PYTHON CONVENTIONS

PEP: **PEP** stands for **Python** Enhancement Proposal. A**PEP** is a design document providing information to the **Python** community, or describing a new feature for **Python** or its processes or environment. The **PEP**should provide a concise technical specification of the feature and a rationale for the feature. **PEP 0** is the Index of Python Enhancement Proposals.

The Zen of Python: PEP 20.

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one and preferably only one obvious way to do it.
- Although that way may not be obvious at first.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are a great idea

The PSF

The Python Software Foundation is the organization behind Python.
CREATOR OF PYTHON: GUIDO VAN ROSSUM

PYTHON CONVENTIONS

NAMING

Python Identifiers

A Python identifier is a name *used to identify a Data Type, function, class, module or other object*. An identifier starts with a letter ***A** to **Z** or **a** to **z** or an **underscore** (_)* followed by zero or more **letters**, **underscores** and **digits** (0 to 9).

Python **does not allow** punctuation characters such as @, $, and % within identifiers.

Python is a ***case sensitive*** programming language. Thus, *Manpower* and *manpower* are two different identifiers in Python. Here are naming conventions for Python identifiers.

- Class names start with an uppercase letter.
- All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language defined special name.
-

RESERVED WORDS

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

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

PYTHON CONVENTIONS

LINES AND INDENTATION

Python does not use braces({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

MULTI-LINE STATEMENTS

Statements in Python typically end with a new line. Python, however, allows the use of the line continuation character (\) to denote that the line should continue. For example

```
>>>total= item_one + \
item_two + \
item_three
>>>
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example

```
>>>Days = ['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday']
```

QUOTATION IN PYTHON

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
The triple quotes are used to span the string across multiple lines.

```
>>>Word = 'word'
>>>sentence = "This is a sentence."
>>>paragraph = """This is a paragraph. It is
>>>made up of multiple lines and sentences."""
```

COMMENTS IN PYTHON

A hash sign **(#)** that is not inside a string literal is the beginning of a comment. All characters after the #, up to the end of the physical line, are part of the comment and the Python interpreter ignores them.

PYTHON CONVENTIONS

```
>>>#!/usr/bin/python3
>>># ----This is a comment -------
```

MULTIPLE STATEMENTS ON A SINGLE LINE

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block. Here is a sample snip using the semicolon

```
>>>import sys; x = 'foo'; sys.stdout.write(x + '\n')
>>> x=2; y=3; x+y
5
```

VARIABLE

Variable names in Python are actually **references**. They are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables. It is a Python convention to start your variables with a lowercase letter. A good variable name describes the data it contains. By default, we define variables in Python with assigning value.

```
>>> num = 2
>>> area = 58.7
>>> country = "Germany"
>>> z = 10 + 5j          # complex number
```

If you want to declare variables without assigning values, you can set it using '**None**'.

```
>>> counter = None
>>> index = None
```

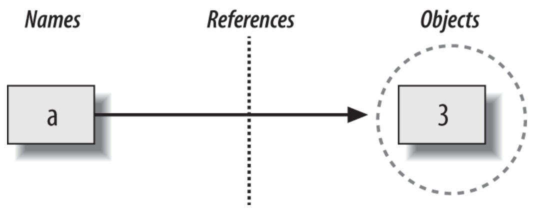
MULTIPLE ASSIGNMENT

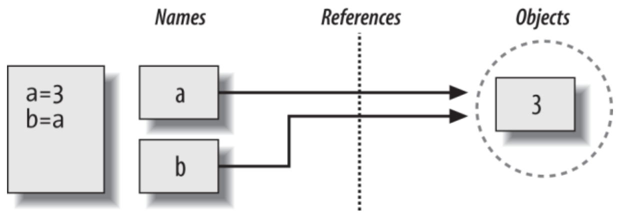
Python allows you to assign a single value to several variables simultaneously.

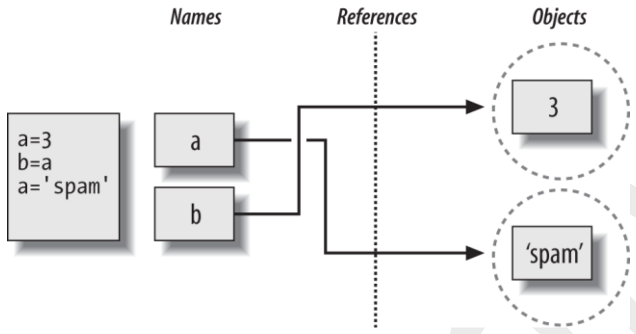
```
>>>a = b = c = 1
>>>x, y, pi, empname = 10, 20, 3.14, "John"
```

NAMESPACES

Name Spaces are assigned to data containers. In Python, all objects are assigned namespaces when they are used. This assignment is done using the assignment operator.





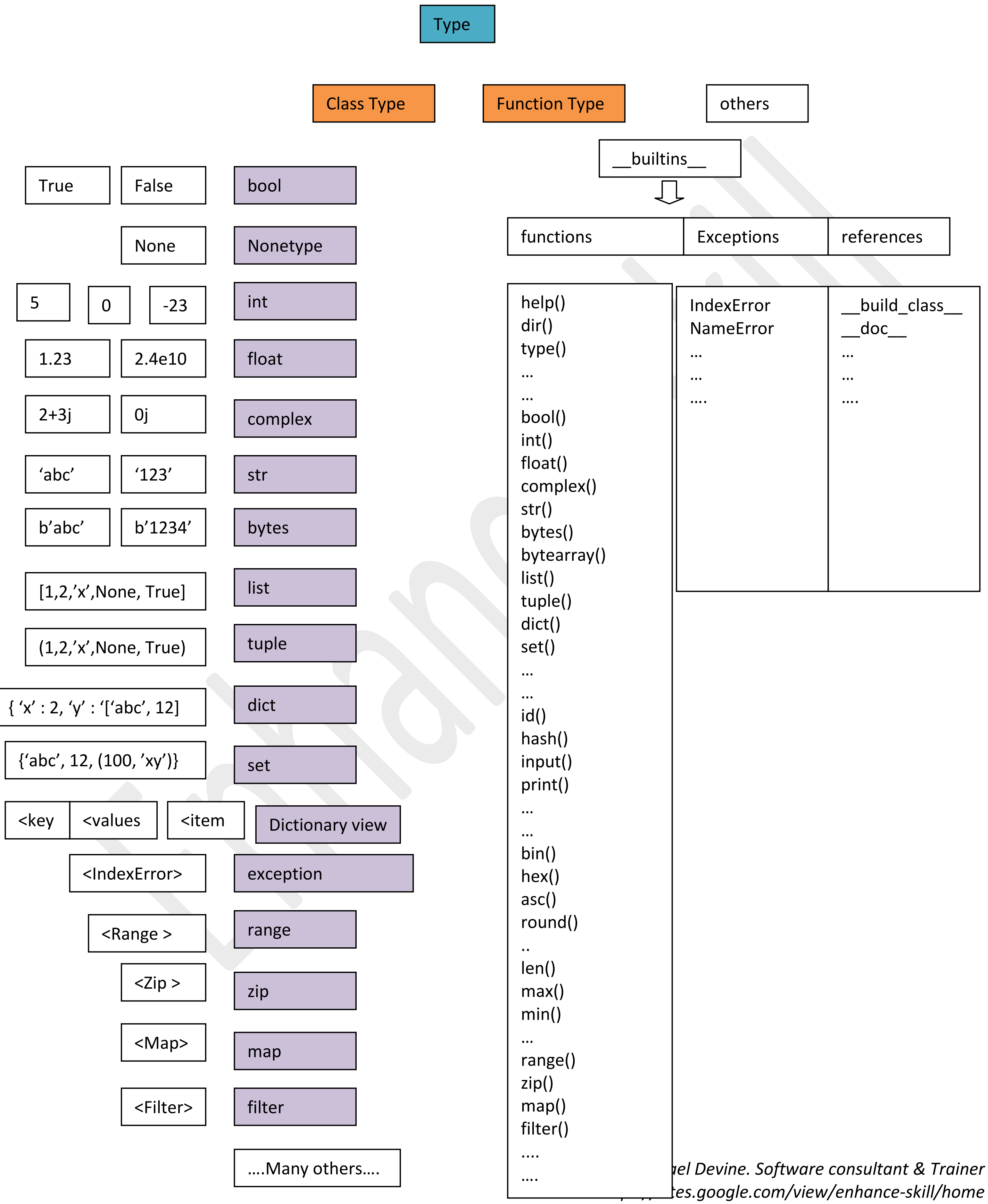


Removing a namespace or deleting a variable is done using the '**del**' statement.

```
>>>del a
>>>del pi
```

PYTHON CONVENTIONS

PYTHON DATATYPES



PYTHON CONVENTIONS

Everything is an **object** in Python, so there are types like *module*, *function*, *class*, *method*, *file*, and even *compiled code*.

In Python, every value has a datatype, but you don’t need to declare the datatype of variables. Based on each variable’s original assignment, Python figures out what type it is and keeps tracks of that internally. Python has many basic datatypes. Here are the important ones:

1. **Booleans** are either True or False.
2. **Numbers** can be integers (1 and 2), floats (1.1 and 1.2) and complex numbers (21+3j).
3. **Strings** are sequences of Unicode characters, *e.g.* an H T M L document.
4. **Bytes** and **byte arrays**, *e.g.* a J P E G image file.
5. **Lists** are ordered sequences of values.
6. **Tuples** are ordered, immutable sequences of values.
7. **Sets** are unordered container of values.
8. **Dictionaries** are unordered container of key-value pairs.
9. **None** as a null value.

BOOLEANS

Booleans are either **True** or **False**. Python has two constants, cleverly named True and False, which can be used to assign boolean values directly. Expressions can also evaluate to a Boolean value. Boolean is an inheritance of INT.

NONE

None is a special constant in Python. It is a null value. None is not the same as False. None is not 0. None is not an empty string. Comparing None to anything other than None will always return False. None is the only null value. It has its own datatype (NoneType). You can assign None to any variable. All variables whose value is None are equal to each other.

NONE IN A BOOLEAN CONTEXT:

In a Boolean context, None is false and not None is true. Boolean is an inheritance of INT.

PYTHON CONVENTIONS

NUMBERS

Python supports integers, floating point and Complex numbers.

```
>>> type(1)
<class 'int'>
>>> 1 + 1
2
>>> 1 + 1.0
2.0
>>> type(2.0)
<class 'float'>
```

NUMBERS IN A BOOLEAN CONTEXT:

You can use numbers in a Boolean context, such as an **if statement**. Zero values are false, and non-zero values are true.

STRINGS

From a functional perspective, strings can be used to represent just about anything that can be encoded as text or bytes.

In the text department, this includes **symbols and words** (e.g., your name), contents of text files loaded into memory, Internet addresses, Python source code, and so on.

Strings can also be used to hold the **raw bytes** used for media files and network transfers, and both the encoded and decoded forms of non-ASCII Unicode text used in internationalized programs. Strings are used to record both textual information (your name, for instance) as well as arbitrary collections of bytes.

Python 3.X comes with three string object types—one for textual data and two for binary data:

• **str** for representing decoded Unicode text (including ASCII)
• **bytes** for representing binary data (including encoded text)
• **bytearray**, a mutable flavor of the bytes type

STRINGS IN A BOOLEAN CONTEXT:

You can use strings in a Boolean context, such as an ‘if’ statement. Empty strings are false, and non-zero strings are true.

PYTHON CONVENTIONS

LISTS

Lists are Python's most used datatype. Creating a list is easy: use square brackets to wrap a comma-separated list of values.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[0]
'a'
>>> a_list[4]
'example'
>>> a_list[-1]
'example'
```

LISTS IN A BOOLEAN CONTEXT:

You can also use a list in a boolean context, such as an 'if' statement.
1. In a boolean context, an empty list is false.
2. Any list with at least one item is true.

TUPLES

A tuple is an immutable list. A tuple cannot be changed in any way once it is created.

TUPLES IN A BOOLEAN CONTEXT:

You can use tuples in a Boolean context, such as an if statement.
1. In a boolean context, an empty tuple is false.
2. Any tuple with at least one item is true.

SETS

A set is an unordered "container" of unique values. A single set can contain values of any immutable datatype. Once you have two sets, you can do standard set operations like union, intersection, and set difference.

SETS IN A BOOLEAN CONTEXT: You can use sets in a Boolean context, such as an if statement.
1. In a boolean context, an empty set is false.
2. Any set with at least one item is true.

DICTIONARIES

A dictionary is an unordered container of key-value pairs.

DICTIONARIES IN A BOOLEAN CONTEXT: You can also use a dictionary in a Boolean context, such as an if statement.
1. In a Boolean context, an empty dictionary is false.
2. Any dictionary with at least one key-value pair is true.

FILES

File objects are Python code's main interface to external files on your computer. They can be used to read and write text memos, audio clips, Excel documents, saved email messages, and whatever else you happen to have stored on your machine. Files

Other types in Python either are objects related to program execution (like functions, modules, classes, Exceptions and compiled code). There are many othe objects that we will see as we go along.

PYTHON OPERATORS

Operators are the constructs, which can manipulate the value of operands.

Types of Operator

Python language supports the following types of operators-

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

Python Arithmetic Operators

Assume variable **a holds the value 10** and variable **b holds the value 21**, then-

+	Addition Adds values on either side of the operator.	a + b = 31
-	Subtraction Subtracts right hand operand from left hand operand.	a – b = -11
*****	Multiplication Multiplies values on either side of the operator	a * b = 210
/	Division Divides left hand operand by right hand operand	b / a = 2.1
%	Modulus Divides left hand operand by right hand operand and returns remainder	b % a = 1
******	Exponent Performs exponential (power) calculation on operators	a**b =10 to the power 21
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Example

```
a = 21
b = 10
c = 0

c = a + b
print ("Line 1 - Value of c is ", c)
c = a - b
print ("Line 2 - Value of c is ", c )
c = a * b
print ("Line 3 - Value of c is ", c)
c = a / b
print ("Line 4 - Value of c is ", c )
c = a % b
print ("Line 5 - Value of c is ", c)

a = 2
b = 3
c = a**b

print ("Line 6 - Value of c is ", c)

a = 10
b = 5
c = a//b

print ("Line 7 - Value of c is ", c)
```

When you execute the above program, it produces the following result-

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2.1
Line 5 - Value of c is 1
Line 6 - Value of c is 8
Line 7 - Value of c is 2
```

Python Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators. Assume variable **a holds the value 10** and variable **b holds the value 20**, then-

==	If the values of two operands are equal, then the condition becomes true.	(a == b)
!=	If values of two operands are not equal, then condition becomes true.	(a!= b)
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b)
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b)
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true	(a >= b)
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b)

Example

```
>>> a=20
>>> b=10
>>> a<b
False
>>> b<a
True
>>> c='xyz'
>>> d='xxz'
>>> c<d
False
>>> d<c
True
>>>
```

Python Assignment Operators

Assume variable **a holds 10** and variable **b holds 20**, then-

=	Assigns values from right side operands to left side operand	c = a + b
+=	Add AND It adds right operand to the left operand and assign the result to left operand. Is equivalent to c = c + a	c += a
-=	Subtract AND It subtracts right operand from the left operand and assign the result to left operand	c -= a
*=	Multiply AND It multiplies right operand with the left operand and assign the result to left operand	c *= a
/=	Divide AND It divides left operand with the right operand and assign the result to left operand	c /= a
%=	Modulus AND It takes modulus using two operands and assign the result to left operand	c %= a
**=	Exponent AND Performs exponential (power) calculation on operators and assign value to the left operand	c **= a
//=	Floor Division It performs floor division on operators and assign value to the left operand	c //= a

Example

Assume variable a holds 10 and variable b holds 20, then-

```
a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c)
c += a
print ("Line 2 - Value of c is ", c )
c *= a
print ("Line 3 - Value of c is ", c )
c /= a
print ("Line 4 - Value of c is ", c )
c = 2
c %= a
print ("Line 5 - Value of c is ", c)
c **= a
print ("Line 6 - Value of c is ", c)
c //= a
print ("Line 7 - Value of c is ", c)
```

When you execute the above program, it produces the following result-

```
Line 1 - Value of c is 31
Line 2 - Value of c is 52
Line 3 - Value of c is 1092
Line 4 - Value of c is 52.0
Line 5 - Value of c is 2
Line 6 - Value of c is 2097152
Line 7 - Value of c is 99864
```

Python Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if **a = 60**; and **b = 13**; Now in binary format they will be as follows: a = 0011 1100 b = 0000 1101	a&b = 0000 1100 a|b = 0011 1101 a^b = 0011 0001 ~a = 1100 0011

&	Binary AND Operator copies a bit to the result, if it exists in both operands	(a & b) (means 0000 1100)
|	Binary OR It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^	Binary XOR It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~	Binary Ones Complement. It is unary and has the effect of 'flipping' bits	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number).
<<	Binary Left Shift The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>>	Binary Right Shift The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Example

```
a = 60                                          # 60 = 0011 1100
b = 13                                          # 13 = 0000 1101
print ('a=',a,':',bin(a),'b=',b,':',bin(b))
c = 0
c = a & b;                                      # 12 = 0000 1100
print ("result of AND is ", c,':',bin(c))
c = a | b;                                      # 61 = 0011 1101
print ("result of OR is ", c,':',bin(c))
c = a ^ b;                                      # 49 = 0011 0001
print ("result of EXOR is ", c,':',bin(c))
c = ~a;                                          # -61 = 1100 0011
print ("result of COMPLEMENT is ", c,':',bin(c))
c = a << 2;                                          # 240 = 1111 0000
print ("result of LEFT SHIFT is ", c,':',bin(c))
c = a >> 2;                                          # 15 = 0000 1111
print ("result of RIGHT SHIFT is ", c,':',bin(c))
```

When you execute the above program, it produces the following result

```
a=60 : 0b111100 b= 13 : 0b1101
result of AND is 12 : 0b1100
result of OR is 61 : 0b111101
result of EXOR is 49 : 0b110001
result of COMPLEMENT is -61 : -0b111101
result of LEFT SHIFT is 240 : 0b11110000
result of RIGHT SHIFT is 15 : 0b111
```

Python Logical Operators

The following logical operators are supported by Python language.
Assume variable **a holds True** and variable **b holds False** then-

and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true	(a or b) is True
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below-

in	Evaluates to true, if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true, if it does not find a variable in the specified sequence and false otherwise	x not in y, here not in results in a 1 if x is not a member of sequence y

Example

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]
if ( a in list ):
	print ("Line 1 - a is available in the given list")
else:
	print ("Line 1 - a is not available in the given list")
if ( b not in list ):
	print ("Line 2 - b is not available in the given list")
else:
	print ("Line 2 - b is available in the given list")
c=b/a
if ( c in list ):
	print ("Line 3 - a is available in the given list")
else:
	print ("Line 3 - a is not available in the given list")
```

When you execute the above program, it produces the following result-

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below:

is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Example

```
a = 20
b = 20
print ('Line 1','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is b ):
        print ("Line 2 - a and b have same identity")
else:
        print ("Line 2 - a and b do not have same identity")

if ( id(a) == id(b) ):
        print ("Line 3 - a and b have same identity")
else:
        print ("Line 3 - a and b do not have same identity")

b = 30
print ('Line 4','a=',a,':',id(a), 'b=',b,':',id(b))

if ( a is not b ):
        print ("Line 5 - a and b do not have same identity")
else:
        print ("Line 5 - a and b have same identity")
```

When you execute the above program, it produces the following result-

```
Line 1 a= 20 : 1594701888 b= 20 : 1594701888
Line 2 - a and b have same identity
Line 3 - a and b have same identity
Line 4 a= 20 : 1594701888 b= 30 : 1594702048
Line 5 - a and b do not have same identity
```

Python Operators Precedence

**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not or and	Logical operators

Operator precedence affects the evaluation of an expression. For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because the operator * has higher precedence than +, so it first multiplies 3*2 and then is added to 7.

Example

```
a = 20
b = 10
c = 15
d = 5
print ("a:%d b:%d c:%d d:%d" % (a,b,c,d ))
e = (a + b) * c / d #( 30 * 15 ) / 5
print ("Value of (a + b) * c / d is ", e)
e = ((a + b) * c) / d # (30 * 15 ) / 5
print ("Value of ((a + b) * c) / d is ", e)
e = (a + b) * (c / d) # (30) * (15/5)
print ("Value of (a + b) * (c / d) is ", e)
e = a + (b * c) / d # 20 + (150/5)
print ("Value of a + (b * c) / d is ", e)
```

When you execute the above program, it produces the following result

```
a:20 b:10 c:15 d:5
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

The **print() function** prints the given object to the standard output device (screen) or to the text stream file.

The full syntax of print() is:

```
>>>print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

print() Parameters

objects - object to the printed. ***** indicates that there may be more than one object

sep - objects are separated by sep. **Default value**: ' '

end - end is printed at last

file - must be an object with write(string) method. If omitted it, sys.stdout will be used which prints objects on the screen.

flush - If True, the stream is forcibly flushed. **Default value**: False

> **Note:** sep, end, file and flush are keyword arguments. If you want to use sep argument, you have to use:
>
> print(*objects, sep = 'separator')
> not
> print(*objects, 'separator')

Return Value from print()
It doesn't return any value; returns None.

EXAMPLE:

```
>>> a = 5

>>>print("a =", a, sep='00000', end='\n\n\n')

>>>print("a =", a, sep='0', end='')
```

Python input()

The input() method reads a line from input, converts into a string and returns it.

The syntax of input() method is:

```
>>>input([prompt])
```

input() Parameters

The input() method takes a single optional argument:

- **prompt (Optional) -** a string that is written to standard output (usually screen) without trailing newline

Return value from input()

The input() method reads a line from input (usually user), converts the line into a string by removing the trailing newline, and returns it.

EXAMPLE

```
>>># get input from user
>>> x=input('Enter a number')
>>>x=int(x)
>>>inputString = input('Enter a string:')
```

EXERCISES:

1. Ask the user to enter their name and print a greeting
2. Ask the user to enter a number and print it.
3. Ask the user to enter 2 numbers and do math with:
   a. Add
   b. Subtract
   c. Multiply
   d. Divide
   e. Floor divide
   f. Modulus
   g. Power

4. Ask the user to enter their address (multiple lines) and print it.
5. Ask the user to enter a number 5 times
   a. Find the average of the numbers
   b. Print the numbers on the same line
   c. Change the separator to comma
   d. Print the sum on a different line
   e. Print the average on a different line