Introduction to Programming

"Everybody in this country should learn how to program a computer, should learn a computer language, because it teaches you how to think".

------ Steve Jobs

Source: http://www.businessinsider.com/the-best-quotes-from-the-lost-steve-jobs-interview-showing-this-weekend-2011

IF, IF…Else, nested if statements

- **IF statement:** The **if** statement contains a logical expression using which the data is compared and a decision is made based on the result.
- **Syntax:**

```
if expression:
    statement(s)
```

Sample: e.g.

```
a = 10
if a:
    print("1- got a true exp")
    print(a)
b = 0
if b:
    print("2 - got a true exp")
    print(b)
print("exiting...")
```

- If the Boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed the first set of code after the end of block is executed.

IF…Else, nested if statements

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

```
if expression:
    statement(s)
else:
    statement(s)
```

Sample: e.g.

```
score = int(input("enter the input: "))
if score<50:
    print("fail grade:")
else:
    print("Pass grade:")
```

Elif statements

- The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

- **Syntax:**

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

Sample: e.g

```
score = float(input("enter the input: "))
if score<=40.5:
        grade = score * 0.05
        print("fail grade:", grade)
elif score<70.0:
        grade = score * 0.10
        print("Pass grade:", grade)
else:
        grade = score * 0.15
        print("outstanding grade:",grade)
print("end")
```

- Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at the most one statement, there can be an arbitrary number of **elif** statements following an **if.**

- Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case.

Nested IF Statements

- There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if** construct.
- In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.
- **Syntax:**

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Sample: e.g.

```
num=int(input("enter a number"))
if num%2==0:
        if num%3==0:
                print("number is divisibe by 3 and 2")
        else:
                print("number divisible by 2 not by 3 ")
else:
        if num%3==0:
                print("number divisible by 3 not by 2")
        else:
                print("number not divisible by 2 and not by 3")
```

Single Statement Suites

- Groups of individual statements, which make a single code block are called **suites** in Python.
- If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.
- **Syntax:**

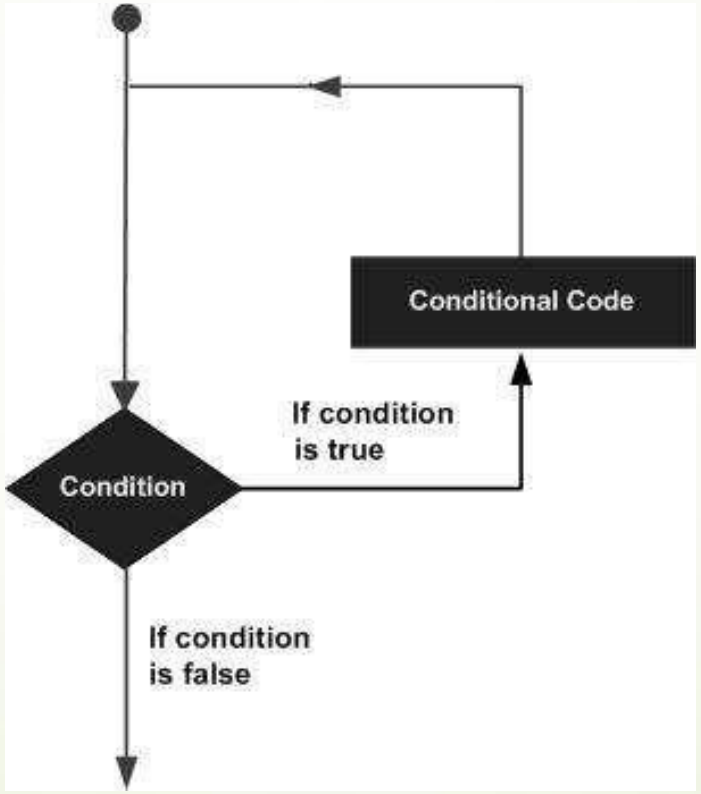
```
if expression:
suite
```

- **Sample:e.g.**

```
var = 100
if ( var == 100 ) : print ("Value of expression is 100")
print ("Good bye!")
```

Loops

- There may be a situation when you need to execute a block of code several number of times. A loop statement allows us to execute a statement or group of statements multiple times.



while Loop Statements

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- **Syntax:**

```
while expression:
   statement(s)
```

- The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- **Sample: e.g.**

```
count=0
while(count<10):
    print("count is :",count)
    count=count+1
```

The Infinite Loop

- A loop becomes infinite loop if a condition never becomes FALSE. Such a loop is called an infinite loop.
- An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.
- **Sample: e.g.**

```
var = 1
while var == 1 : # This constructs an infinite loop
   num = int(input("Enter a number :"))
   print ("You entered: ", num)
print ("Good bye!")
```

Using else Statement with Loops

- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.
- While loop can be terminated with a break statement. In such a case, the else part is ignored.
- Thus a while loop's else part runs if no break occurs and condition is false.
- **Sample: e.g.**

```
count=1
while(count<=3):
    print("loop inside:")
    count=count+1
else:
    print("false part")
print("exit")
```

For Loop Statements

- The for statement in Python has the ability to iterate over the items of any sequence.
- **Syntax:**

  for iterating_var in sequence:

    statements(s)

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the

  sequence is assigned to the iterating variable *iterating_var*.

- Next, the statements block is executed.
- Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted.
- **Sample: e.g.**

  for letter in 'Python':

    print ('Current Letter :', letter)

Nested Loops – For construct

- We need to place a loop inside another loop. It can be used on for & while construct.
- **Syntax:**

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
    statement(s)
```

- **Sample: e.g.**

```
for i in range(1,5):
    for j in range(1,5):
        k=i*j
        print (k,end='\n')
        #print ("\n",k)
print()
```

The print() function inner loop has end=' ' which appends a space instead of default newline. Hence, the numbers will appear in one row.

Nested Loops – While construct

- **Syntax:**

```
While expression:
    while expression:
        statements(s)
    statement(s)
```

- **Sample: e.g.**

```
n=int(input("enter a value:"))
i=1;
while i<=n:
   k=round(n)
   j=4
   while j<=k:
        if ( I % j==0):
        j+=1
   else: print (i)
    i+=1
```

Datatypes

Type	Mutable	Syntax example
str	Immutable	'Wikipedia' "Wikipedia" """Spanning multiple lines"""
list	Mutable	[4.0, 'string', True]
tuple	Immutable	(4.0, 'string', True)
set	Mutable	{4.0, 'string', True}
frozenset	Immutable	frozenset([4.0, 'string', True])
dict	Mutable	{'key1': 1.0, 3: False}
int	Immutable	42
float	Immutable	3.1415927
complex	Immutable	3+2.7j
bool	Immutable	True False

Numbers

- Number data types store numeric values.
- Number objects are created when you assign a value to them
- Python supports three different numerical types −
  - int (signed integers)
  - float (floating point real values)
  - complex (complex numbers)

```
Var1 = 1
var2 = 10
```

```
del var
del var_a, var_b
```

```
del var1[,var2[,var3[....,varN]]]
```

Numbers examples in Python

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

Lists, Dictionaries,Tuples

Lists

- Lists are the most versatile of Python's compound data types.
- Lists are ordered sequence of items.
- A list contains items separated by commas and enclosed within square brackets ([]).
- To some extent, lists are similar to arrays in C but are different by allowing the items to be of different types. (integer, float, string etc.).

- **List creation:**

1. **Empty list : my_list = [ ]**
2. **List of integers: my_list = [1,2,3]**
3. **List of floating type : my_list = [2.2, 4.0, 99.99, .45]**
4. **List of mixed data types: my_list = [1, “hai”, 9.99]**
5. **Nested list : [“ hello” , [3, 5 , 7.7] , [‘z’]]**

Accessing List Elements

- Lists can be accessed in two ways:
  - **List Index**
  - Negative Indexing
- In case of List Index technique use the index operator [] to access an item in a list.
- Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

  E.g.

  my_list = [1 , 2, 3, 4, 5]

  Now to access 3rd element : print (my_list[2])
- In case of nested lists we use nested indexing as shown in example.

  E.g .

  nes_list = [“welcome” , [2 , 0 , 1, 8]]

  Now to print 0 , we have to write : print (nes_list [1][1])

Accessing List Elements contd..

- Lists can be accessed in two ways:
  - List Index
  - **Negative Indexing**
- In case of Negative Indexing technique use the index operator [] to access an item in a list.
- Index starts from -1. So, a list having 5 elements will have index from -1 to -5.

  E.g.

  my_list = [‘d’,’j’,’k’,’l’,’n’,’g’].

  Now to print k: print (my_list[-4])
- In case of nested lists we use nested indexing as shown in example.

  E.g .

  my_list = ['d007','j','k','l','n','g'], [1,2,3,8.8,5.5]

  Now to print 8.8 , we have to write : print(my_list[-1][3])

  Now to print d007, we have to write print(my_list[-2][-6])

List Operations

- Some of the list operations are concatenation, repetition, comparison, membership operators, etc..

1. Concatenation: The "+" operator is used to concatenate lists.
2. Repetition: The "*" operator is used to repeat the list a number of times specified.
3. Comparison: The "==" operator is used to compare the exact contents of the list.
4. Membership: The "in" and "not in" operators are used to test the existence of items.

E.g.: n1 = [1, 2, 3, 4, 5] and

n2 = [3, 4, 5, 6, 7]

```
print(n1 == n2)
print(n1 + n2)
print(n1 * 3)
print(2 in n1)
print(2 in n2)
Print(-2 not in n1)
```

Outputs:

```
False
[1, 2, 3, 4, 5, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
True
False
True
```

Slicing Lists

- It is an operation that extracts certain elements from a list and forms a sub list possibly with a different number of indices and different index ranges.

The syntax for list slicing is : **[start:end:step]**

- where start, end, step parts of the syntax are integers and optional.
- They can be both positive and negative.
- Note that the value having the end index is not included in the slice.

E.g:

n = [1, 2, 3, 4, 5, 6, 7, 8]

print(n[1:5]) has values with indexes 1,2,3,and 4. so output is : [2,3,4,5]

print(n[:5]) has values starting from the default index i.e. 0 so output is: [1,2,3,4,5]

print(n[1:]) has values starting from index 1 till the default end index i.e. -1 so the output ls : [2,3,4,5,6,7,8] excluding 1 since it is a value at index 0.

print(n[:]) has all the values starting from default start index i.e. 0 to default end index i.e. -1 so the output is : [1,2,3,4,5,6,7,8].

Slicing Lists contd..

- The third index in a slice syntax is the step. It allows us to take every n-th value from a list.

E.g: n = [1, 2, 3, 4, 5, 6, 7, 8]

print(n[1:9:2]) → a slice having every 2nd element from the n list starting from 2nd element ,ending at 8th element is created. So output is [2,4,6,8].

print(n[::2]) → a slice is built taking every 2nd value from the begining till end of list. So the output is [1,3,5,7].

print(n[::1]) → this is just creating a copy of the list. So output is [1,2,3,4,5,6,7,8].

print(n[1::3]) → a slice is built for every 3rd element starting from 2nd element till the end of list. So output of the list is [2,5,8].

- Indexes can be negative numbers. Indexes with lower negative numbers must come first in the syntax.

E.g: **print(n[-1:-4])**→ returns an empty list. Now change index to **print(n[-4:-1])** to get the output [5,6,7].

print(n[::-1])→ creates a reversed list so output is [8,7,6,5,4,3,2,1].

Adding List Elements

- The elements can be added to the list by using built-in functions and methods or by specifying index value or also by using slicing.

E.g:

1. even=[2,4,6,8] Now to insert a value 1 instead of 2 we have to just write: **even[0]=1**.
2. Now to change the list to contain only odd numbers using slice operation I can write **even[1:4]=[3,5,7]** so that now print(even) will output[1,3,5,7].
3. Now to add an item 9 to the list using built-in method I have to write: **even.append(9)** so the value 9 is added to the end of list automatically.
4. Suppose I want to add 11, 13,15 also to the list then I have to use another built-in method extend to add all items together to the list and not individually. So **even.extend([11,13,15])** adds 3 more elements to the list.

- On printing the list finally we have **print(even)**→ [1,3,5,7,9,11,13,15] as output.

5. We can insert an item at a desired location by using a built-in method insert().

E.g: even=[2,10] so we write **even.insert(1,4)** to get sublist as [2,4,10].

6. Squeeze multiple items into list by writing **even[2:2]=[6,8]** to get [2,4,6,8,10] as O/P.

Deleting List Elements

- The elements can be removed from the list by using built-in functions and methods or by specifying index value using a **keyword del** or also by using slicing.

E.g:

1. my_list = ['d','j','007','k','l','n','g'] so to delete an item '007' we write **del my_list[2]**.
2. Now to delete multiple items from the list we write **del my_list[2:7]** to get ['d','j'].
3. Now to delete entire list using del keyword we write **del my_list**.
4. Now if you want an empty list then we write **del my_list[:].**
5. Now to remove an item from the list using built-in method we write **my_list.remove('007')**

- We can use another method pop() to remove an item from the list. The pop() method removes or returns the last item if index is not provided.

6. **print(my_list.pop(-2))** will remove 'n' from the list.
7. **print(my_list.pop(1))** will remove the 2nd element from the list with index being 1.
8. **print(my_list.pop())** will remove the last item from the list.
9. **my_list.clear()** will clear the contents and output an empty list.

Lists and Functions

- There are a number of built-in functions that can be used on lists that allow you to quickly look through a list without writing your own loops:
- >>> nums = [3, 41, 12, 9, 74, 15]
- >>> print(len(nums))
- 6
- >>> print(max(nums))
- 74
- >>> print(min(nums))
- 3
- >>> print(sum(nums))
- 154
- >>> print(sum(nums)/len(nums))
- 25

Contd..

- The sum() function only works when the list elements are numbers.
- (max(), len(), etc.) work with lists of strings and other types that can be comparable.

Example:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1
average = total / count
print('Average:', average)
```

OR

```
numlist = list()
while (True):
inp = input('Enter a number: ')
if inp == 'done': break
value = float(inp)
8.9. LISTS AND STRINGS 97
numlist.append(value)
average = sum(numlist) / len(numlist)
print('Average:', average)
```

In this program, we have count and total variables to keep the number and running total of the user's numbers as we repeatedly prompt the user for a number instead use lists to append a number everytime and compute sum and average at the end.

Lists and strings

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string.
- To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

- The list function breaks a string into individual letters.
- If you want to break a string into words, you can use the split method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

List with a delimiter – split and join

- You can call split with an optional argument called a *delimiter that specifies* which characters to use as word boundaries.
- The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

- Join takes list of strings and concatenates the elements. **(inverse of a list is join).**
- join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

- In this case the delimiter is a space character, so join puts a space between words.

Aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

- The association of a variable with an object is called a *reference*.
- An object with more than one reference has more than one name, so we say that the object is *aliased*.
- If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

- Although this behavior can be useful, it is error-prone.
- In general, it is safer to avoid aliasing when you are working with mutable objects.

List Arguments

- When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change.

Example: delete_head removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

- The parameter **t** and the variable **letters** are **aliases** for the **same object**.
- Use the append method modifies a list. Example:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
```

Contd..

```
>>> print(t2)
None
```

- the + operator creates a new list: **Example**

```
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

- This difference is important when you write functions that are supposed to modify lists.

Example:

```
def bad_delete_head(t):
    t = t[1:]                    # wrong
```

Q: Why?

Contd..

- This function *does not delete the head of a list* The slice operator creates a new list and the assignment makes t refer to it, but none of that has any effect on the list that was passed as an argument.

Q: What is the alternative solution?

A: An alternative is to write a function that creates and returns a new list.

Example: tail returns all but the first element of a list:

```
def tail(t):
    return t[1:]
```

- This function leaves the original list unmodified.
- Example:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Dictionary

- Python's dictionaries are collection of key-value pairs. (also called as items)
- A dictionary key can be of almost any Python type, but are usually numbers or strings.
- Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ({ }).
- values can be assigned and accessed using square braces ([]).
- Duplicate keys are not allowed and keys are immutable.

```
dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print (dict['one']) # Prints value for 'one' key
print (dict[2]) # Prints value for 2 key
print (tinydict) # Prints complete dictionary
print (tinydict.keys()) # Prints all the keys
print (tinydict.values()) # Prints all the values
```

outputs

This is one

This is two

{'dept':'sales', 'name':'john', 'code':'6734'}

['dept','name' ,'code',]

['sales','john' ,6734]

Contd..

- The order of the key-value pairs is not the same.
- In general, the order of items in a dictionary is unpredictable.

Q: Is it because the elements of a dictionary are never indexed with integer indices?

A: Nope!!! It is because the keys to look up the corresponding values:

- Key always maps to value as we saw in previous example. So order does not matter.
- If key is not in the dictionary then?? Throws an exception.

```
>>> print(tinydict['four'])
KeyError: 'four'
```

- The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(tinydict)
3
```

- The in operator works on dictionaries; it tells you whether something appears as a *key in the dictionary*.

```
>>> 'code' in tinydict
True
```

Contd..

- To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the in operator:

```
>>> vals = list(tinydict.values())
>>> 'john' in vals
True
>>> vals = list(tinydict.values())
>>> '9999' in vals
False
```

- The in operator uses different algorithms for lists and dictionaries.
- For lists, it uses a linear search algorithm.
- As the list gets longer, the search time gets longer in direct proportion to the length of the list.
- For dictionaries, Python uses an algorithm called a *hash table that has a remarkable property*.
- the in operator takes about the same amount of time no matter how many items there are in a dictionary.

Looping and dictionaries

- If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary.

Example:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

Example:

if we wanted to find all the entries in a dictionary with a value above ten, we could write:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print(key, counts[key])
```

- The for loop iterates through the *keys of the dictionary, so we must use the index* operator to retrieve the corresponding *value for each key.*
- *So in the output only jan 100 and annie 42 is displayed.*

Contd..

Steps to print the keys in alphabetical order:

1. **M**ake a list of the keys in the dictionary using the keys method available in dictionary objects.
2. **S**ort that list.
3. Loop through the sorted list.
4. Look up each key and print out key-value pairs in sorted order.

Example:

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

- In the output first you see the list of keys in unsorted order that we get from the keys method.
- Then we see the key-value pairs in order from the for loop.

Advanced Text Parsing

- If we assume to take a text file romoe.txt, some problems we face if split() is used:
- Since the Python split function looks for spaces and treats words as tokens separated by spaces, we would treat the words "soft!" and "soft" as *different words.*
- Also since the file has capitalization, we would treat "who" and "Who" as different words with different counts.

Soln: Use the string methods lower, punctuation, and translate.

1. **Translate:** The translate is the most subtle of the methods.

Syntax: line.translate(str.maketrans(fromstr, tostr, deletestr))

- *Replace the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr.*
- *The fromstr and tostr can be empty strings and the deletestr parameter can be omitted.*
- We will even let Python tell us the list of characters that it considers "punctuation":

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Code: advtxtprsng.py

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()

  line = line.translate(line.maketrans('', '', string.punctuation))
  line = line.lower()
  words = line.split()
  for word in words:
      if word not in counts:
          counts[word] = 1
      else:
          counts[word] += 1

print(counts)
```

Using this output We can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python *tuples*.

Tuples

- Tuple is another data type that is similar to the list.
- Tuples are also ordered collections.
- A tuple consists of a number of values separated by commas.
- Unlike lists, however, tuples are enclosed within parenthesis.
- The values stored in a tuple can be any type, and they are indexed by integers.
- Tuples are *immutable, comparable and hashable*.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
print (tuple) # Prints complete tuple
print (tuple[0]) # Prints first element of the tuple
print (tuple[1:3]) # Prints elements starting from 2nd till 3rd
print (tuple[2:]) # Prints elements starting from 3rd element
print (tinytuple * 2) # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

Creation of Tuples

- To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

- Without the comma Python treats ('a') as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

- Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
()
```

Contd..

- If the argument is a sequence (string, list, or tuple), the result of the call to tuple is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

- Trying to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

- You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>>t= ('a', 'b', 'c', 'd', 'e')
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

Comparing Tuples - DSU

- Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

- The sort function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.
- This feature lends itself to a pattern called ***DSU → Decorate Sort and Undecorate.***
- **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
- **Sort** the list of tuples using the Python built-in sort, and
- **Undecorate** by extracting the sorted elements of the sequence.

DSU example –dsu.py

- Sort list of words you want from longest to shortest:

Code:

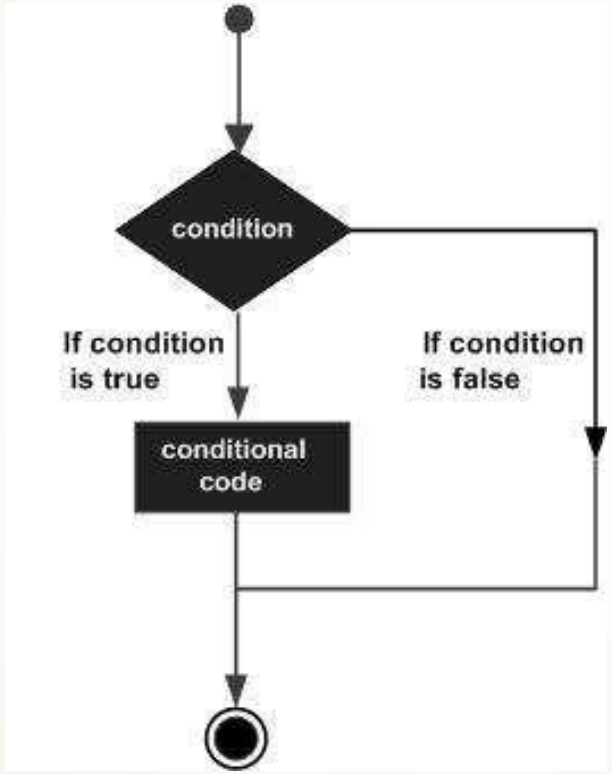
```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))
t.sort(reverse=True)
res = list()
for length, word in t:
    res.append(word)
print(res)
```

1. The first loop builds a list of tuples, where each tuple is a word preceded by its length.
2. sort compares the first element, length, first, and only considers the second element to break ties.
3. The keyword argument reverse=True tells sort to go in decreasing order.
4. The second loop traverses the list of tuples and builds a list of words in descending order of length.
5. The four-character words are sorted in *reverse alphabetical order,. Thus "what" appears before "soft".*

Output: ['yonder', 'window', 'breaks', 'light',
'what',
'soft', 'but', 'in']

Decision Making

- Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome.



- Python programming language assumes any **non-zero** and **non-null** values as TRUE, and any **zero** or **null values** as FALSE value.

Loop Control Statements

- Control statements change the execution from normal sequence.
- Sometimes we may want to terminate the current iteration or even whole loop without checking test expression.
- In these cases the following control statements are supported and used in Python:

1. Break:
2. Continue
3. Pass

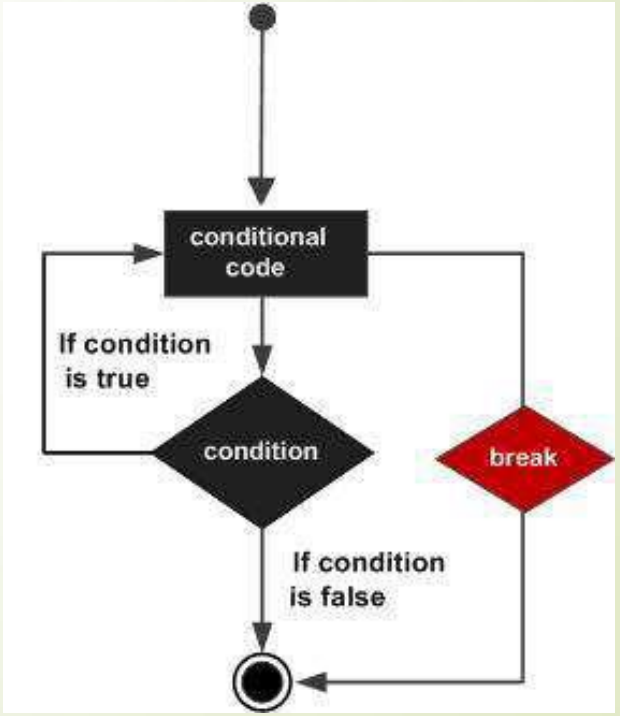
Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
pass statement	The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break Statement

- The most common use of break is when some external condition is triggered requiring a hasty exit from a loop.
- The break statement can be used in both *while and for loops.*
- If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.

- **Sample: e.g**.

```
for letter in 'python':
   if letter == 'h':
      break
   print("current letter is :",letter)
```

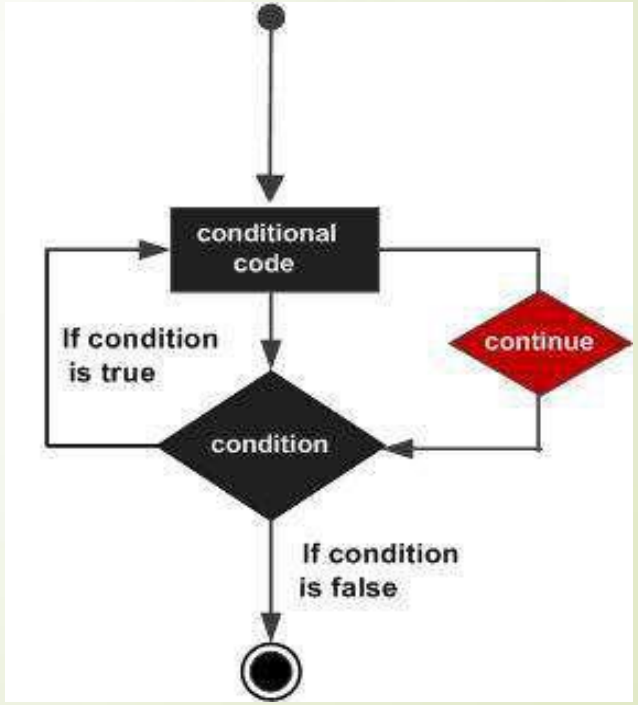


Continue Statement

- The continue statement in Python returns the control to the beginning of the current loop.
- When encountered, the loop starts next iteration without executing the remaining statements in the current iteration.
- The break statement can be used in both *while and for loops.*

- **Sample: e.g**.

```
for letter in 'python':
   if letter == 'h':
      continue
   print("current letter is :",letter)
```



Pass Statement

- In python, pass statement is a null operation; nothing happens when it executes.
- It is used as a place holder.
- *Whenever we want to implement a loop or a function in future but eventually now* you do not want to code anything then use a pass statement since the python interpreter will not allow you to have an empty body.

- **Sample: e.g**.

```
for letter in 'python':
  if letter == 'h':
     pass
     print (“this is pass block”)
  print("current letter is :",letter)
print(“good bye!”)
```

File Handling

How to read Information from a File

1. Open the file and associate the file with a file variable.
2. A command to read the information.
3. A command to close the file.

1. Opening Files

Prepares the file for reading:

A. Links the file variable with the physical file (references to the file variable are references to the physical file).

B. Positions the file pointer at the start of the file.

Format:

```
<file variable> = open(<file name>, "r")
```

Example:

```
inputFile = open("data.txt", "r")
```

OR

(Variable file name: entered by user at runtime)

```
filename = input("Enter name of input file: ")
inputFile = open(filename, "r")
```

2. Reading Information From Files

- Typically reading is done within the body of a loop
- Each execution of the loop will read a line from the file into a string

Format:

```
for <variable to store a string> in <name of file variable>:
    <Do something with the string read from file>
```

Example:

```
for line in inputFile:
    print(line)  # Echo file contents back onscreen
```

File handle as a sequence

- A file handle open for read can be treated as a sequence of strings
- a sequence is an ordered set

- EXAMPLE:

- **xfile = open('dat.txt')**
- **for cheese in xfile:**
- **   print cheese**

3. Closing The File

- Although a file is automatically closed when your program ends it is still a good style to explicitly close your file as soon as the program is done with it.
  - What if the program encounters a runtime error and crashes before it reaches the end? The input file may remain 'locked' an inaccessible state because it's still open.

- **Format:**

```
<name of file variable>.close()
```

- **Example:**

```
inputFile.close()
```

Reading From Files: Putting It All Together

File Name : `cmdinpfilnm.py`

Input files: `NEWS.txt`

```
inputFileName = input("Enter name of input file: ")
inputFile = open(inputFileName, "r")
print("Opening file", inputFileName, " for reading.")

for line in inputFile:
    sys.stdout.write(line)


inputFile.close()
print("Completed reading of file", inputFileName)
```

How to write Information to a File

1. Open the file and associate the file with a file variable (file is "locked" for writing).

2. A command to write the information.

3. A command to close the file.

Writing To A File: Putting It All Together

- Name of the online example: "`opwr.py`"
- Input file: "`letters.txt`"
- output file name: "`gpa.txt`"

```
inputFileName = input("Enter the name of input file to read the
                      grades from: ")
outputFileName = input("Enter the name of the output file to
                      record the GPA's to: ")

inputFile = open(inputFileName, "r")
outputFile = open(outputFileName, "w")

print("Opening file", inputFileName, " for reading.")
print("Opening file", outputFileName, " for writing.")
gpa = 0
```

Contd..

```
for line in inputFile:
    if (line[0] == "A"):
        gpa = 4
    elif (line[0] == "B"):
        gpa = 3
    elif (line[0] == "C"):
        gpa = 2
    elif (line[0] == "D"):
        gpa = 1
    elif (line[0] == "F"):
        gpa = 0
    else:
        gpa = -1
    temp = str (gpa)
    temp = temp + '\n'
    print (line[0], '\t', gpa)
```

Contd..

```
inputFile.close ()

outputFile.close ()

print ("Completed reading of file", inputFileName)

print ("Completed writing to file", outputFileName)
```

Data Processing: Files

- Files can be used to store complex data given that there exists a predefined format.

- Format of the example input file: '`employees.txt`'

  <Last name><SP>*<First Name>*,*<Occupation>*,*<Income>*

Example Program: datprocng.py

```
inputFile = open ("employees.txt", "r")


print ("Reading from file input.txt")
for line in inputFile:
    name,job,income = line.split(',')
    last,first = name.split()
    income = int(income)


    income = income + (income * BONUS)
    print("Name:      %s,      %s\t\t\tJob:%s\t\tBonus:%d\tIncome      $%.2f"
%(first,last,job,bonus,income))
print ("Completed reading of file input.txt")
inputFile.close()
```

CSV File Reading and Writing

- The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no “CSV standard”, so the format is operationally defined by the many applications which read and write it.
- The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel.
- The csv module’s reader and writer objects read and write sequences.
- Programmers can also read and write data in dictionary form using the DictReader and DictWriter classes.

Reading a CSV file

- In python, we use csv.reader() module to read the csv file. Here, we will show you how to read different types of csv files with different delimiter like quotes(""), pipe(|) and comma(,).
- We have a csv file called **pwd.csv** having default delimiter comma(,) with following data:

- SN, Name, City

  1, John, Washington

  2, Eric, Los Angeles

  3, Brad, Texas

Read people.csv file:

where delimiter is comma (,)

```
import csv
with open('pwd.csv', 'r') as csvFile:
    reader = csv.reader(csvFile)
    for row in reader:
        print(row)
  csvFile.close()
```

where custom delimiter (|)

```
import csv
csv.register_dialect('myDialect', delimiter = '|')

with open('rwd.csv', 'r') as f:
    reader = csv.reader(f, dialect='myDialect')
    for row in reader:
        print(row)
```

Extract specific data from the spreadsheet into lists

```
import csv
with open('Example.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    dates = [ ]
    colors = [ ]
    for row in readCSV:
        color = row[3]
        date = row[0]
        dates.append(date)
        colors.append(color)
    print(dates)
    print(colors)
```

Writing to a CSV file

- In python, we use csv.writer() module to write the csv file.
- A csv file called **some.csv** having default delimiter comma(,) with following data will be created.

- 2

- ,M,a,r,i,e

- ,C,a,l,i,f,o,r,n,i,a

Write into .csv files:

where delimiter is comma (,)

```
import csv
row = ['2', ' Marie', ' California']
with open('some.csv', 'w') as f:
    writer = csv.writer(f)
    writer.writerows(row)
f.close()
```

where custom delimiter (|)

```
import csv
person = [['SN', 'Person', 'DOB'],
['1', 'John', '18/1/2017'],
['2', 'Marie','19/2/2018'],
['3', 'Simon','20/3/2009']]
csv.register_dialect('myDialect',delimiter = '|')
with open('dob.csv', 'w') as f:
    writer = csv.writer(f, dialect='myDialect')
    for row in person:
        writer.writerow(row)
f.close()
```

Questions ??

THANK YOU ☺