) none of the mentioned



FUNCTIONS

"To err is human, but to really foul things up you need a computer." (Paul R. Ehrlich)

What is a Function?

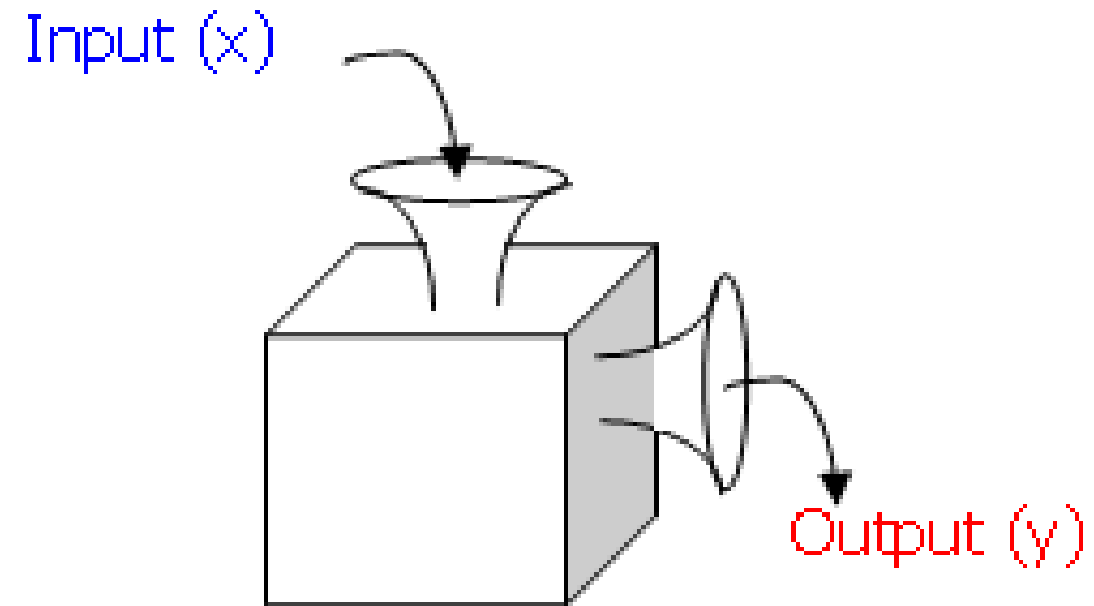
- Lets try writing a function and calling it. We will get to understand a function first hand , by writing it.

```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def hello_world():
        print("Hello World")
        print("My life is crazy")
end;
SyntaxError: invalid syntax
>>> def hello_world():
        print("Hello World")
        print("My life is crazy")

>>> hello_world()
Hello World
My life is crazy
>>> |
```

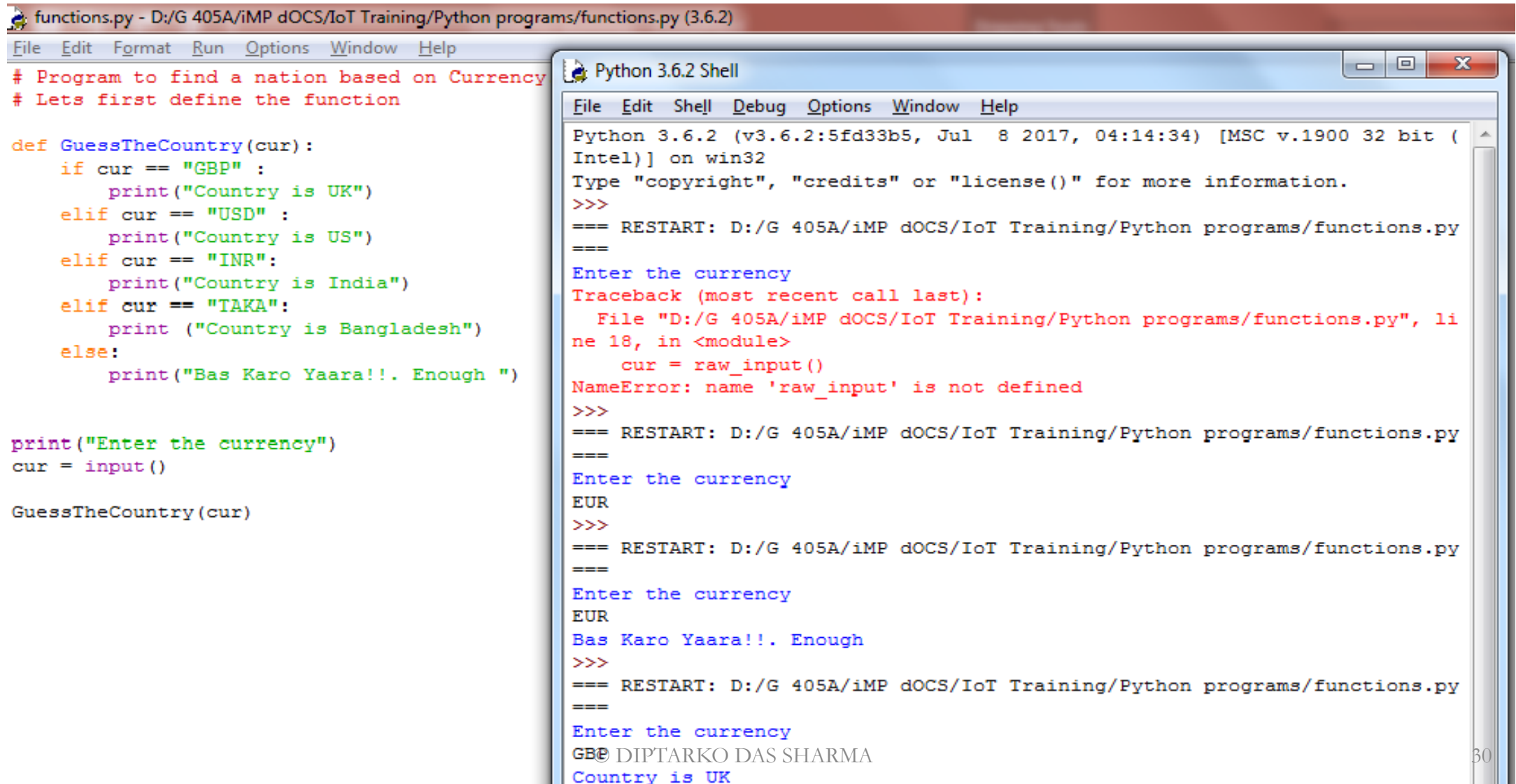
- As the above example proves, a function is thus a group of statements clubbed together , reusable and used to perform a specific task.
- Function begins with a keyword **def**(remember reserved keywords in the previous sections??) and are enclosed with **parentheses()** and a **:**

What is a Function?



What is a Function?

- Now let's consider a scenario, you are required to find out a Country's name based on its Currency. And you need to do this time and again, how would you take it up?



The image shows a Python IDE window titled 'functions.py - D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py (3.6.2)'. The code in the editor is as follows:

```
# Program to find a nation based on Currency
# Lets first define the function

def GuessTheCountry(cur):
    if cur == "GBP" :
        print("Country is UK")
    elif cur == "USD" :
        print("Country is US")
    elif cur == "INR":
        print("Country is India")
    elif cur == "TAKA":
        print ("Country is Bangladesh")
    else:
        print("Bas Karo Yaara!! . Enough ")

print("Enter the currency")
cur = input()

GuessTheCountry(cur)
```

Overlaid on the right is a 'Python 3.6.2 Shell' window showing the execution of the script. It displays the program's output and the error messages from the first two attempts to run it.

```
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py ===
Enter the currency
Traceback (most recent call last):
  File "D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py", line 18, in <module>
    cur = raw_input()
NameError: name 'raw_input' is not defined
>>>
=== RESTART: D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py ===
Enter the currency
EUR
>>>
=== RESTART: D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py ===
Enter the currency
EUR
Bas Karo Yaara!! . Enough
>>>
=== RESTART: D:/G 405A/iMP dOCS/IoT Training/Python programs/functions.py ===
Enter the currency
GBP
Country is UK
```

At the bottom right of the shell window, the text '© DIPTARKO DAS SHARMA' is visible.

What is a Function?

- Why did we do all this? Just to show how functions “function”, how we can pass certain inputs , and then invoke a function.
- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.
- You can call it again and again. You pass inputs to a function by means of passing by reference or by values. We will come to these details shortly. Quick few more points.
- Any input parameters or arguments should be placed within these parentheses.
- A function can also invoke another function from within it.
- A function can also return back values.(more of it in the next few pages).
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Calling a function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Function definition is here

```
def printme( str ):
```

```
    print(str)
```

```
    return
```

Now you can call printme function

```
printme("I'm first call to user defined function!")
```

```
printme("Again second call to the same function")
```


Pass by ref Vs Pass by reference

- All parameters in the Python language are passed by reference. Any change within a function also reflects back in the calling function.

Example of pass by reference:

```
>>> l=[1,2,3]
>>> def changeList(list):
        print(id(list))
        list = list.append('A')
        return
>>> l
[1, 2, 3]
>>> id(l)
38227080
>>> changeList(l)
38227080
>>> l
[1, 2, 3, 'A']
```

Keyword Arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the **printme()** function in the following ways –

Try and Learn:

```
def printinfo( name, age ):
```

```
    "This prints a passed info into this function"
```

```
    print ("Name: ", name)
```

```
    print ("Age ", age)
```

```
    return
```

```
# Now you can call printinfo function
```

```
printinfo( age = 50, name = "miki" )
```

Default Arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

Try and Learn

```
def printinfo( name, age = 35 ):
```

```
    "This prints a passed info into this function"
```

```
    print ("Name: ", name)
```

```
    print ("Age ", age)
```

```
    return
```

```
# Now you can call printinfo function
```

```
printinfo( age = 50, name = "miki" )
```

```
printinfo( name = "miki" )
```

Variable Arguments

- User may need to process a function for more arguments than specified while defining the function. They are called variable-length arguments
- An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

function definition

```
def print_student_marks( name, *student_marks):
```

```
    print(name)
```

```
    for marks in student_marks:
```

```
        print(marks)
```

function invocation

```
print_student_marks('xyz', 90,80,70)
```

```
print_student_marks('abc', 90, 88)
```

Anonymous Functions

- You can use the *lambda* keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to *inline* statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Anonymous Functions

Syntax:

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

Example:

Function definition is here

sum = lambda arg1, arg2: arg1 + arg2

Now you can call sum as a function

print("Value of total : ", sum(10, 20))

print("Value of total : ", sum(20, 20))

Return statement

Syntax:

- The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.
- **Try and learn :-**
 - Write a function which takes the length and breadth of a rectangle and returns the area.

Scope of variables

- All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
- The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python –
 - ✓ Global variables : Variables defined outside all functions are global
 - ✓ Local variables : Variables defined in a function are local and cannot be accessed outside it
- Therefore same names can be used for local and global variables. Local variable shadows global variable.

Global variable Vs Local variable

Try and learn:

```
a = 99 # Global variable
def func():
    a = 88 # Local variable
    print(a)
func()
print(a)
```

Invoking a global variable inside a function:

- ✓ To access a global variable inside a function, use the statement `global <<Variable Name>>`.

Try and learn:

- ✓ Try the above example again, only make sure that this time the value of “a” is same as the global a

MAP()

- The map() function applies a given function to each item of an iterable (list, tuple etc.) and returns a list of the results.
- The syntax of map() function is:

```
map(function, iterable)
```

map() Parameters

- function - map() passes each item of the iterable to this function.
- **iterable** iterable which is to be mapped

We can pass multiple iterable to the map() function.

Return Value from map()

The map() function applies a given to function to each item of an iterable and returns a list of the results.

The returned value from map() (map object) then can be passed to functions like [list\(\)](#) (to create a list), [set\(\)](#) (to create a set) and so on.

EXAMPLE OF MAP()

How map() works

```
def calculateSquare(n):  
    return n*n  
numbers = (1, 2, 3, 4)  
result = map(calculateSquare, numbers)  
print(result)  
# converting map object to list  
numbersSquare = list(result)  
print(numbersSquare)
```

Result :

```
<map object at 0x7f722da129e8>  
[16, 1, 4, 9]
```

Using Lambda function with map()

```
numbers = (1, 2, 3, 4)  
result = map(lambda x: x*x, numbers)  
print(result)  
# converting map object to set  
numbersSquare = list(result)  
print(numbersSquare)
```

Result :

```
<map object at 0x7f722da129e8>  
[16, 1, 4, 9]
```

EXAMPLE OF MAP()

Passing multiple iterators to map()

In this example, corresponding items of two lists are added.

```
num1 = [4, 5, 6]num2 = [5, 6, 7]  
result = map(lambda n1, n2: n1+n2, num1, num2)  
print(list(result))
```

Result :
[9, 11, 13]

FILTER()

- The filter() function constructs an iterator from elements of an iterable for which a function returns true.
- In simple words, the filter() method filters the given iterable with the help of a function that tests each element in the iterable to be true or not.
- The syntax of filter() method is:

filter(function, iterable)

filter() Parameters

The filter() method takes two parameters:

function - function that tests if elements of an iterable returns true or false

If None, the function defaults to Identity function - which returns false if any elements are false

iterable - iterable which is to be filtered, could be sets, lists, tuples, or containers of any iterators

FILTER()

Return Value from filter()

The filter() method returns an iterator that passed the function check for each element in the iterable.

The filter() method is equivalent to:

```
# when function is defined  
(element for element in iterable if function(element))  
# when function is None  
(element for element in iterable if element)
```

```
# list of alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o'] # function that filters vowels  
def filterVowels(alphabet):  
    vowels = ['a', 'e', 'i', 'o', 'u']  
    if(alphabet in vowels):  
        return True  
    else:  
        return False  
filteredVowels = filter(filterVowels, alphabets)  
print("The filtered vowels are:")  
for vowel in filteredVowels:  
    print(vowel)
```

Reduce()

Reduce()

from functools import reduce

```
>>> nums = (1,2,3,4,5,6)
>>> sum = reduce(lambda a,b: a+b,
nums)
>>> sum
21
```

Partial Functions

- Partial functions involve fixing a certain number of arguments to a function, producing another function of fewer arguments. This can be particularly useful for customizing a function for specific tasks without having to repeatedly provide the same arguments.

Example of Partial Functions in Python

Python provides a convenient way to create partial functions using the `'functools.partial'` function

Partial Functions

```
from functools import partial
```

```
# Original function
```

```
def multiply(x, y):
```

```
    return x * y
```

```
# Create a new function that multiplies by 2
```

```
double = partial(multiply, 2)
```

```
print(double(5)) # Output: 10
```

```
print(double(10)) # Output: 20
```

In this example, the partial function fixes the first argument `x` to 2, resulting in a new function `double` that only requires the `y` argument.

Partial Functions – Use Case

- **1. Customizing Functions with Common Arguments**
- When you have a function that is frequently called with the same set of arguments, partial functions can reduce redundancy by pre-filling those common arguments.

```
from functools import partial  
  
def greet(greeting, name):  
    return f'{greeting}, {name}!'  
  
# Create a new function that always uses the greeting "Hello"  
say_hello = partial(greet, "Hello")  
  
print(say_hello("Alice")) # Output: Hello, Alice!  
print(say_hello("Bob"))  # Output: Hello, Bob!
```

Partial Functions – Use Case

4. Data Processing Pipelines

- In data processing and transformation pipelines, partial functions can help in creating customized data processing steps.

Example:

```
from functools import partial

def process_data(data, func):
    return [func(item) for item in data]

def scale(factor, value):
    return factor * value

data = [1, 2, 3, 4, 5]
# Create a new function that scales values by 2
scale_by_2 = partial(scale, 2)

processed_data = process_data(data, scale_by_2)
print(processed_data) # Output: [2, 4, 6, 8, 10]
```

Currying Functions

Currying is the process of transforming a function that takes multiple arguments into a series of functions that each take a single argument. It allows the partial application of functions in a more granular manner, and can lead to more reusable and modular code.

Example of Currying

Currying can be manually implemented in Python or achieved through functional programming techniques.

Currying Functions

Original function

```
def multiply(x, y):  
    return x * y
```

Curried version of the function

```
def curried_multiply(x):  
    def inner(y):  
        return x * y  
    return inner
```

Using the curried function

```
double = curried_multiply(2)  
print(double(5)) # Output: 10  
print(double(10)) # Output: 20
```

Stimulants

1. Write a Python program to check if a list is empty (Hint : Try using a Lambda Function)
2. Consider a Fish Tuple as follows :- 'blowfish', 'clownfish', 'catfish', 'octopus'. Write a program that creates a list of all fish except 'octopus'.
3. ***Read in an alphabet. Write a program to check if it is a Vowel or a Consonant.***
 1. Write a Python program to construct the following pattern, using a nested loop number.

```
1
2 2
3 3 3
4 4 4 4
```

5. What does the following lines of code do?
A0 = dict(zip(('a','b','c','d','e'),(1,2,3,4,5)))
A1 = range(10)
A2 = sorted([i for i in A1 if i in A0])
A3 = sorted([A0[s] for s in A0])
A4 = [i for i in A1 if i in A3]
A5 = {i:i*i for i in A1}
A6 = [[i,i*i] for i in A1]

6. Which are the advantages of functions in python?

- a) Reducing duplication of code
- b) Decomposing complex problems into simpler pieces
- c) Improving clarity of the code
- d) All of the mentioned

7. What is the output of the below program?

```
x = 50
def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)
func()
print('Value of x is', x)
```

- a) x is 50
Changed global x to 2
Value of x is 50
- b) x is 50
Changed global x to 2
Value of x is 2
- c) x is 50
Changed global x to 50
Value of x is 50
- d) None of the mentioned

8. What is the output of below program?

```
def say(message, times = 1):  
    print(message * times)  
say('Hello')  
say('World', 5)
```

- a) Hello
WorldWorldWorldWorldWorld
- b) Hello
World 5
- c) Hello
World,World,World,World,World
- d) Hello

9. Where can a function be defined?

- a) Module
- b) Class
- c) Another function
- d) All of the mentioned