Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #17 and 18
Operators and Expressions

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Verified by,
PCPS Team - 2023

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Operators and expression

Operator and Operand: A **symbol that denotes the operation** to be performed on the data in **the process of computational problem solving** is known as an operator. The **data used with operators** is said to be **operands.**

Expression: A **valid combination of operators and operands** which evaluates to a value is called as **expression**. Examples of expression are mentioned here.

- Constants / Literals

  1234.67, “python”, -23

- Variables

  a = 45 # **a is an expression but a= 45 is not an expression but a statement**

- Any Operations on literals and variables

  5+6, 2/2, a > 2 & b <= 5

Few points to note:

- An expression has a value. Statement does not have a value.
- An expression is also a statement. A statement is not necessarily an expression.
- **Assignment is not an expression. a1 = 5 is just a statement**

Categories of Operators:

- **Based on the number of operands, the operator takes**

  Unary, Binary, Ternary

- **Based on the operation**

  Arithmetic, Logical, Relational, Membership, Identity, Bitwise, Assignment, Shorthand etc.

Arity / Rank: Refers to the number of operands required for the operator.

- **Unary – 1 operand**

  Example expressions: +5, -2, ~a, not 5, (-a)

- **Binary - 2 operands**

  Example expressions: 7+5, 1-2, a^b, a>b, a>10 or b<20, (s//3)

- **Ternary - 3 operands [Will be discussed later]**

  if else

Arithmetic Operators: Addition(+), Subtraction(-), Multiplication(*), Division(/), Truncation Division(//), Remainder(%), Exponentiation(**)

For testing some of the operators and expressions, **use the interpreter mode of python prompt as below.**

```
>>> 25 / 4    #True Division
6.25
>>> 25 // 4    #Truncation division or Integer Division. 25 is the dividend, 4 is the divisor, perform division and integer quotient is the result.
6
>>> 25 % 4    #25 is the dividend, 4 is the divisor, result is the remainder after performing integer division
1
>>> 25 ** 3     #25 power 3
15625
```

Note:

 # is the Single line comment. Comments are ignored by the compiler/interpreter.

‘’’ and ‘’’ or “”” and “”” are used as multiline comments. They are multiline strings in actual.

Consider the below statements. And see what is happening here?

```
print(25 / 4, 25 % 4, 25 // 4, 27 // 4, -25 // 4, -27 // 4)
#      6.25    1      6       6       -7       -7
print(25.8 % 4.2) # modulus operator works on float values in python
# 0.6
```

While using **truncation division(//), note these points**:

- When **both operands are integer values, the result is a truncated integer** referred to as integer division. Example: 40//10 evaluates to 4
- When **at least one of the operands is a float type, the result is a truncated floating point**. Example: 44.0//10 evaluates to 4.0

Relational Operators: <, <=, >, >=, ==, !=

These are used to compare two values. The **result is of bool type** with values True and False. For testing some of the operators and expressions, use the interpreter mode of python prompt as below.

```
>>> 3 > 1
True
>>> 3 >= 3
True
>>> 3 > 3
False
>>> 2.7 < 10.4
True
>>> 2 == 2
True
>>>2 != 2
False
>>> print( 3>=3.0<2.5)
False
```

Consider the below statements. And see what is happening here?

```
print(10 == 10)   #True
print(3 > 2)   #True
print(3 > 2 > 1)   #True . This is known as cascading of operations.
```

Cascading operations

```
print(3 > 2 > 1)      #Same as 3>2 and 2> 1
print(10 == 10 == 10)      #Same as 10 == 10 and 10 == 10
print(3>2<1)      #False
```

Think about how these operators work with other collections like Lists, strings etc. ?

Logical Operators: and, or, not

This is used to construct arbitrarily complex **Boolean expressions**. Hence, the other name for Logical operators is Boolean operators.

x	y		x and y	x or y	not x
False	False		False	False	True
True	False		False	True	False
False	True		False	True	
True	True		True	True	

These operators work on **Boolean values and Boolean equivalent values**

- **False Values**:
  False. 0 , '' (Empty String), [] , {} , () (Empty Collections)

- **True Values:**
  True**,** non – Zero numbers, Non Empty String, Non Empty Collections

These operators perform **short circuit evaluation or lazy evaluation** -Evaluate a logical expression left to right and stop the evaluation as soon as the truth or the falsehood is found.

Logical **and:** If the first operand evaluates to false, then regardless of the value of the second operand, the expression is false

Logical **or**: If the first operand evaluates to true, regardless of the value of the second operand, the expression is true.

Note: Python interpreter does not evaluate the second operand when the result is known by the first operand alone

```
>>> 2 and 5
5
>>> 0 and 5
0
>>> 3 and 0.0
0.0
>>> 3 and "pes"
'pes'
>>> 3 and ""
''
```

```
>>> 0.0 or "pes"
'pes'
>>> "pes" or ""
'pes'
>>> 5.2 or 5
5.2
>>> not 0
True
>>> not 1.3
False
```

```
>>> a=3
>>> b=-4
>>> a>b and b<4
True
>>> a>b and b>4
False
>>> a>b or b>a
True
>>> "" and 0.0
''
>>> 3 or 5
3
>>> 3 or 0
3
>>> 0 or 7
7
```

```
>>> not 1
False
>>> not "pes"
False
>>> not ""
True
>>> not 0.000
True
>>> a =1
>>> a and True
True
>>> 0 and True
0
>>> True and 0
0
>>> 0 or True
True
```

Bitwise Operators: &, |, ^, >>, <<, ~

These operators **operate on the binary value** of the given operand. They operate on a **bit-level,** which means that **each binary digit** is considered**.**

Operator	Name	Result
&	AND	result is 1 if the corresponding bits are one
|	OR	result is 1 if even at least one of the bits is one
^	Exclusive OR	result is 1 if and only if one of the bits is 1
<<	LEFT SHIFT	multiply by 2 for each left shift
>>	RIGHT SHIFT	divide by 2 for each right shift
~	ONE'S COMPLIMENT	change 0 to 1 and 1 to 0

& Operator

```
a = 5          # 0101
b = 6          # 0110
c = a & b      # 0100  (4)
```

| Operator

```
a = 5          # 0101
b = 6          # 0110
c = a | b      # 0111  (7)
```

^ Operator

```
a = 5          # 0101
b = 6          # 0110
c = a ^ b      # 0011  (3)
```

~ One's complement

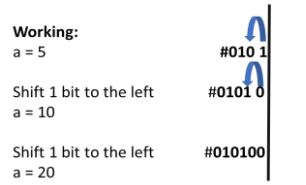
```
a = 5
print ( "~ a ", ~a )
# 1111111111010 => -6
```

A general rule of thumb **is**

~x = -(x+1)

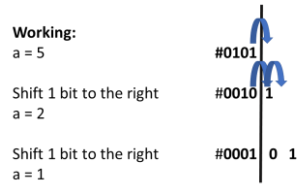
<< Left Shift Operator

```
a = 5          # 0101
b = a << 2     # 010100
```



>> Right Shift Operator

```
a = 5          # 0101
b = a >> 2     # 0001
```



```
>>> 4 & 8
0
>>> 4 | 8
12
>>> 4 ^ 8
12
>>> 10 >> 3
1
>>> 10 / (2**3)
1.25
>>> 10 // (2**3)
1
>>> 10 << 3
80
>>> 10 * (2**3)
80
>>> ~5
-6
>>> ~-6
5
```

Membership Operators: in, not in

These operators can be used to determine if a particular value occurs within a specified collection of values.

```
>>> 11 in [23,44,"pai",11,0,"sindhu"]
True
>>> '11' in [23,44,"pai",11,0,"sindhu"]
False
>>> "s" in "sindhu"
True
>>> print("app" in "whatsapp")   #True
True
>>> print("pap" in "whatsapp")  #checks for the string "pap" in "whatsapp". Not the character p and a and p
False
>>> print("pap" not in "watsapp")
True
```

Identity Operators: is, is not

Checks if the operands on either side of the operator point to the same object ornot. It **compares the ids of both the operands.**

```
>>> 6 is 6
<stdin>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
True
>>> 6 is not 6
<stdin>:1: SyntaxWarning: "is not" with a literal. Did you mean "!="?
False
>>> 6 is not 7
<stdin>:1: SyntaxWarning: "is not" with a literal. Did you mean "!="?
True
```

```
>>> a = 6
>>> b = 6
>>> a is b
True
>>> a is not b
False
>>> id(a) is id(b)   #DO not worry much about the output as the id function works differently in interpreter mode and batch mode. Kindly do not break your head!!! Just to wake you up, this expression has been added.
False
```

Assignment Operator: =

It assigns the value of the right side expression to a left side variable. Using = in an expression doesn't evaluate to a value. Hence this is considered as a **statement in python** rather than an expression.

```
>>> a = 5
>>> a
5
>>> a = 6//3
>>> a
2
>>> print(a)
2
>>>b = 6
>>> a = b+a
8
```

```
>>> print(a = 3)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: 'a' is an invalid keyword argument for print()
>>> if(a = 3):
File "<stdin>", line 1
        if(a = 3):
        ^^^^^
        SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Shorthand Operators: += , -=, *= , /= , //= , %= , **=

It combines arithmetic and assignment operators.

Expression	Short Hand
a = a + b	a += b
a = a - b	a -= b
a = a * b	a *= b
a = a / b	a /= b
a = a // b	a //= b
a = a % b	a %= b
a = a ** b	a **= b

```
>>> a = 5
>>> a += 10
>>> a
15
>>> b = 7
>>> a -= b
>>> a
8
>>> a **= 2
>>> a
64
```

Operator polymorphism:

A type of polymorphism where **an operator behaves differently based on the type of the operands.** Polymorphism means many forms.

Consider + operator:

- Acts as **Addition operator** if the operands are numbers.
- Acts as **Concatenation operator** if the operands are strings.

```
>>> 2 + 45
47
>>> "2" + "45"
'245'
>>> 2 + "45"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Consider * operator

- Works as **Multiplication operator** if the operands are numbers.
- Works as **Repetition operator** if one operand is a string and other operand a Non-Negative integer

```
>>> 2 * 8
16
>>> 2 *"8"
'88'
>>> "2" * 8
'22222222'
>>> "2" * "8"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
>>> -2 * "8"
''
>>> "sindhu" * 2
'sindhusindhu'
>>> "sindhu" * -3
''
>>> ■
```

Consider this: -> Boolean literal True represents value 1 and False represents value 0

```
>>> True + "1"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'bool' and 'str'
>>> True + 1
2
>>> True * 1
1
>>> True * 3
3
>>> 4 + 5.0
9.0
>>> '4' + '5.0'
'45.0'
>>> "0" * False
''
>>> "0" * 0
''
>>> "0" * 2
'00'
>>>
```

Point to Think: If more than one operator is there in an expression, how the evaluation happens? Refer to Lecture #19