A special note for Immutable data types

Immutable objects don't have the same id, and as a matter of fact this is not true for any type of objects that you define separately. Generally speaking, every time you define an object in Python, you'll create a new object with a new identity. However, for the sake of optimization (mostly) there are some exceptions for small integers (between -5 and 256) and interned strings, with a special length --usually less than 20 characters--* which are singletons and have the same id (actually one object with multiple pointers).

string interning is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned.

You can check this like following:

```
>>> 30 is (20 + 10)
True
>>> 300 is (200 + 100)
False
>>> 'aa' * 2 is 'a' * 4
True
>>> 'aa' * 20 is 'a' * 40
False
```

And for a custom object:

```
>>> class A:
...    pass
...
>>> A() is A() # Every time you create an instance you'll have a new instance with new identity
False
```

Also note that the is operator will check the object's identity, not the value. If you want to check the value you should use ==:

```
>>> 300 == 3*100
True
```

And since there is no such optimizational or interning rule for tuples or any mutable type for that matter, if you define two same tuples in any size they'll get their own identities, hence different objects:

```
>>> a = (1,)
>>> b = (1,)
>>>
>>> a is b
False
```

It's also worth mentioning that rules of "singleton integers" and "interned strings" are true even when they've been defined within an iterator.

```
>>> a = (100, 700, 400)
>>> b = (100, 700, 400)
>>> a[0] is b[0]
True
>>> a[1] is b[1]
False
```

Operator priorities in Python

Operator	**Description**
(expressions...), [expressions...], {key: value...}, {expressions...}	Binding or tuple display, list display, dictionary display, set display
x[index], x[index:index], x(arguments...),	Subscription, slicing, call, attribute reference

x.attribute	
await x	Await expression
**	Exponentiation
+x, -x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if – else	Conditional expression
lambda	Lambda expression

Keywords in Python

These are words that have some predefined meaning so can‘t be used as variable name or identifiers. Take a look at table that has all the keywords

and	exec	not	assert	finally	or
break	for	pass	class	from	print
continue	global	raise	def	if	return
del	import	try	elif	in	while
else	is	with	except	lambda	yield

Punctuators in python

Punctuators (separator) are the symbols that are used in programming languages to organized sentence structures, and indicate the rhythm and emphasis of expressions, statements and program structure. Common punctuators are

```
' ″ # \ ( ) [ ] { } @ , : . ` =
```

*Note: Literals, variables, keywords, operators and punctuators are collectively called **tokens** and it is smallest lexical unit of any programming language.*

String function

Some methods that are applicable for String data type are as follow

Method	**Description**	
len(string)	This function is used to find out courting of characters in a string	
capitalize()	returns a string with first letter capitalized. It doesn't modify the old string.	
count(substring, start=..., end=...)	returns the number of occurrences of a substring in the given string. Index in Python starts from 0, not 1.	
	Substring	string whose count is to be found.
	Start	starting index within the string where search starts.
	end	ending index within the string where search ends. end is excluded
endswith(suffix[, start[, end]])	returns True if strings ends with the specified suffix, false otherwise.	
	suffix	String or tuple of suffixes to be checked
	start	Beginning position where suffix is to be checked within the string.
	end	Ending position where suffix is to be checked within the string. end is excluded
find(sub[, start[, end]])	substring exists inside the string, it returns the lowest index where substring is found, -1 otherwise.	
	Sub	It's the substring to be searched in the str string.
	start and end	substring is searched within str[start:end]

index(sub[, start[, end]])	If substring exists inside the string, it returns the lowest index in the string where substring is found. If substring doesn't exist inside the string, it raises a ValueError exception.	
	Sub	substring to be searched in the string str.
	start and end	substring is searched within str[start:end], end is excluded
isalnum()	Return True if all characters in the string are alphanumeric, False if at least one character is not alphanumeric	
isalpha()	True if all characters in the string are alphabets (can be both lowercase and uppercase). False if at least one character is not alphabet.	
islower()	True if all alphabets that exist in the string are lowercase alphabets. False if the string contains at least one uppercase alphabet.	
isnumeric()	True if all characters in the string are numeric characters. False if at least one character is not a numeric character.	
isupper()	True if all characters in a string are uppercase characters. False if any characters in a string are lowercase characters	
lower()	returns the lowercased string from the given string. It converts all uppercase characters to lowercase. If no uppercase characters exist, it returns the original string.	
replace(old, new [, count])	The replace() method returns a copy of the string where old substring is replaced with the new substring. The original string is unchanged.	
	Old	old substring you want to replace
	New	new substring which would replace the old substring
	count	the number of times you want to replace the old substring with the new substring If count is not specified, replace() method replaces all occurrences of the old substring with the new substring.
upper()	returns the uppercased string from the given string. It converts all lowercase characters to uppercase. If no lowercase characters exist, it returns the original string.	

Decision Making

if statement

if statement contains a relational expression using which some condition is made and a decision is made based on the result of the comparison.

Syntax:
```
if condition:
      Statement(s)
```

If the condition is evaluated to true then the statements (which are part of body) are executed. If the condition is evaluated to false then statements (which are part of body) will be bypassed.

Important: Python uses indentation as its method of grouping statements instead of { } as we do in other programming languages.

An Example:
```
marks = 90.0
maxmarks = 100.0

if marks < maxmarks:
    print("Your marks are valid")

marks = 100.0
if marks == maxmarks:
    print("Marvellous! full marks")

marks = 0.0
if marks:
    print("Too upset to print")
print("Bye bye")
```

The above code will produce following output

```
Your marks are valid
Marvellous! full marks
Bye bye
```

From the above example we can drive one more conclusion that is a variable can be used as an expression, if it has zero value then it will be considered as false, for non zero values it will be considered as true. Similarly an empty String is also considered as false.

if-else statement:

The else part can be placed with if statement optionally. It contains the code to be executed when if condition is evaluated to false. Only one else can be placed with one if statement.

Syntax:

```
if condition:
      Statement(s) for if
else:
      Statement(s) for else
```

An Example:

```
marks = input("Please enter your marks ")
marks = float(marks)
maxmarks = 100.0

if marks <= maxmarks:
    print("your marks are valid")
else:
    print("your marks are invalid")
```

When user is providing marks less than or equal to 100 like 75.00

```
Please enter your marks 75.00
your marks are valid
```

On the other if user is providing marks above 100.00 like 135.00

```
Please enter your marks 135.00
your marks are invalid
```

The elif statement

It is a short form of if-else statement. We can place multiple elif statements with the if statement. Similar to the else statement it is also optional for if.

Syntax:

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

An Example:

```
marks = input("Please enter your marks ")
marks = float(marks)

if marks >= 80.00:
    print("A Grade")
elif (marks >= 60.00) & (marks < 80.00):
    print("B grade")
elif (marks >= 33.00) & (marks < 60.00):
    print("C grade")
else:
    print("Fail")
```
The above code will produce output as follow

Case 1:
```
Please enter your marks 98.00
A Grade
```
Case 2:
```
Please enter your marks 68.00
B Grade
```
Case 3:
```
Please enter your marks 55.00
C Grade
```
Case 4:
```
Please enter your marks 25.00
Fail
```
Note: if clause that consists of only a single line then it may go on the same line like

```
num = input("Please enter number ")
num = int(num)

if num %2 == 0: print("Even number")
print("Bye bye")
```

The above code will produce following output

Case 1:
```
Please enter number 28
Even number
Bye bye
```
Case 2:
```
Please enter number 27
Bye bye
```

Loops
A loop is used to execute a set of statements multiple times.

while loop
```
while condition:
      statement(s)
```

The condition may be any expression, that evaluates to true or any non--zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop body.

An Example:
```
count = 1
while count < 5:
    print("count is", count)
    count = count + 1
print("Loop is over")
```

The code given above will produce following output
```
count is 1
count is 2
count is 3
count is 4
Loop is over
```

If condition of while loop never evaluated to false then it will run forever means it will execute forever. To stop execution of that we have to press ctrl + C.

We can also use **else statement** with loop that will be executed when condition will be false. Note the else will not be executed if loop is terminated by `break` statement. We will look at the `break` statement after for loop.

An Example:
```
count = 1
while count < 5:
    print("count is ", count)
    count = count + 1
else:
    print("count is equal to 5");
print("bye bye")
```

The code given above will produce following output

```
count is  1
count is  2
count is  3
count is  4
cout is equal to 5
bye bye
```

for loop

It has the ability to iterate over the items of any sequence, such as a list or a string. It has following 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_var. Next, the statements block is executed. Each item in the list is assigned to iterating_var, and the statement(s) block is executed till the entire sequence is not traversed.

We will discuss about list data type in details in subsequent chapters. Before moving forward first take a look at the range() function.

An Example

```
Str1 = "python"
for i in Str1:
    print(i, end = ' ')
print('\n')
for i in range(len(str)):
    print(str[i],end = ' ')
```

The above code will produce following output

```
P y t h o n
P y t h o n
```

In the above example we have to discuss about two functions the first is len() that return total number of characters in a string, this function can also be used with list. range(n) function returns sequence starting from 0 to n-1.

Similar to the while loop, we can use **else statement** with for loop also that will be executed when condition will be false, also else will not be executed if loop is terminated with `break` statement.

An Example:

```
str = "python"
for i in str:
    print(i, end = ' ')
else:
    print("\nprinted all characters of ",str)
```

The above code will produce following output

```
p y t h o n
printed all characters of  python
```

Nested loop

Similar to other programming language, we can also use nested loop, means one loop can be nested inside another loop. We can place while loop in another while loop, we can place for loop inside another for loop, apart from that mixing is also allowed in nested means for inside while or while inside for is also allowed.

An Example:

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

That code will produce following output

```
0
01
012
0123
01234
```

break statement

This statement terminates the execution of loop and resumes execution at the next statement. It has power to break only on loop hence if it is used inside inner loop then it will break only nested loop, outer loop will remain unaffected from that.

An Example

```
x = input("Enter number to be checked for prime ")
x = int(x);
i = 2
while i < x:
    if x%i == 0:
        break
    i = i + 1
else:
    print(x,"is prime number")
if(i < x):
    print(x, "is not prime number")
```

The above code will print following output

```
Enter number to be checked for prime 4
4 is not prime number

Enter number to be checked for prime 7
7 is prime number
```

continue statement

It takes the control to the staring of the loop. It rejects all the remaining statements in the current iteration of the loop. It can be used with both while and for loop.

An Example

```
for i in range(21):
    i = i+1
    if i%2 == 1:
        continue      #taking loop to the next iteration
    print(i,end =' ')
```

The above code will print following output

```
2 4 6 8 10 12 14 16 18 20
```

pass statement

The pass statement represent a null operation or simply operation, nothing takes place when it is executed. It is a kind of placeholder. It is placed wherever we are not able to give implementation right now.