UNIT 2
OPERATORS IN PYTHON

Operators

Operators are used to perform operations on variables and values.

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Operator	Symbols
Arithmetic	+ , - , * , **, / , //, %
Assignment	= , += , -= , *= , **= , /= , //=, %=
Relational	==, != , < , >, <= , >=
Logical	and , or , not
Identity	is , is not
membership	In , not in
Bitwise	& , | , ^ , ~ , << , >>

Arithmetic Operators:

Arithmetic operators are used to perform arithmetic operations on variables and data.

Ex: int a=4, b=2

Operator	Example	Output
+	a+b	6
-	a-b	2
*	a*b	8
/	a/b	2
%	a%b	0

```
print(7+2)
print(7-2)
print(7*2)
print(7**2)
print(7/2)
print(7//2)
```

```
9
5
14
49
3.5
3
```

- **Assignment Operators**

Assignment operators are used to assign values to variables

```
a,b=7,2
a+=b
print(a)
a-=b
print(a)
a*=b
print(a)
a**=b
print(a)
a//=b
print(a)
a/=b
print(a)
```

```
9
7
14
196
98
49.0
```

- **Comparison Operators**

Comparison operators are used to compare two values. And output of these are either True or False

```
print(7==2)    False
print(7!=2)    True
print(7>2)     True
print(7>2)     True
print(7>=2)    True
print(7<=2)    False
```

- **Logical Operators**

Logical operators are used to combine conditional statements

- ✓ and (Logical AND): returns true when both conditions are true.
- ✓ or (Logical OR): returns true if at least one condition is true.
- ✓ not (Logical NOT): returns true when a condition is false and vice-versa

```
print(7>2 and 2!=0)        True
print(7==2 or 2<3)         True
print(not(7==2 or 2<3))    False
```

- **Identity Operators**

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location

- ✓ is
- ✓ is not

```
a=[10,20,'aaa']
b=a
c=[10,20,'aaa']
print(a is b)        True
print(a is c)        False
print(a is not c)    True
```

- **Membership Operators**

Membership operators are used to test if a sequence is presented in an object

- ✓ In
- ✓ Not in

```
a=[10,20,'aaa',[10,20]]
b='aaa'
c=[10,20]
print(b in a)        True
print(b not in a)    False
print(c in a)        True
```

- **Bitwise Operators**

These operators are used to perform the manipulation of individual bits of a number

- ✓ &, Bitwise AND operator: returns bit by bit AND of input values.
- ✓ |, Bitwise OR operator: returns bit by bit OR of input values.
- ✓ ^, Bitwise XOR operator: returns bit-by-bit XOR of input values.
- ✓ ~, Bitwise Complement Operator: This is a unary operator which returns the one's

complement representation of the input value, i.e., with all bits inverted.

```
a,b=10,12        8
print(a & b)     14
print(a | b)     6
print(a ^ b)     -13
print(~b)        5
print(a>>1)      20
print(a<<1)
```

Working of Bitwise Complement Operator

Bitwise complement Operation of 2 (~ 0010): 1101

Calculate 2's complement of 3:

Binary form of 3 = 0011

1's Complement of 3 = 1100

Adding 1 to 1's complement = 1100 +1

2's complement of 3 = 1101

```
a=-2
b=2
print(~a)
print(~b)
```

```
1
-3
```

```
'''
  2 -> 0010
 ~2 -> 1101 (1s comp)--|
2s comp =1s + 1        |
2s of 3 ->0011         |
      1100 (1s comp)   |---same
        1              |
     -------           |
      1101  -----------|
'''
```

Note: The bitwise Complement of 2 is same as the binary representation of -3

Input

- This function first takes the input from the user and converts it into a string. The type of the returned object always will be <class 'str'>.
- When input() function executes program flow will be stopped until the user has given input.
- The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.
  Syntax: inp=input("Statement")

```
name=input("Enter Name")
print(name)
```

```
Enter Namesteve
steve
```

- Whatever you enter as input, the input function converts it into a string. if you enter an integer value still input() function converts it into a string. You need to explicitly convert it into an integer in your code using typecasting .

```
val=input("Enter value ")
print(val)
print(type(val))
```

```
Enter value 30
30
<class 'str'>
```

- Note: input() function takes all the input as a string only
- There are various function that are used to take as desired input few of them are : –
  - ✓ int(input())
  - ✓ float(input())

```
num=int(input("Enter Number "))
print(num)
print(type(num))
```

```
Enter Number 30
30
<class 'int'>
```

✓ **Taking multiple inputs using split()**

This function helps in getting multiple inputs from users. It breaks the given input by the specified separator. If a separator is not provided then any white space is a separator.

Syntax : var1, var2,…=input().split(separator, maxsplit)

```
x, y = input().split('@',2)
print(x,y)
print(type(x))
```

```
11@13
11 13
<class 'str'>
```

```
x, y = input().split()
print(x,y)
print(type(x))
```

```
10 20
10 20
<class 'str'>
```

- ✓ **Taking list as input:**

  In Python, taking a list as input can be done efficiently using the split() and map() functions together. The split() method breaks a single input string into multiple parts based on a delimiter (by default, a space). Then, the map() function applies a transformation (such as int() to convert each element to an integer) to all these parts. This method is commonly used to read multiple values from the user and store them as a list of a specific data type for further processing.

  Syntax: variable1, variable2, ... =list(map(datatype, input().split()))

Output

```
a,b=list(map(int,input().split()))
print(f"first num {a}, second num {b}")
```

```
10 20
first num 10, second num 20
```

print()

The print() function in Python is used to display output on the screen. It is one of the most commonly used functions in Python programming.

Syntax: `print(object(s), sep=' ', end='\n', file=sys.stdout, flush=False)`

Parameter	Description
object(s)	One or more values (strings, numbers, variables, etc.) to be printed.
sep	Optional. Separator between multiple values. Default is a space (' ').
end	Optional. What to print at the end. Default is newline ('\n').
file	Optional. Where to send the output (default is the screen).
flush	Optional. If True, forces the output to be printed immediately.

```
print("Python", "is", "skill", sep="-")
```

```
Python-is-skill
```

```
print("Welcome", end=" ")
print("to Python")
```

```
Welcome to Python
```

```
with open("ani.txt", "w") as f:
    print("This is written to file", file=f)
```

jupyter ani.txt 3 minutes ago

File Edit View Language

1 This is written to file

```
import time
for i in range(3):
    print(i, end=" ", flush=True)
    time.sleep(1)
```

```
0 1 2
```

Reading from a file

```
with open("ani.txt","r") as f:
    data=f.read()
print(data)
```

```
This is written to file
```

Conditional Statements

Conditional statement allows a program to test several conditions and execute instructions based on which condition is true.

Some decision control statements are:

- ✓ if
- ✓ if-else
- ✓ nested if

- ✓ if-elif-else

- **if**

It help us to run a particular code, but only when a certain condition is met or satisfied. A if only has one condition to check.

```
#TO CHECK IF A PERSON IS ELIGIBLE TO VOTE
age=int(input())
if age >= 18:
   print("ELIGIBLE TO VOTE")
```

```
20
ELIGIBLE TO VOTE
```

- **if-else**

The if-else statement evaluates the condition and will execute the body of if if the test condition is True, but if the condition is False, then the body of else is executed.

```
#TO CHECK IF A PERSON IS ELIGIBLE TO VOTE OR NOT
age=int(input())
if age >= 18:
   print("ELIGIBLE TO VOTE")
else:
    print("NOT ELIGIBLE TO VOTE")
```

```
12
NOT ELIGIBLE TO VOTE
```

- **Nested if**

Nested if statements are an if statement inside another if statement.

```
#PYTHON PROGRAM FOR GREATEST OF 3 NUMBERS
a = 35
b = 30
c = 45
if a > b:
    if a > c:
        print("a value is big")
    else:
        print("c value is big")
else:
    if b > c:
        print("b value is big")
    else:
         print("c is big")
```

```
c value is big
```

```
#PYTHON PROGRAM FOR GREATEST OF 3 NUMBERS
a = 20
b = 30
c = 45
if a > b:
    if a > c:
        print("a value is big")
    else:
        print("c value is big")
elif b > c:
    print("b value is big")
else:
     print("c is big")
```

```
c is big
```

- **if-elif-else (chained conditional)**

The if-elif-else statement is used to conditionally execute a statement or a block of statements.

```
#PROGRAM TO CHECK THE NUMBER IS +VE, -VE OR ZERO
x = -10
if x > 0:
    print("POSITIVE")
elif x < 0:
    print("NEGATIVE")
else:
    print("ZERO")
```

```
NEGATIVE
```

- **Inline (ternary) conditional:**
  - ✓ An inline statement is a short statement written in a single line instead of spreading it across multiple lines.
  - ✓ It is used to make the code short and cleaner.
  - ✓ Inline statements are useful when the logic is simple and doesn't need extra lines.

```
x = 5
result = "Even" if x % 2 == 0 else "Odd"
print(result)
```

Repetition or Looping

A repetition statement is used to repeat a group(block) of programming instructions.

In Python, we generally have two loops/repetitive statements:

- ✓ while loop
- ✓ for loop

While Loop

In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

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

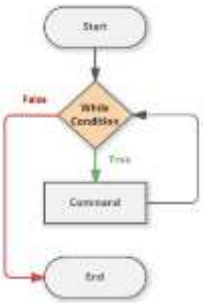
Ex :

```
i = 0
while (i < 3):
    print("Hello DataScience")
    i = i + 1
```

```
Hello DataScience
Hello DataScience
Hello DataScience
```

```
i=0
while(i<=5):
    print(i)
    i+=1
```

```
0
1
2
3
4
5
```



User input validation

User input validation ensures that the input entered by the user is correct and meets the program's requirements. We often use a while loop to keep asking for input until it is valid.

```
pswd=""
while pswd!='abcxyz':
  pswd=input("Enter password: ")
print("Access Granted!")
```

```
Enter password: abc
Enter password: hello
Enter password: abcxyz
Access Granted!
```

Infinite loop with break

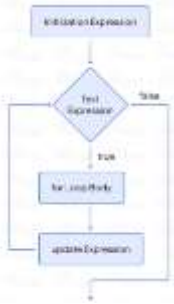
An infinite loop is a loop that keeps running forever unless stopped using a break statement. We use this pattern to repeatedly ask user input until a certain condition is met.

```
while True:
  res=input("Enter EXIT to quit ")
  if res=="exit":
    print("Goodbye")
    break
```

```
Enter EXIT to quit hi
Enter EXIT to quit exit
Goodbye
```

for loop

A for loop is used to iterate over a sequence that is either a list, tuple, dictionary, or a set (or) For loops are used for sequential traversal, for example: traversing a list or string or array etc. We can execute a set of statements once for each item in a list, tuple, or dictionary. In Python, there is “for in” loop which is similar to foreach loop in other languages. It can be used to iterate over a range and iterators.



Syntax:

```
for iterator_var in sequence:
    statements(s)
```

The range() Function

With the help of the range() function, we may produce a series of numbers. range(10) will produce values between 0 and 9. (10 numbers).

We can give specific start, stop, and step size values in the manner range(start, stop, step size). The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Loop with **range()**

```
a=range(5)
print(a)
```

```
range(0, 5)
```

```
a=range(5)
for i in a:
    print(i,end=" ")
```

```
0 1 2 3 4
```

```
a=range(10,20,2)
for i in a:
    print(i,end=" ")
```

```
10 12 14 16 18
```

List, Tuple, String, and Dictionary Iteration Using for Loop

```
#list iteration
a=[10,20,30,40,50]
for i in a:
    print(i,end=" ")
```

```
10 20 30 40 50
```

```
#tuple iteration
a=(10,20,30,40,50)
for i in a:
    print(i,end=", ")
```

```
10, 20, 30, 40, 50,
```

```
#set iteration
a={10,20,30,40,50}
for i in a:
    print(i,end=" ")
```

```
50 20 40 10 30
```

```
#dictionary iteration
a={1:'a',2:'b',3:'c'}
for i in a:
    print("%d-%c "%(i,a[i]),end=" ")
```

```
1-a  2-b  3-c
```

```
#dictionary iteration
a={1:'a',2:'b',3:'c'}
for i in a:
    print(i,a[i],end="\n")
```

```
1 a
2 b
3 c
```

```
for i in "DataScience":
    print(i,end=' ')
```

```
D a t a S c i e n c e
```

Looping with conditions

```
a=[2,6,3,8,9]
for i in a:
    if i%2==0:
        print(f'{i} is even')
    else:
        print(f'{i} is odd')
```

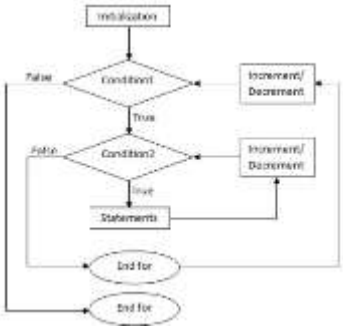
```
2 is even
6 is even
3 is odd
8 is even
9 is odd
```

Nested Loops in Python

Python programming language allows to use one loop inside another loop which is called nested loop.

Syntax:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
    statements(s)
```



Step by step explanation (read from text book pg : 61

Program for multiplication table

```
for i in range(1,3):
    for j in range(1,5):
        print(i,'*',j,'=',i*j)
    print()
```

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

Flattening a 2D list

```
#flattening a 2D list
a=[[1,2,3],[4,5,6],[7,8,9]]
l=[]
for i in a:
    for j in i:
        l.append(j)
print(l)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Factorial of a number

```
n=int(input())
res=1
for i in range(n,0,-1):
    res*=i
print(res)
```

```
6
720
```

Check prime number or not

```
n=int(input())
flag=True
for i in range(2,n):
    if n%i==0:
        flag=False
        break
if flag:
    print("Prime Number")
else:
    print("Not Prime")
```

```
37
Prime Number
```

Iterating through Nested Dictionaries

```
data={
    'Ram':{'mat':56,'eng':70},
    'Lakshman':{'mat':90,'eng':59}
}
for i,j in data.items():
    print("marks of ",i)
    for a,b in j.items():
        print(a,b)
```

```
marks of  Ram
mat 56
eng 70
marks of  Lakshman
mat 90
eng 59
```

Loop Control Statements (visit text book examples also pg: 54)

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

- ✓ continue statement
- ✓ break statement
- ✓ pass statement

continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next.

```
for i in range(5):
    if i==3:
        continue
    print(i,end=" ")
```

```
0 1 2 4
```

```
for i in range(5):
    print(i,end=" ")
    if i==3:
        continue
```

```
0 1 2 3 4
```

break Statement

With the break statement we can stop the loop before it has looped through all the items.

```
for i in range(5):
    if i==3:
        break
    print(i,end=" ")
```

```
0 1 2
```

```
for i in range(5):
    print(i,end=" ")
    if i==3:
        break
```

```
0 1 2 3
```

pass Statement

We use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.

```
for i in range(5):
    pass
print(i)
```

```
4
```

Fruitful functions

- In Python, when a function gives back an output using the return statement, it is called a fruitful

function.

- These functions are the opposite of void (non-fruitful) functions, which do not return any value and are only used for actions like printing, updating, etc.
- Fruitful functions are very useful in real programming because:
  - ✓ They allow us to reuse results and store them in variables.
  - ✓ They support modular programming (breaking code into reusable blocks).
  - ✓ They can be tested easily (unit testing of return values).
  - ✓ They increase code readability and reduce redundancy.

Key points

- ✓ Must use return: A fruitful function must have a return statement. Without it, the function returns None by default.
- ✓ Return ends function: Once return is executed, the function ends immediately—even if other code follows.
- ✓ Return can send expressions: You can return variables, expressions, or even multiple values.
- ✓ Multiple return values: Python allows returning more than one value using tuples.
- ✓ Return values can be stored: Returned values can be used in further operations.

Syntax:

```
def function_name(parameters):
    # computation
    return value
```

Ex:

```
def add(a,b):
    return a+b

ad=add(10,20)
print(ad)
```

```
30
```

Return values

- In Python, the return statement is an essential part of a function that sends back the result of the function to the calling environment.
- When a function is executed, it may perform a task or calculation. If that result needs to be used later, it must be returned using the return keyword.
- This statement marks the end of a function's execution and provides output to the caller.
- If no return statement is written, the function returns None by default.

Importance of Return Values

- ✓ Communication: Allows a function to give output back to the caller.
- ✓ Reusability: Output can be reused elsewhere in the program.
- ✓ Data Flow: Enables transfer of data between functions.
- ✓ Clean Design: Promotes modular and readable code.

```
def add(a,b):
    return a+b

ad=add(10,20)
print(ad)
```

```
30
```

Returning Multiple values from function

```
def cal(x,y):
    a=x+y
    s=x-y
    m=x*y
    d=x/y
    return a,s,m,d
add,sub,mul,div=cal(4,2)
print("addition ",add)
print("substraction ",sub)
print("multiplication ",mul)
print("division ",div)
```

Function Parameters

1. Positional Parameters

- ✓ Arguments are passed in order.
- ✓ First argument goes to the first parameter, second to second, and so on.
- ✓ Order of arguments is important.
- ✓ Used when all arguments are mandatory and in fixed sequence.

```
def fun(name,age):
    print("I'm = ",name)
    print("My age is = ",age)
fun(20,"syam")
```

```
I'm =  20
My age is =  syam
```

2. Keyword Parameters
- ✓ Arguments are passed using parameter names (key=value).
- ✓ Order does not matter because names are used for matching.
- ✓ Useful when functions have many parameters.

```
def fun(fname,lname):
    print("first name = ",fname)
    print("last name = ",lname)
fun(lname="Van Rossum",fname="guido")
```

```
first name =  guido
last name =  Van Rossum
```

3. Default Parameters
- ✓ A default value is assigned to the parameter in the function definition.
- ✓ If no value is passed, the default is used.
- ✓ Reduces the number of arguments needed at the time of function call.

```
def fun(x,y=10):
    print("X = ",x)
    print("Y = ",y)
fun(5)
```

```
X =  5
Y =  10
```

4. Variable-Length Positional Parameters (*args)
- ✓ Allows the function to accept any number of positional arguments.
- ✓ The arguments are received as a tuple.

```
def fun(*nums):
    print(*nums)
fun(20,30,40,50)
```

```
20 30 40 50
```

5. Variable-Length Keyword Parameters (**kwargs)
- ✓ Allows passing multiple keyword arguments (key=value).
- ✓ These are collected into a dictionary inside the function.

```
def fun(**nums):
    print(nums)
fun(a=20,b=30,c=40,d=50)
```

```
{'a': 20, 'b': 30, 'c': 40, 'd': 50}
```

Local and Global Scope in Functions

The scope of a variable refers to the region of the program where the variable is recognized and accessible. Python has two main scopes:
- Local Scope
- Global Scope

These determine where a variable can be used or modified.

Local Scope

Local scope refers to variables declared inside a function. These variables are accessible only within that function and not outside.

Key Points:
- ✓ A variable defined inside a function is called a local variable.
- ✓ It is created when the function is called.
- ✓ It is destroyed when the function ends.
- ✓ Local variables are not accessible outside the function.
- ✓ Different functions can have local variables with the same name, as they are independent.
- ✓ If you try to access a local variable from outside, it causes an error.

```
def func():
    b=20
    print("global variable ",a)
    print("local variable ",b)
a=10
func()
#print(b)  #Leads to Error
```

Global Scope

Global scope refers to variables declared outside any function. These variables are accessible throughout the program, including inside functions.

Key Points:
- ✓ A variable declared outside all functions is called a global variable.
- ✓ It can be accessed from any function in the program.
- ✓ If you modify a global variable inside a function, you must use the global keyword.
- ✓ Without global, assigning a value to a variable inside a function creates a new local variable.

Ex:

```
def f():
    s = "programming"
    print("Inside : ",s)

s = "python"
f()
print("Outside : ",s)
```

```
Inside :  programming
Outside :  python
```

✓ Global variables are useful for values that are shared across functions.

Global Keyword

- The global keyword is used to declare that a variable inside a function refers to the global version, not a new local one.
- It allows the function to modify global variables.

```
def f():
    global s
    s = "programming" #modifying global in Local scope
    print("Inside : ",s)
s = "python"
f()
print("Outside : ",s)
```

```
Inside :  programming
Outside :  programming
```

Function Composition:

Function composition is a process where the output of one function is used as the input to another. It means combining two or more functions to perform a sequence of operations.

- Function composition allows chaining functions together.
- It helps in breaking down complex problems into smaller functions.
- The result of one function is passed into another.
- Improves readability, modularity, and code reuse.
- Can be done using simple functions or lambda expressions.

Structure of Function Composition

f(g(x))

✓ First, g(x) is evaluated.

✓ Then, f(...) is applied to the result.

```
def double(x):
    return x * 2
def square(x):
    return x * x
#square of double of 3 → square(double(3))
result = square(double(3))  # Output: 36
print(result)
```

```
36
```

Function Composition Using lambda

- lambda is an anonymous function (function without a name).
- Useful for writing short, single-expression functions.
- Can be composed similarly to normal functions.
- Syntax: lambda arguments: expression

```
double = lambda x: x * 2
square = lambda x: x * x
# First double(4) = 8 → square(8) = 64
result = square(double(4))
print(result)
```

```
64
```

Recursion

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem.

Keypoints:

- Recursion is used to break down a problem into smaller sub-problems.
- A recursive function calls itself with a smaller input.
- Every recursive function must have a base case to stop recursion.
- Without a base case, the function will run infinitely and cause an error.
- Recursion is often used in solving problems like factorial, Fibonacci series, searching, sorting, tree traversal, etc.
- It is an alternative to loops, useful in problems with repetitive structure.

Structure:

```
def function_name(parameters):
    if base_case_condition:
        return base_result
    else:
        return function_name(smaller_problem)
```

Ex:

```
def fun(n):
    if n==0 or n==1:
        return 1
    else:
        return n*fun(n-1)
n=int(input())
fun(n)
```

```
6

720
```

Advantages

- Makes code simpler and shorter, especially for problems with repetitive patterns.
- Reduces complexity for problems like tree/graph traversal, permutations, etc.
- Natural fit for divide-and-conquer strategies.
- Improves code readability in many mathematical problems.

Disadvantages

- Uses more memory due to multiple function calls (call stack).
- Slower than iteration in some cases.
- Improper base case or logic may lead to infinite recursion.
- Difficult to debug for beginners.

Fibonacci using Recursion

The Fibonacci series is a sequence where each number is the sum of the previous two numbers, starting from 0 and 1.

Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Formula:

F(n) = F(n-1) + F(n-2)

With base cases:

F(0) = 0

F(1) = 1

Example:

```
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
n=int(input())
print(fib(n))
```

```
7
13
```

IMPORTANT PROGRAMS:

```
#factorial of a number
n=int(input())
res=1
for i in range(1,n+1):
    res*=i
print(res)
```

```
5
120
```

```
#fibonacci Series
f1,f2,f3=0,1,0
for i in range(10):
    print(f1,end=' ')
    f3=f1+f2
    f1,f2=f2,f3
```

```
0 1 1 2 3 5 8 13 21 34
```

```
#check prime number
n=int(input())
flag=True
for i in range(2,n):
    if n%i==0:
        flag=False
        break
if flag:
    print("Prime Number")
else:
    print("Not Prime")
```

```
47
Prime Number
```

```
#primes in range
s,e=map(int,input().split())
for i in range(s,e+1):
    flag=True
    for j in range(2,i):
        if i%j==0:
            flag=False
            break
    if flag:
        print(i,end=' ')
```

```
10 20
11 13 17 19
```

```
#palindrome or not
n=int(input())
rem=rev=0
temp=n
while n!=0:
    rem=n%10
    rev=rev*10+rem
    n=n//10
if temp==rev:
    print("PALINDROME")
else:
    print("NOT PALINDROME")
```

```
12321
PALINDROME
```