Introduction to Computing

Basics of Python Programming – II

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

September, 2024

Arithmetic operators

Summation $(+)$

Subtraction $(-)$

Multiplication $(*)$

Power $(**)$

Float Division $(/)$

Floor Division $(//)$

Modulo division $(\%)$

Arithmetic operators

Summation $(+)$

Subtraction $(-)$

Multiplication $(*)$

Power $(**)$

Float Division $(/)$

Floor Division $(//)$

Modulo division $(\%)$

Note: Modulo division operator works on any type of numbers (including floating point values and negatives!!!) and returns the sign of the denominator (with a different kind of computation).

Modular division on negative values

What will be the output of the following program?

```
print(4 / 3, 4 // 3, 4 % 3)
print(-4 / 3, -4 // 3, -4 % 3)
print(4 / -3, 4 // -3, 4 % -3)
print(-4 / -3, -4 // -3, -4 % -3)
```

Modular division on negative values

What will be the output of the following program?

```
print(4 / 3, 4 // 3, 4 % 3)
print(-4 / 3, -4 // 3, -4 % 3)
print(4 / -3, 4 // -3, 4 % -3)
print(-4 / -3, -4 // -3, -4 % -3)
```

Output:

```
1.3333333333333333 1 1
-1.3333333333333333 -2 2
-1.3333333333333333 -2 -2
1.3333333333333333 1 -1
```

Modulo division on floating point values

What will be the output of the following program?

```
print(4.3 % 3)
print(4 % 3.5)
print(4.3 % 3.5)
print(4 % 3)
```

Modulo division on floating point values

What will be the output of the following program?

```
print(4.3 % 3)
print(4 % 3.5)
print(4.3 % 3.5)
print(4 % 3)
```

Output:

```
1.2999999999999998
0.5
0.7999999999999998
1
```

Note: $x\%y$ returns r (following IEEE 754, the IEEE Standard for floating-point arithmetic) if and only if $x = n * y + r$, where n is an integer, r has the same sign as y, and $|r| < |y|$.

Modulo division on Boolean values

What will be the output of the following program?

```
var1 = True
var2 = False
print(var1 % var1)
print(var1 % var2)
print(var2 % var1)
print(var2 % var2)
```

Modulo division on Boolean values

What will be the output of the following program?

```
var1 = True
var2 = False
print(var1 % var1)
print(var1 % var2)
print(var2 % var1)
print(var2 % var2)
```

Output:

0

Error

0

Error

Relational operators

Less than $(<)$

Less than equals to $(<=)$

Greater than $(>)$

Greater than equals to $(>=)$

Equals to $(==)$

Not equals to $(!=)$

Logical operators

Logical and (`and`)

Logical or (`or`)

Logical not (`not`)

Assignment operators

Assignment $(=)$

Addition and assignment $(+=)$

Subtraction and assignment $(-=)$

Multiplication and assignment $(*=)$

Power and assignment $(**=)$

Float Division and assignment $(/=)$

Floor Division and assignment $(//=)$

Modulo division and assignment $(\%=)$

Assignment operators

What will be the output of the following program?

```
a = 10
b = 20
print(a, b)
print(hex(id(a)), hex(id(b)))
a, b = b, a # Swapping values
print(a, b)
print(hex(id(a)), hex(id(b)))
```

Assignment operators

What will be the output of the following program?

```
a = 10
b = 20
print(a, b)
print(hex(id(a)), hex(id(b)))
a, b = b, a # Swapping values
print(a, b)
print(hex(id(a)), hex(id(b)))
```

Output:

10 20
0x72fca540 0x72fca8e0
20 10
0x72fca8e0 0x72fca540 # Only the pointers will change!!!

Bitwise operators

Bitwise and $(\&)$

Bitwise or $(|)$

Bitwise not $(\sim)$

Bitwise xor $(\wedge)$

Left shift $(<<)$

Right shift $(>>)$

Bitwise operators

```
i = 11
print(i>>1) # 1 place right shift (i unchanged)
print(i<<2) # 2 places left shift (i unchanged)
```

Bitwise operators

```
i = 11
print(i>>1) # 1 place right shift (i unchanged)
print(i<<2) # 2 places left shift (i unchanged)
```

Output:

5

44

Bitwise operators

```
i = 11
print(i>>1) # 1 place right shift (i unchanged)
print(i<<2) # 2 places left shift (i unchanged)
```

Output:

5

44

Decimal	Binary							
11	0	0	0	0	1	0	1	1
5	0	0	0	0	0	1	0	1
44	0	0	1	0	1	1	0	0

Efficiency of bitwise operators

Verifying whether a number is a power of 2 with `bitwise AND`:

```
n = 32
if n & n-1:
    print(n, "is not a power of 2")
else:
    print(n, "is a power of 2")
```

Efficiency of bitwise operators

Verifying whether a number is a power of 2 with `bitwise AND`:

```
n = 32
if n & n-1:
    print(n, "is not a power of 2")
else:
    print(n, "is a power of 2")
```

Output:

```
32 is a power of 2
```

Efficiency of bitwise operators

Verifying similarity of signs with `bitwise XOR`:

```
m = 10
n = -20
if m ^ n < 0:
    print(m, " and ", n, " have different signs");
else:
    print(m, " and ", n, " have the same signs");
```

Efficiency of bitwise operators

Verifying similarity of signs with `bitwise XOR`:

```
m = 10
n = -20
if m ^ n < 0:
    print(m, " and ", n, " have different signs");
else:
    print(m, " and ", n, " have the same signs");
```

Output:

```
10  and -20  have different signs
```

Identity operators

Identical (`is`)

Not identical (`is not`)

Identity operators

Identical (`is`)

Not identical (`is not`)

Note: Two variables that are equal does not imply that they are identical. For being identical, they must be located on the same part of the memory.

Identity operators

What will be the output of the following program?

```
var1 = 123
var2 = 123
print(var1 is var2)
var1 = 'Python'
var2 = "Python"
print(var1 is not var2)
var1 = [1, 2, 3]
var2 = [1, 2, 3]
print(var1 is var2)
```

Identity operators

What will be the output of the following program?

```
var1 = 123
var2 = 123
print(var1 is var2)
var1 = 'Python'
var2 = "Python"
print(var1 is not var2)
var1 = [1, 2, 3]
var2 = [1, 2, 3]
print(var1 is var2)
```

Output:

```
True
False
False
```

Identity operators

What will be the output of the following program?

```
var1 = 123
var2 = 123
print(hex(id(var1)), hex(id(var2)))
var1 = 'Python'
var2 = "Python"
print(hex(id(var1)), hex(id(var2)))
var1 = [1, 2, 3]
var2 = [1, 2, 3]
print(hex(id(var1)), hex(id(var2)))
```

Identity operators

What will be the output of the following program?

```
var1 = 123
var2 = 123
print(hex(id(var1)), hex(id(var2)))
var1 = 'Python'
var2 = "Python"
print(hex(id(var1)), hex(id(var2)))
var1 = [1, 2, 3]
var2 = [1, 2, 3]
print(hex(id(var1)), hex(id(var2)))
```

Output:

```
0x812ee0 0x812ee0
0x9227e8 0x9227e8
0x27f9430 0x31b8740
```

Some comments

The following popular operators are not available in Python:

- **Increment** (++)
- **Decrement** (- -)
- **Comma** (,)

Use of comma

What will be the output of the following program?

```
var = 20, 30, 40
print(var)
var = (50, 60, 70)
print(var)
```

Use of comma

What will be the output of the following program?

```
var = 20, 30, 40
print(var)
var = (50, 60, 70)
print(var)
```

Output:

```
(20, 30, 40)
(50, 60, 70)
```

Operator precedence (highest to lowest)

Associativity	Operator	Description
Left-to-right	()	Parentheses (grouping)
Left-to-right	*f*(args...)	Function call
Left-to-right	*x*[index:index]	Slicing
Left-to-right	*x*[index]	Array Subscription
Right-to-left	**	Exponentiation
Left-to-right	~*x*	Bitwise not
Left-to-right	+*x* -*x*	Positive, Negative
Left-to-right	* / %	Multiplication Division Modulo
Left-to-right	+ -	Addition Subtraction
Left-to-right	<< >>	Bitwise left shift Bitwise right shift
Left-to-right	&	Bitwise AND
Left-to-right	^	Bitwise XOR
Left-to-right	|	Bitwise OR
Left-to-right	in, not in, is, is not, <, <=, >, >=, <>, == !=	Membership Relational Equality Inequality
Left-to-right	not *x*	Boolean NOT
Left-to-right	and	Boolean AND
Left-to-right	or	Boolean OR
Left-to-right	lambda	Lambda expression

Conditional – if-else

```
if <Condition>:
    statement 1
    statement 2
else:
    statement 3 # Execute if Condition fails
```

Note: Boundary of the conditional block is demarcated by indentation.

Iterative – if-elif-else

```
if <Condition 1>:
    statement 1
elif <Condition 2>:
    Statement 2
else:
    statement 3 # Execute if Condition 1 and 2 fails
```

Note: Boundary of the conditional block is demarcated by indentation.

Iterative – `for` loop

```
for <variable> in <container>:
    statement 1
    statement 2
```

Note: Boundary of the iterative block is demarcated by indentation.

Iterative – `for` loop

We can create a list of consecutive integers using the `range()` function as follows.

```
for <variable> in range(<value>):
    statement 1
    statement 2
```

- `range(x)` returns a list whose items are consecutive integers from [0, x).
- `range(x, y)` returns a list (feasible when x $<$ y) whose items are consecutive integers from [x, y).
- `range(x, y, step)` returns a list of integers from [x, y), such that the difference between each two adjacent items in the list is step. If step is less than 0, it counts down from x to y. If step equals 0, it raises an exception.

Iterative – `for` loop

The loop variable within the `for` is optional. You can skip mentioning a loop variable if it is not to be used iteratively.

```
for i in range(3):
    print(i)
for _ in range(3):
    print('Hi')
```

Output:

```
0
1
2
Hi
Hi
Hi
```

Efficient `for` loop

```
list = []
for i in range(6):
    list.append(i*2)
print(list)
```

Efficient `for` loop

```
list = []
for i in range(6):
    list.append(i*2)
print(list)
```

Output: [0, 2, 4, 6, 8, 10]

Efficient `for` loop

```
list = []
for i in range(6):
    list.append(i*2)
print(list)
```

Output: [0, 2, 4, 6, 8, 10]

```
list = [i*2 for i in range(6)]
print(list)
```

Efficient `for` loop

```
list = []
for i in range(6):
    list.append(i*2)
print(list)
```

Output: [0, 2, 4, 6, 8, 10]

```
list = [i*2 for i in range(6)]
print(list)
```

Output: [0, 2, 4, 6, 8, 10]

Iterative – `while` loop

```
while <Condition>:
    statement 1
    statement 2
```

Note: Boundary of the iterative block is demarcated by indentation.

`break` and `continue`

- **break:** Immediately jump to the next operation after the loop
- **continue:** Do the operation, if applicable, and continue with the next iteration of the loop

break and continue

- **break:** Immediately jump to the next operation after the loop
- **continue:** Do the operation, if applicable, and continue with the next iteration of the loop

```
for i in range(1, 100):
    print(i)
    if i%10 != 0:
        break
```

Prints only 1

break and continue

- **break:** Immediately jump to the next operation after the loop
- **continue:** Do the operation, if applicable, and continue with the next iteration of the loop

```
for i in range(1, 100):
    print(i)
    if i%10 != 0:
        break
```

Prints only 1

```
i = 0
while i < 100:
    i = i+1
    if i%10 != 0:
        continue
    print(i)
```

Prints 10, 20, ..., 100

Let's try solving a problem

Given a positive integer n as user input, find out the number of trailing zeros in $n!$.

Let's try solving a problem

Given a positive integer n as user input, find out the number of trailing zeros in $n!$.

Hint: This can be done with $\log_5 n$ number of divisions.

Let's try solving a problem

Given a positive integer n as user input, find out the number of trailing zeros in $n!$.

Hint: This can be done with $\log_5 n$ number of divisions.

```
n = input("Enter n: ")
n = int(n)
count = 0
while n:
    n //= 5
    count += n
print('Number of trailing zeros:', count)
```

Standard Input/Output functions

I/O from the terminal:

Value can be printed without mentioning the type as:

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

Value is taken in a string and can be converted to appropriate type using int(), float(), bool(), etc. as:

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

I/O from files:

- `open()`, `close()` # Files are opened in r/w/a mode and the address is returned to a file pointer
- `read()`, `write()` # With a file pointer
- `readline()` # With a file pointer
- `readlines()`, `writelines()` # With a file pointer

print() – Special features

```
a, b, c = 1, 2, 3
print(a, b, c, sep = '')
print(a, b, c, sep = '-')
print(a, b, c, sep = '1.1')
x = print('Hi')
print(x)
print(type(x))
```

print() – Special features

```
a, b, c = 1, 2, 3
print(a, b, c, sep = '')
print(a, b, c, sep = '-')
print(a, b, c, sep = '1.1')
x = print('Hi')
print(x)
print(type(x))
```

Output:

```
123
1-2-3
11.121.13
Hi
None
<class 'NoneType'>
```

print() – Special features

```
a = 7
print(a, 'is prime', end = ';') # ',' includes a space
b = 'prime'
print('7 is ' + b, end = ';') # '+' works only on strings
```

print() – Special features

```
a = 7
print(a, 'is prime', end = ';') # ',' includes a space
b = 'prime'
print('7 is ' + b, end = ';') # '+' works only on strings
```

Output:

```
7 is prime;7 is prime;
```

print() – Special features

```
ls = [[1, 2, 3], [4, 5, 6]]
for i in range(len(ls)):
    for j in range(len(ls[0])):
        print(ls[i][j], end = ' ')
    print()
```

print() – Special features

```
ls = [[1, 2, 3], [4, 5, 6]]
for i in range(len(ls)):
    for j in range(len(ls[0])):
        print(ls[i][j], end = ' ')
    print()
```

Output:

```
1 2 3
4 5 6
```

print() – Special features

```
inputFile = open('test.txt', 'w')
print('Write your own fate!!!', file = inputFile)
inputFile.close()
```

print() – Special features

```
inputFile = open('test.txt', 'w')
print('Write your own fate!!!', file = inputFile)
inputFile.close()
```

Output:

```
Write your own fate!!!
```

- written within test.txt.

input() – Special features

```
x = int(input())
print(x)
n = input('Enter three integers: ')
print(n, list(n))
n1, n2, n3 = input('Enter three integers: ').split()
print(n1+n2+n3, int(n1)+int(n2)+int(n3))
```

input() – Special features

```
x = int(input())
print(x)
n = input('Enter three integers: ')
print(n, list(n))
n1, n2, n3 = input('Enter three integers: ').split()
print(n1+n2+n3, int(n1)+int(n2)+int(n3))
```

Output:

```
10
10
Enter three integers: 1 2 3
1 2 3 ['1', ' ', '2', ' ', '3']
Enter three integers: 1 2 3
123 6
```

input() – Special features

```
r = int(input('Enter the number of rows: '))
c = int(input('Enter the number of columns: '))
MAT = [[int(input()) for i in range(c)] for j in range(r)]
print(MAT)
```

input() – Special features

```
r = int(input('Enter the number of rows: '))
c = int(input('Enter the number of columns: '))
MAT = [[int(input()) for i in range(c)] for j in range(r)]
print(MAT)
```

Output:

```
Enter the number of rows: 2
Enter the number of columns: 3
1
2
3
4
5
6
[[1, 2, 3], [4, 5, 6]]
```

`input()` – Special features

Explore the `reshape()` function!!!

Reading data from file

```
def read(file):
    f = open(file, 'r')
    output = f.read()
    f.close()
    return output
output = read('Data.txt')
```

Reading data from file

```
def read(file):
    f = open(file, 'r')
    output = f.read()
    f.close()
    return output
output = read('Data.txt')
```

Reading data from file (alternative approach):

```
with open('Data.txt', 'r') as f: output = f.read();
```

Special Input/Output functions

Reading data from a CSV file:

```
import pandas as pd # Import pandas
pd.read_csv("file.csv") # reading CSV file
```

Special Input/Output functions

Reading data from a CSV file:

```
import pandas as pd # Import pandas
pd.read_csv("file.csv") # reading CSV file
```

Reading data from an XLS file:

```
import pandas as pd # Import pandas
pd.read_excel("file.xls") # supports old XLS file formats
pd.read_excel("file.xls", engine='openpyxl') # new formats
```

Special Input/Output functions

Reading data from a CSV file:

```
import pandas as pd # Import pandas
pd.read_csv("file.csv") # reading CSV file
```

Reading data from an XLS file:

```
import pandas as pd # Import pandas
pd.read_excel("file.xls") # supports old XLS file formats
pd.read_excel("file.xls", engine='openpyxl') # new formats
```

Note: The Python library `openpyxl` must be used to read/write Excel 2010 xlsx files.