

# CONTROL FLOW, FUNCTIONS

# 3

*Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: state, while, for, break, continue, pass; Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Lists as arrays. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search.*

## 3.1 Conditionals

In programming and scripting languages, conditional statements or conditional constructs are used to perform different computations or actions depending on whether a condition evaluates to true or false.

### 3.1.1 Floor division and modulus

- The ***floor division*** operator divides two numbers and ***rounds down to an integer***.
- Conventional division returns a floating-point number:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

- If suppose it is about hours, we don't normally write hours with decimal points.
- Floor division returns the integer number of hours, dropping the fraction part:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

- ***modulus operator***, %, which divides two numbers and ***returns the remainder***.

```
>>> remainder = minutes % 60
>>> remainder
45
```

### Benefits of modulus operator

- You can check whether one number is divisible by another. if  $x \% y$  is zero, then  $x$  is divisible by  $y$ .
- You can extract the right-most digit or digits from a number. For example,  $x \% 10$  yields the right-most digit of  $x$  (in base 10).
- We can use the modulus operator as  $x \% 100$  yields the last two digits.

## 3.1.2 Boolean expressions

- A ***boolean expression*** is an expression that is either ***true or false***.
- The following examples use the operator `==`, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

- True and False are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

## 3.1.3 Operators

- Operators are particular symbols which ***operate on some values*** and produce an output.
- The values are known as Operands.

**Example:**

$4 + 5 = 9$
-------------

Here 4 and 5 are Operands and (+) , (=) signs are the operators. They produce the output 9.

**Python supports the following operators:**

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.
7. Bitwise Operators.

**Arithmetic Operators:****Arithmetic Operators:**

Operators	Description
//	Perform Floor division(gives integer value after division)
+	To perform addition
-	To perform subtraction
*	To perform multiplication
/	To perform division
%	To return remainder after division(Modulus)
**	Perform exponent(raise to power)

**Example:**

<pre>&gt;&gt;&gt; 10+20 30 &gt;&gt;&gt; 20-10 10 &gt;&gt;&gt; 10*2 20 &gt;&gt;&gt; 10/2</pre>
---

```
5
>>> 10%3
1
>>> 2**3
8
>>> 10//3
3
>>>
```

### Relational Operators:

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<>	Not equal to(similar to !=)

### Example:

```
>>> 10<20
True
>>> 10>20
False
>>> 10<=10
True
>>> 20>=15
True
>>> 5==6
False
>>> 5!=6
True
>>> 10<>2
True
>>>
```

## Assignment Operators:

Operators	Description
=	Assignment
/=	Divide and Assign
+=	Add and assign
-=	Subtract and Assign
*=	Multiply and assign
%=	Modulus and assign
**=	Exponent and assign
//=	Floor division and assign

## Example:

```
>>> c=10
>>> c
10
>>> c+=5
>>> c
15
>>> c-=5
>>> c
10
>>> c*=2
>>> c
20
>>> c/=2
>>> c
10
>>> c%=3
>>> c
1
>>> c=5
>>> c**=2
>>> c
25
>>> c//=2
>>> c
```

```
12
```

```
>>>
```

### Logical Operators:

Operators	Description
and	Logical AND(When both conditions are true output will be true)
or	Logical OR (If any one condition is true output will be true)
not	Logical NOT(Compliment the condition i.e., reverse)

### Example:

```
a=5>4 and 3>2
```

```
print a
```

```
b=5>4 or 3<2
```

```
print b
```

```
c=not(5>4)
```

```
print c
```

### Output:

```
>>>
```

```
True
```

```
True
```

```
False
```

```
>>>
```

### Membership Operators:

Operators	Description
in	Returns true if a variable is in sequence of another variable, else false.
not in	Returns true if a variable is not in sequence of another variable, else false.

### Example:

```

a=10
b=20
list=[10,20,30,40,50];
if (a in list):
    print "a is in given list"
else:
    print "a is not in given list"
if(b not in list):
    print "b is not given in list"
else:
    print "b is given in list"

```

### Output:

```

>>>
a is in given list
b is given in list

>>>

```

### Identity Operators:

Operators	Description
is	Returns true if identity of two operands are same, else false
is not	Returns true if identity of two operands are not same, else false.

### Example:

```

a=20
b=20
if( a is b):
    print ?a,b have same identity?
else:
    print ?a, b are different?
b=10
if( a is not b):
    print ?a,b have different identity?
else:

```

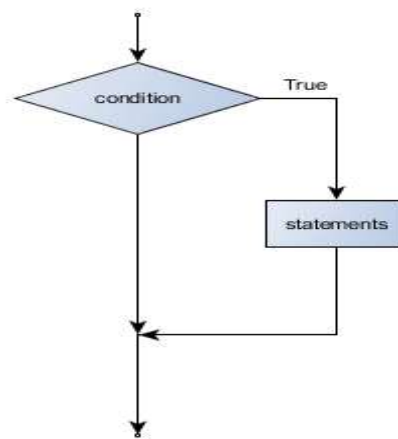
```
print ?a,b have same identity?
```

**Output:**

```
>>>  
a,b have same identity  
a,b have different identity  
>>>
```

### 3.1.4 Conditional execution (if)

- **Conditional statements** give us this ability *to check conditions* and act accordingly.
- The simplest form is the if statement:
- if statements have the same structure as function definitions: a header followed by an indented body.
- Statements like this are called *compound statements*.



**Syntax:**

```
if condition:  
    Statements 1...  
    .....  
    Statements n.
```



### Example :

```
>>> if x > 0:
    print('x is positive')

x is positive.

>>> if 2**2==4:
    Print('Obvious')

Obvious
```

- The Boolean expression after if is called the ***condition***.
- If it is true, the indented statement runs.
- If not, nothing happens.
- There is no limit on the number of statements that can appear in the body, but there has to be at least one.
- Blocks are delimited by indentation
- The Ipython shell automatically increases the indentation depth after a column : sign; to decrease the indentation depth, go four spaces to the left with the Backspace key.
- Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet).
- In that case, you can use the ***pass statement***, which does nothing.

```
if x < 0:

pass # TODO: need to handle negative values!
```

### Example Program to check if the number is even

```
no= int(input("Enter a number"))
if ((no%2)==0):
    print("Even number")
```

#### Output

```
Enter a number 4
Even number
```

## 3.1.5 Alternative execution ( if-else)

- A second form of the if statement is “*alternative execution*”, in which there are two possibilities and the condition determines which one runs.
- If the first condition is true the corresponding statement is executed
- If the condition is false the else part will be executed

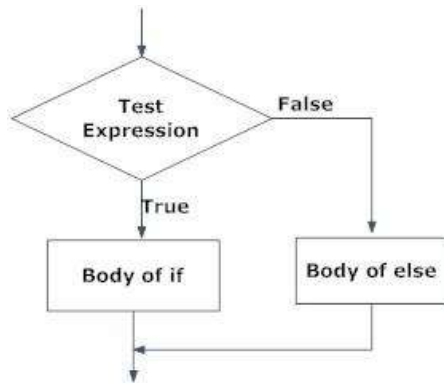


Fig: Operation of if...else statement

### Syntax

```

If condition:
    Statement
else:
    Statement.
  
```

### Example: Program to check if the number is even or odd

```

x= int(input("Enter a number"))
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
  
```

### Output

```

Enter a number 4
Even number
  
```

- If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message.
- If the condition is false, the second set of statements runs.
- Since the condition must be true or false, exactly one of the alternatives will run.
- The alternatives are called **branches**, because they are branches in the flow of execution.

## 3.1.6 Chained conditionals (if-elif-else)

- There are more than two possibilities and we need more than two branches.
- One way to express a computation like that is a **chained conditional**:

## Syntax

```
if condition:
    statement
elif condition:
    statement
else:
    statement
```

### Example 1: Program to explicit if..elif...else

```
a = 10
if a == 1:
    print(1)
elif a == 2:
    print(2)
else:
    print('A lot')
```

#### Output

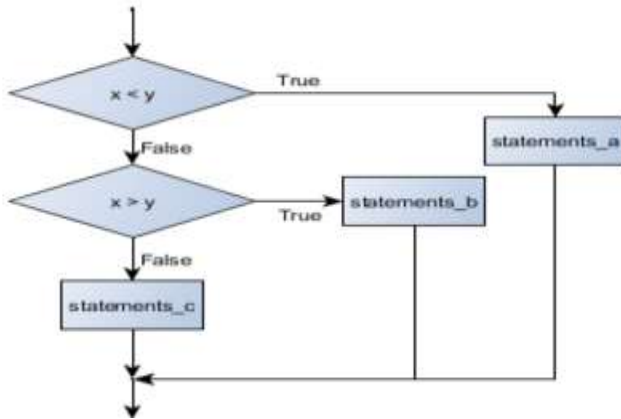
A lot

### Example 2 : To find the largest of two numbers

```
x= int(input("Enter a number"))
y= int(input("Enter a number"))
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

#### Output

```
Enter a number
5
Enter a number
5
x and y are equal
```



- elif is an abbreviation of “else if”. Again, exactly one branch will run.
- There is no limit on the number of elif statements.
- If there is an else clause, it has to be at the end, but we can have even without else clause.

```
if choice == 'a':  
    draw_a()  
elif choice == 'b':  
    draw_b()  
elif choice == 'c':  
    draw_c()
```

- Each condition is checked in order.
- If the first is false, the next is checked, and so on.
- If one of them is true, the corresponding branch runs and the statement ends.
- Even if more than one condition is true, only the first true branch runs.

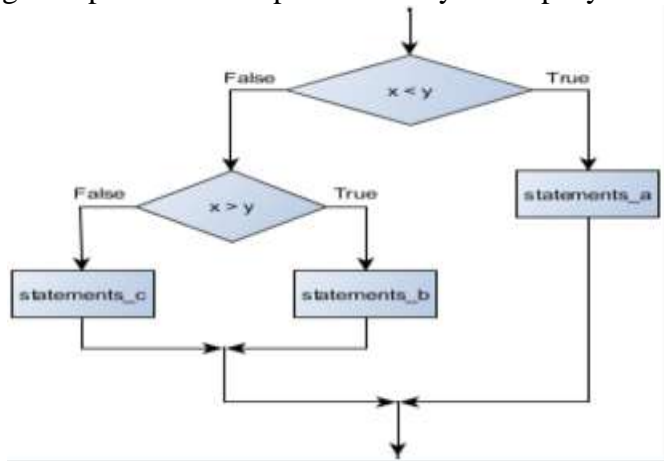
### 3.1.7 Nested conditionals

- One conditional can also be nested within another.

```
if x == y:  
    print('x and y are equal')  
else:  
    if x < y:  
        print('x is less than y')  
    else:  
        print('x is greater than y')
```

- The outer conditional contains two branches.

- The first branch contains a simple statement.
- The second branch contains another if statement, which has two branches of its own.
- Those two branches are both simple statements, although they could have been conditional statements as well.
- Logical operators often provide a way to simplify nested conditional statements.



**For example,**

- we can rewrite the following code using a single conditional:

```

x= int(input("Enter a number"))
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
  
```

### Output

Enter a number

5

x is a positive single-digit number.

- The print statement runs only if we make it pass with both the conditionals,
- so we can get the same effect with the **and** operator:

```

if 0 < x and x < 10:
    print('x is a positive single-digit number.')
  
```

- For this kind of condition, Python provides a more concise option:

```

if 0 < x < 10:
    print('x is a positive single-digit number.')
  
```

- This does the same operation that is done by the and operator but the code becomes concise and efficient.

## 3.2 Iteration

- Computers are often used to automate repetitive tasks.
- Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.
- In a computer program, repetition is also called **iteration**.

### 3.2.1 State

- An algorithm describe a *computation*
- when *executed*, proceeds through a finite number of well-defined *successive states*,
- States are the temporary *intermediate* stages in the flow of execution.
- *Partial output* like intermediate output obtained called states.
- These outputs are then eventually produces the output terminating at a final ending state.
- The transition from one state to the next is not necessarily *deterministic*

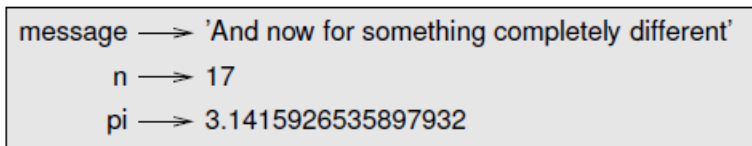


Figure 3.1 State diagram

- **State diagram:** A graphical representation of a set of variables and the values they refer to.
- Reassignment**

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

- It is legal to make more than one assignment to the same variable.
- A new assignment makes an existing variable refer to a new value (and stop referring to the old value).
- This is called as *reassignment*
- The first time we display x, its value is 5; the second time, its value is 7.

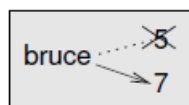


Figure 3.1 shows what **reassignment** looks like in a state diagram.

```
>>> a = 5
>>> b = a # a and b are now equal
>>> a = 3 # a and b are no longer equal
>>> b
5
```

- The third line changes the value of a but does not change the value of b, so they are no longer equal.
- Reassigning variables is often useful, but you should use it with caution.
- If the values of variables change frequently, it can make the code difficult to read and debug.

### Updating variables

- A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

```
>>> x = x + 1
```

- This means “get the current value of x, add one to it, and then update x with the new value.”
- If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
```

NameError: name 'x' is not defined

- Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

- Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

## 3.2.2 The while statement

- while loop is used to execute number of statements or body till the condition passed in while is true.
- Once the condition is false, the control will come out of the loop.

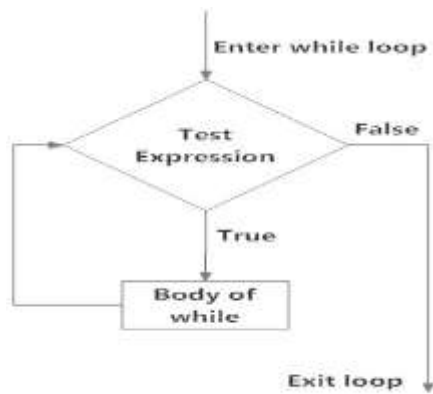


Fig: operation of while loop

### Syntax:

```
while <expression>:
```

```
    Body
```

- Here, body will execute multiple times till the expression passed is true.
- The Body may be a single statement or multiple statements.

### Example:

```
a=10
while a>0:
    print "Value of a is",a
    a=a-2

print "Loop is Completed"
```

#### Output:

```
>>>
Value of a is 10
Value of a is 8
Value of a is 6
Value of a is 4
Value of a is 2
Loop is Completed
```



### Explanation:

- Firstly, the value in the variable is initialized.
- Secondly, the condition/expression in the while is evaluated. Consequently if condition is true, the control enters in the body and executes all the statements. If the condition/expression passed results in false then the control exit the body and straight away control goes to next instruction after body of while.
- Thirdly, in case condition was true having completed all the statements, the variable is incremented or decremented. Having changed the value of variable step second is followed. This process continues till the expression/condition becomes false.
- Finally Rest of code after body is executed.

### The flow of execution for a while statement:

1. Determine whether the condition is true or false.
  2. If false, exit the while statement and continue execution at the next statement.
  3. If the condition is true, run the body and then go back to step 1.
- This type of flow is called a loop because the third step loops back around to the top.
  - The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates.
  - Otherwise the loop will repeat forever, which is called an **infinite loop**.
  - An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat”, are an infinite loop.

### Example 1: Program to add digits of a number/ sum of digits of a number

```
n=153
sum=0
while n>0:
    r=n%10
    sum+=r
    n=n/10
print sum
```

#### Output:

9

### Example 2: Countdown program

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

- if n is zero or negative, the loop never runs.
- Otherwise, n gets smaller each time through the loop, so eventually we have to get to 0.
- You can almost read the while statement as if it were English. It means,
- “While n is greater than 0, display the value of n and then decrement n.
- When you get to 0, display the word Blastoff!”

### Example 3: Sequence of odd/even:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0: # n is even
            n = n / 2
        else: # n is odd
            n = n*3 + 1
```

#### Output

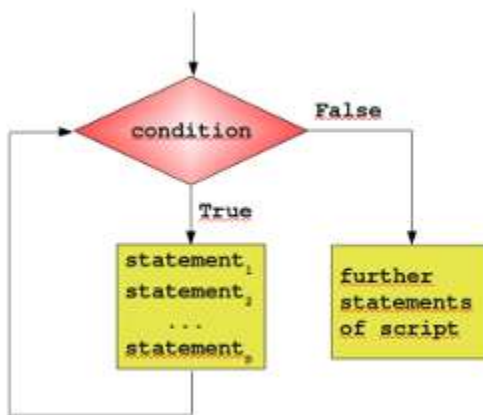
3, 10, 5, 16, 8, 4, 2, 1.

- The condition for this loop is  $n \neq 1$ , so the loop will continue until n is 1, which makes the condition false.
- Each time through the loop, the program outputs the value of n and then checks whether it is even or odd.
- If it is even, n is divided by 2
- . If it is odd, the value of n is replaced with  $n*3 + 1$ .
- For example, if the argument passed to sequence is 3,
- the resulting values of n are  
n=3, -> odd so  $n=n*3+1$   
           $n=3*3+1=10$   
n=10 -> even so  $n=n/2$

$n=10/2=5$   
 $n=5 \rightarrow$  odd odd so  $n=n*3+1$   
 $n=5*3+1=16$ ,  
 $n=16 \rightarrow$  even so  $n=n/2$   
 $n=16/2=8$   
 $n=8 \rightarrow$  even so  $n=n/2$   
 $n=8/2=4$   
 $n=4 \rightarrow$  even so  $n=n/2$   
 $n=4/2=2$   
 $n=2 \rightarrow$  even so  $n=n/2$   
 $n=2/2=1$   
 $n=1 \rightarrow$  condition fails

## while and else statement

- There is a structural similarity between while and else statement. Both have a block of statement(s) which is only executed when the condition is true.
- The difference is the block belongs to if statement executes repeatedly.
- We can attach the optional else clause with while statement.



### Syntax:

```

while(expression):
    statement_1
    statement_2
else:
    statement_3
    statement_4
.....
  
```

- The while loop repeatedly tests the expression(condition) and, if it is true it executes the first block of program statements.
- The else clause is only executed when the condition is false. It does not execute if the loop breaks, or if an exception is raised.
- If a break statement executes in first program block and terminates the loop then the else clause is not executed.
- In the following example, while loop calculates the sum of digits of integer after completing the loop, the else statement executes

### Program: sum of digits of integer number using while loop and else

```
num=int(input("Enter the number to find its reverse:"))
sum_digit=0
while(num>0):
    digit=num%10
    sum_digit = sum_digit + digit
    num = num//10
else:
    print("sum of digits of integer is:", sum_digit)
```

#### Output:

Enter number to find its reverse:1234  
sum of digits of integer is: 10(1+2+3+4)

## 3.2.3 for

- The for loop processes each item in a sequence,
- Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed.
- The general form of a for loop is:

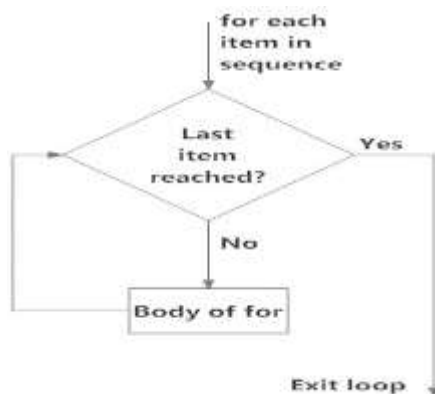


Fig: operation of for loop

## Syntax:

```
for LOOP_VARIABLE in SEQUENCE:  
    STATEMENTS
```

- It has a header terminated by a colon (:) and a body consisting of a sequence of one or more statements indented the same amount from the header.
- The loop variable is created when the for statement runs, so you do not need to create the variable before then.
- Each iteration assigns the loop variable to the next element in the sequence, and then executes the statements in the body.
- The statement finishes when the last element in the sequence is reached.
- This type of flow is called a **loop** because it loops back around to the top after each iteration.

## For in string

```
for friend in ['Margot', 'Kathryn', 'Prisila']:  
    invitation = "Hi " + friend + ". Please come to my party on Saturday!"  
    print(invitation)
```

- Often times you will want a loop that iterates a given number of times, or that iterates over a given sequence of numbers. The range function comes in handy for that.

## For with range function

```
>>> for i in range(5):  
    print('i is now:', i)
```

### Output

```
i is now 0  
i is now 1  
i is now 2  
i is now 3  
i is now 4
```

## for Iterating over tuple

- The following example counts the numbers of even and odd numbers from a series of numbers. In the below example a tuple named numbers is declared which holds the integer 1 to 9.

- The best way to check if a given number is even or odd is use to the modulus operation (%).The % operator return the remainder when dividing two numbers Modulus of 8% 2 returns 0 as 8 is divided by 2, therefore 8 is even and modulus of 5% 2 returns 1 therefore 5 is odd.
- The for loop iterates through the tuple and we test modulus of x % 2 is true or not, for every item in the tuple and the process will be continue until we reach the end of the tuple.
- When it true count, even is increased by one otherwise count\_odd is increased by one. Finally ,we print the number even and odd numbers through print statement.

### **Program: Count Odd & Even numbers in tuple**

```
#declaring tuples with values
numbers =(1,2,3,4,5,6,7,8,9)
odd_no_count =0
even_no_count =0
for x in numbers :
    if ( x%2) ==0)
        even_no_count += 1
    else :
        odd_no_count += 1
print("Total odd number count is tuples :", odd_no_count)
print("Total Even number count is tuples :", even_no_count)
Output :
Total odd number count in tuples : 5
Total even number count in tuples : 4
```

### **Iterating over list**

In the following example for loop iterates through the list “week\_day” and prints each day and its corresponding python type

### **Program: for printing weekday using for loop**

```
#declaring tuples with values
week_days = ["sun", "mon", "tues", "wed", "thur", "fri", "sat"]
for days in week_days:
    print("Week days:", days)

Output:
Week days: sun
Week days: mon
```

```
Week days: tues
Week days: wed
Week days: thur
Week days: fri
Week days: sat
```

## iterating over dictionary.

In the following example for loop iterates through the dictionary “color” through its keys and prints each key.

### Program: for loop to iterate dictionary keys

```
color={"c1": "Red", "c2": "Green", "c3": "Orange"}
for key in color:
    print("keys:", key)
```

Output:

```
keys:c1
keys:c2
keys:c3
```

## For and else

you can attach an optional else clause with for statement. In this case , syntax will be:

```
for variable_name in sequence:
    statement_1
    statement_2
    .....
else:
    statement_3
    statement_4
```

### Program: for loop to iterate dictionary values with else block

```
color = {"c1": "Red", "c2": "Green", "c3": "Orange"}
for value in color.values():
    print("values:", value)
else:
    print("All values successfully retrieved")
```

**Output:**

```
values: Red
values: Green
values: Orange
```

All values are successfully retrieved

The else clause is only executed after completing the for loop. If a break statement executes in first program block and terminates the loop then the else clause will not execute

### 3.2.4 Break

- break statement is a jump statement that is used to pass the control to the end of the loop.
- When break statement is applied the control points to the line following the body of the loop
- Hence applying break statement makes the loop to terminate and controls goes to next line pointing after loop body.
- Consider the example to take input from the user until they type done.

```
write:
while True:
line = input('> ')
if line == 'done':
break
print(line)
print('Done!')
```

**output:**

```
> not done
not done
> done
Done!
```

- The loop condition is True, which is always true, so the loop runs until it hits the break statement.
- Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop.
- Otherwise the program echoes whatever the user types and goes back to the top of the loop.
- This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively (“stopwhen this happens”) rather than negatively (“keep going until that happens”)

#### Example

```
for i in [1,2,3,4,5]:
    if i==4:
        print "Element found"
```



```
break
print i,
```

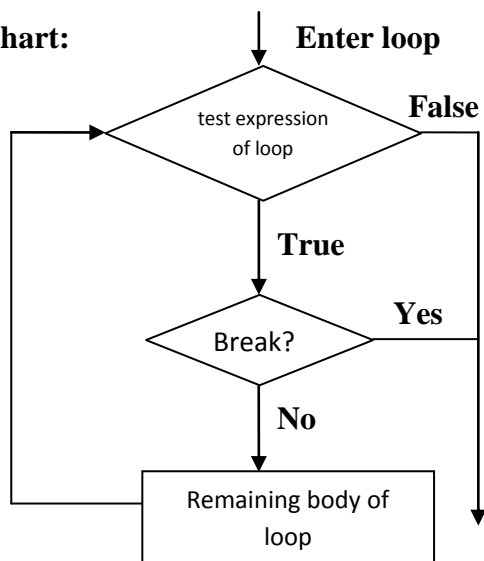
**Output:**

```
>>>
1 2 3 Element found
>>>
```

## Break

- if there is an optional else statement in while or for loop it skips the optional clause also.
- When break statement is encountered then the control of the loop is exited.

**Flowchart:**



## Break in while loop

### Syntax

```
while test expression:
    #codes inside while loop
    if condition:
        break
    #codes inside while loop
→ #codes outside while loop
```

### Program: usage of break in while loop

```
value = 1
while(value>=1):
    print(:current value is:", value)
    value = value + 1
    if (value == 5):
        print("Good bye ! break for me")
        break
```

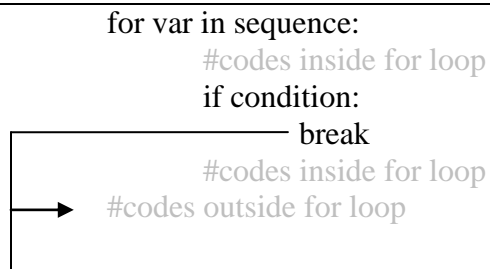
#### Output:

```
current value is: 1
current value is: 2
current value is: 3
current value is: 4
Good bye ! break for me
```

## Break in for loop

### Syntax

```
for var in sequence:
    #codes inside for loop
    if condition:
        break
    #codes inside for loop
#codes outside for loop
```



### Program: usage of break in for loop

```
initial = 1
last = 10
step = 1
for value in range(initial, last, step):
    if(value == 5)
        print("Good bye ! break for me")
        break
    print("current value is:", value)
```

#### Output:

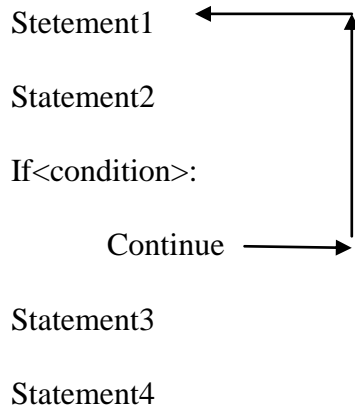
```
current value is: 1
current value is: 2
current value is: 3
```

current value is: 4  
Good bye ! break for me

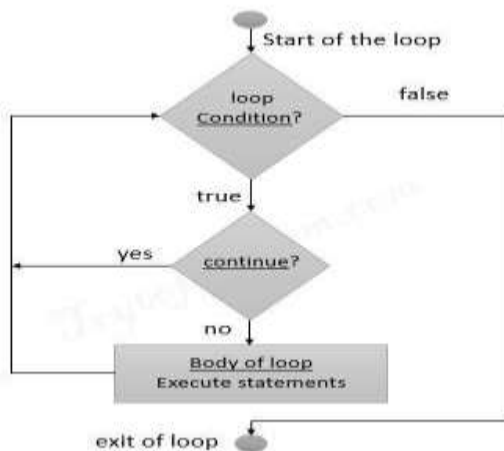
### 3.2.5 Continue

- Continue Statement is a jump statement that is used to skip the present iteration and forces next iteration of loop to take place.
- It can be used in while as well as for loop statements.

While<condition>:



**Flow chart of continue:-**



## Example

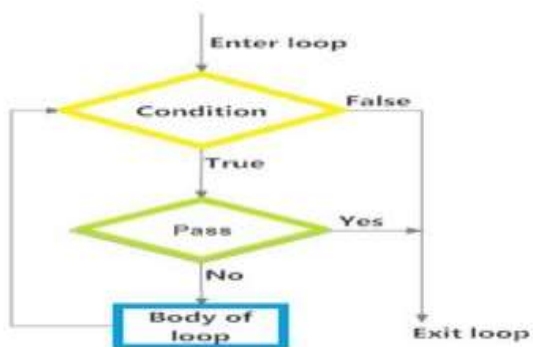
```
a=0
while a<=5:
    a=a+1
    if a%2==0:
        continue
    print a
print "End of Loop"
```

### Output:

```
>>>
1
3
5
End of Loop
>>>
```

## 3.2.6 Pass

- When you do not want any code to execute, pass Statement is used.
- It is same as the name refers to.
- It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.



### Syntax:

```
pass
```

### Example:

```
for i in [1,2,3,4,5]:  
    if i==3:  
        pass  
        print "Pass when value is",i  
    print i,
```

### Output:

```
>>>  
1 2 Pass when value is 3  
3 4 5  
>>>
```

## 3.3 Fruitful functions

- Fruitful functions are those that return a value, Such as the math functions, yield results;
- For lack of a better name, It is called as **fruitful functions**.
- Other functions, like print\_twice, perform an action but don't return a value. They are called **void functions**.
- Whenever a fruitful function is called, it means that you have something to do with the result;
- for example, you might assign it to a variable or use it as part of an expression:

```
x = math.cos(radians)  
golden = (math.sqrt(5) + 1) / 2
```

- When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)  
2.2360679774997898
```

- Void functions might display something on the screen or have some other effect, but they don't have a return value
- If you try to assign the result to a variable, you get a special value called None.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

- The value None is not the same as the string 'None'. It is a special value that has its own type:

```
>>> print type(None)
<type 'NoneType'>
```

### 3.3.1 Return Statement

- Calling the function generates a return value, which we usually assign to a variable or uses part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

- They have no return value, their return value is None.
- We are (finally) going to write fruitful functions. ie it should return a value.
- The first example is area, which returns the area of a circle with the given radius:

```
def area(radius):
    a = math.pi * radius**2
    return a
```

- In a fruitful function the return statement includes an expression.
- This statement means: "Return immediately from this function and use the following expression as a return value."
- the above code can be modified as

```
def area(radius):
    return math.pi * radius**2
```

#### Dead Code

- it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):  
    if x < 0:  
        return -x  
    else:  
        return x
```

- Since these return statements are in an alternative conditional, only one runs.
- As soon as a return statement runs, the function terminates without executing any subsequent statements.
- Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.
- In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):  
    if x < 0:  
        return -x  
    if x > 0:  
        return x
```

- This function is incorrect because if x happens to be 0, neither condition is true, nor the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

### 3.3.2 Parameters

- The function definition consists of the name of the function and a list of parameters enclosed in parentheses.
- The parameter list may be empty, or it may contain any number of parameters.
- In either case, the parentheses are required
- Inside the function, the arguments are assigned to variables called **parameters**.
- Example of a user-defined function that takes an argument:

```
>>> def print_twice(bruce):  
  
    bruce  
    bruce
```

- This function assigns the argument to a parameter named bruce.

- When the function is called, it prints the value of the parameter (whatever it is) twice.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

- You can also use a variable as an argument:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

- Michael is a parameter which holds the arguments ‘ Eric, the half a bee’

### 3.3.3 Local and Global Scope

- Scope of a variable can be determined by the part in which variable is defined.
- Each variable cannot be accessed in each part of a program.
- There are two types of variables based on Scope:
  - 1) Local Variable.
  - 2) Global Variable
- When you create a variable inside a function, it is **local** to that function, which means that it only exists inside the function.

#### **Example:**

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

- This function takes two arguments, concatenates them, and prints the result twice.

#### **Example**

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
```



Bing tiddle tiddle bang.

- When cat\_twice terminates, the variable cat is destroyed.
- If we try to print it, we get an exception:

```
>>> print cat
```

```
NameError: name 'cat' is not defined
```

### 1) Local Variables:

- Variables declared *inside a function body* is known as Local Variable.
- These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

#### Example:

```
def msg():  
    a=10  
    print "Value of a is",a  
    return
```

```
msg()  
print a #it will show error since variable is local
```

#### Output:

```
>>>  
Value of a is 10
```

Traceback (most recent call last):

```
File "C:/Python27/lam.py", line 7, in <module>  
    print a #it will show error since variable is local
```

```
NameError: name 'a' is not defined
```

```
>>>  
</module>
```

### b) Global Variable:

- Variable defined *outside the function* is called Global Variable.
- Global variable is accessed all over program thus global variable have widest accessibility.

### Example:

```
b=20
def msg():
    a=10
    print "Value of a is",a
    print "Value of b is",b
    return

msg()
print b
```

### Output:

```
>>>
Value of a is 10
Value of b is 20
20
>>>
```

## 3.3.4 Function Composition

- Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function.
- For example, The composition of two functions f and g is denoted f(g(x)).
- x is the argument of g,
- The result of g is passed as the argument of f
- The result of the composition is the result of f.

- Assume that the center point of the circle is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`.
- The first step is to find the radius of the circle, which is the distance between the two points.

```
radius = distance(xc, yc, xp, yp)
```

- The next step is to find the area of a circle with that radius.

```
result = area(radius)
```

- Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):  
    radius = distance(xc, yc, xp, yp)  
    result = area(radius)  
    return result
```

- The temporary variables `radius` and `result` are useful for development and debugging,
- we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):  
    return area(distance(xc, yc, xp, yp))
```

### 3.3.5 Recursion

- It is legal for one function to call another; it is also legal for a function to call itself.
- A function that calls itself is **recursive**;
- The process of executing it is called **recursion**.

```
def countdown(n):  
    if n <= 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n-1)
```

- If `n` is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs `n` and then calls a function named `countdown`—itself—passing `n-1` as an argument.
- if the function call is like

```
>>> countdown(3)
```

- The execution of countdown begins with  $n=3$ , and since  $n$  is greater than 0, it outputs the value 3, and then calls itself...
- The execution of countdown begins with  $n=2$ , and since  $n$  is greater than 0, it outputs the value 2, and then calls itself...
- The execution of countdown begins with  $n=1$ , and since  $n$  is greater than 0, it outputs the value 1, and then calls itself...
- The execution of countdown begins with  $n=0$ , and since  $n$  is not greater than 0, it outputs the word, "Blastoff!" and then returns.

The countdown that got  $n=1$  returns.

The countdown that got  $n=2$  returns.

The countdown that got  $n=3$  returns.

- And then you're back in `__main__`. So, the total output looks like this:

```
3
2
1
Blastoff!
```

### Example: Factorial of a number

- Factorial of a number is one of the best examples of recursion.

$0! = 1$

$n! = n(n-1)!$

- This definition says that the factorial of 0 is 1, and the factorial of any other value,  $n$ , is  $n$  multiplied by the factorial of  $n-1$ .
- So  $3!$  is 3 times  $2!$ , which is 2 times  $1!$ , which is 1 times  $0!$ . Putting it all together,  $3!$  equals 3 times 2 times 1 times 1, which is 6.
- To write a recursive function in python the first step is to decide what the parameters should be.
- In this case it should be clear that factorial takes an integer:

```
def factorial(n):
```

- If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
    if n == 0:
        return 1
```

- Otherwise, we have to make a recursive call to find the factorial of  $n-1$  and then multiply it by  $n$ :

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        recurse = factorial(n-1)  
        result = n * recurse  
    return result
```

- If we call factorial with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...  
Since 2 is not 0, we take the second branch and calculate the factorial of  $n-1$ ...  
Since 1 is not 0, we take the second branch and calculate the factorial of  $n-1$ ... Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.

$$\begin{aligned} 1! &\rightarrow 1*(1-1)! = 1*0! = 1*1 = 1 \\ 2! &\rightarrow 2*(2-1)! = 2*1! = 2*1 = 2 \\ 3! &\rightarrow 3*(3-1)! = 3*2! = 3*2 = 6 \end{aligned}$$

- Hence the result 6 will be printed.

## Example 2: Fibonacci Series

- After factorial, the most common example of a recursively defined mathematical function is fibonacci,

```
fibonacci(0) = 0  
fibonacci(1) = 1  
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

In Python, it looks like,

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:
```

```
return 1
else:
return fibonacci(n-1) + fibonacci(n-2)
```

## Infinite Recursion

- If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates.
- This is known as **infinite recursion**,
- it is generally not a good idea.

**Example:** program with an infinite recursion:

```
def recurse():
recurse()
```

- In most programming environments, a program with infinite recursion does not really run forever.
- Python reports an error message when the maximum recursion depth is reached:

```
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
File "<stdin>", line 2, in recurse
```

```
.
.
.
```

```
File "<stdin>", line 2, in recurse
```

```
RuntimeError: Maximum recursion depth exceeded
```

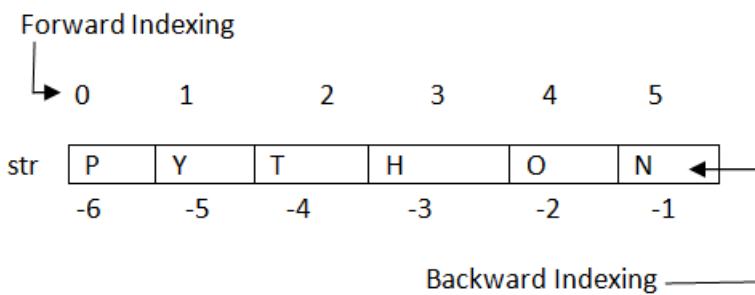
- If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call.
- And if there is a base case, check whether you are guaranteed to reach it.

## 3.4 Strings

- A string is a sequence of characters.
- You can access the characters one at a time with the bracket operator[].
- String python's are immutable( cannot be modified).
- In Python, Strings are stored as individual characters in a contiguous memory location.

- The benefit of using String is that it can be accessed from both the directions in forward and backward.
- Both forward as well as backward indexing are provided using Strings in Python.
  - Forward indexing starts with 0,1,2,3,....
  - Backward indexing starts with -1,-2,-3,-4,....

### Example:



1. `str[0]='P'=str[-6]` , `str[1]='Y' = str[-5]` , `str[2] = 'T' = str[-4]` , `str[3] = 'H' = str[-3]`
2. `str[4] = 'O' = str[-2]` , `str[5] = 'N' = str[-1]`.

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

- [1] is called an **index**.
- The index indicates which character in the sequence you want (hence the name).
- The indexing for character in python starts with 0.

```
>>> letter
'a'
```

```
>>> letter = fruit[0]
>>> letter
'b'
```

- As an index you can use an expression that contains variables and operators:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

- But the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
```

TypeError: string indices must be integers

### Simple program to retrieve String in reverse as well as normal form.

```
name="Rajat"
length=len(name)
i=0
for n in range(-1,(-length-1),-1):
    print name[i],"\t",name[n]
    i+=1
```

#### Output:

```
>>>
R      t
a      a
j      j
a      a
t      R
>>>
```

### Len

- len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```



- To get the last letter of a string, you might be tempted to try something like this:

```
>>> last = fruit[length-1]
>>> last
'a'
```

- Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.
- A lot of computations involve processing a string one character at a time.
- Often they start at the beginning, select each character in turn, do something to it, and continue until the end.
- This pattern of processing is called a **traversal**.
- One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

- This loop traverses the string and displays each letter on a line by itself.
- The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop doesn't run.
- The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

- Another way to write a traversal is with a for loop:

```
for letter in fruit:
    print(letter)
fruit b a n a n a '
index 0 1 2 3 4 5 6
```

## Strings Operators

There are basically 3 types of Operators supported by String:

1. Basic Operators.

2. Membership Operators.
3. Relational Operators.

## Basic Operators:

There are two types of basic operators in String. They are "+" and "\*".

- String Concatenation Operator :(+)
- The concatenation operator (+) concatenate two Strings and forms a new String.

### Example:

```
>>> "ratan" + "jaiswal"
```

### Output:

```
'ratanjaiswal'  
>>>
```

Expression	Output
'10' + '20'	'1020'
"s" + "007"	's007'
'abcd123' + 'xyz4'	'abcd123xyz4'

- Both the operands passed for concatenation must be of same type, else it will show an error

### Example:

```
'abc' + 3  
>>>
```

### output:

```
Traceback (most recent call last):  
  File "", line 1, in  
    'abc' + 3  
TypeError: cannot concatenate 'str' and 'int' objects  
>>>
```

### Replication Operator: (\*)

- Replication operator uses two parameter for operation. One is the integer value and the other one is the String.
- The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

#### Example:

```
>>> 5*"Vimal"
```

#### Output:

```
'VimalVimalVimalVimalVimal'
```

Expression	Output
"soono"*2	'soonosoono'
3*'1'	'111'
('\$'*5	'\$\$\$\$\$'

- We can use Replication operator in any way i.e., int \* string or string \* int. Both the parameters passed cannot be of same type.

### Membership Operators

- Membership Operators are already discussed in the Operators section. Let see with context of String.

#### There are two types of Membership operators:

1) **in:**"in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

2) **not in:**"not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

#### Example:

```
>>> str1="javatpoint"  
>>> str2='ssit'  
>>> str3="seomount"
```

```
>>> str4='java'
>>> str5="it"
>>> str6="seo"
>>> str4 in str1
True
>>> str5 in str2
>>> str5 in str2
True
>>> str6 in str3
True
>>> str4 not in str1
False
>>> str1 not in str4
True
```

### Relational Operators:

- All the comparison operators i.e., (<,>,<=,>=,==,!=,<>) are also applicable to strings. The Strings are compared based on the ASCII value or Unicode (i.e., dictionary Order).

### Example:

```
>>> "RAJAT"=="RAJAT"
True
>>> "afsha">='Afsha'
True
>>> "Z"<"z"
True
```

### Explanation:

- The ASCII value of a is 97, b is 98, c is 99 and so on. The ASCII value of A is 65, B is 66, C is 67 and so on. The comparisons between strings are done on the basis on ASCII value.

### 3.4.1 String slices

- A segment of a string is called a **slice**.
- String slice can be defined as substring which is the part of string.
- Therefore further substring can be obtained from a string.
- There can be many forms to slice a string.
- As string can be accessed or indexed from both the direction and hence string can also be sliced from both the direction that is left and right.

#### Syntax:

```
<string_name>[startIndex:endIndex],  
<string_name>[:endIndex],  
<string_name>[startIndex:]
```

#### Example:

```
>>> s = 'Monty Python'  
>>> s[0:5]  
'Monty'  
>>> s[6:12]  
'Python'  
  
>>> fruit = 'banana'  
>>> fruit[:3]  
'ban'  
>>> fruit[3:]  
'ana'
```

- The operator [n:m] returns the part of the string from the “n-eth” character to the “m-eth” character, including the first but excluding the last.
- If you omit the first index (before the colon) as fruit[:3], the slice starts at the beginning of the string.
- If you omit the second index[3:], the slice goes to the end of the string:

- If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
```

- An empty string contains no characters and has length 0, but other than that, it is the same as any other string Slice Notation:
- String slice can also be used with Concatenation operator to get whole string.

#### Example:

```
> str="Mahesh"
> str[:6]+str[6:]
'Mahesh'
```

//here 6 is the length of the string.

## 3.4.2 Immutability

- Strings are **immutable**, which means you can't change an existing string.
- You can only create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
```

TypeError: 'str' object does not support item assignment

- This error is because we are trying to replace the existing string, ie we are trying to replace 'H' at position greeting[0] with 'J' which is not possible in string.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

- This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

### 3.4.3 String functions and methods

- There are many predefined or built in functions in String. They are as follows:

capitalize()	It capitalizes the first character of the String.
count(string,begin,end)	Counts number of times substring occurs in a String between begin and end index.
endswith(suffix,begin=0,end=n)	Returns a Boolean value if the string terminates with given suffix between begin and end.
find(substring,beginIndex,endIndex)	It returns the index value of the string where substring is found between begin index and end index.
index(substring,beginIndex,endIndex)	Same as find() except it raises an exception if string is not found.
isalnum()	It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.
isalpha()	It returns True when all the characters are alphabets and there is at least one character, otherwise False.
isdigit()	It returns True if all the characters are digit and there is at least one character, otherwise False.
islower()	It returns True if the characters of a string are in lower case, otherwise False.
isupper()	It returns False if characters of a string are in Upper case, otherwise False.

isspace()	It returns True if the characters of a string are whitespace, otherwise false.
len(string)	len() returns the length of a string.
lower()	Converts all the characters of a string to Lower case.
upper()	Converts all the characters of a string to Upper Case.
startswith(str, begin=0, end=n)	Returns a Boolean value if the string starts with given str between begin and end.
swapcase()	Inverts case of all characters in a string.
lstrip()	Remove all leading whitespace of a string. It can also be used to remove particular character from leading.
rstrip()	Remove all trailing whitespace of a string. It can also be used to remove particular character from trailing.

## Examples:

### 1) capitalize()

```
>>> 'abc'.capitalize()
```

#### Output:

```
'Abc'
```

### 2) count(string)

```
msg = "welcome to sssit";
substr1 = "o";
print msg.count(substr1, 4, 16)
substr2 = "t";
print msg.count(substr2)
```

#### Output:

```
>>>
2
2
>>>
```



### 3) endswith(string)

```
string1="Welcome to SSSIT";
substring1="SSSIT";
substring2="to";
substring3="of";
print string1.endswith(substring1);
print string1.endswith(substring2,2,16);
print string1.endswith(substring3,2,19);
print string1.endswith(substring3);
```

#### Output:

```
>>>
True
False
False
False
>>>
```

### 4) find(string)

```
str="Welcome to SSSIT";
substr1="come";
substr2="to";
print str.find(substr1);
print str.find(substr2);
print str.find(substr1,3,10);
print str.find(substr2,19);
```

#### Output:

```
>>>
3
8
3
-1
>>>
```

## 5) index(string)

```
str="Welcome to world of SSSIT";
substr1="come";
substr2="of";
print str.index(substr1);
print str.index(substr2);
print str.index(substr1,3,10);
print str.index(substr2,19);
```

### Output:

```
>>>
3
17
3
Traceback (most recent call last):
  File "C:/Python27/fin.py", line 7, in
    print str.index(substr2,19);
ValueError: substring not found
>>>
```

## 6) isalnum()

```
str="Welcome to sssit";
print str.isalnum();
str1="Python47";
print str1.isalnum();
```

### Output:

```
>>>
False
True
>>>
```

## 7) isalpha()

```
string1="HelloPython";    # Even space is not allowed
print string1.isalpha();
string2="This is Python2.7.4"
print string2.isalpha();
```

**Output:**

```
>>>
True
False

>>>
```

## 8) isdigit()

```
string1="HelloPython";
print string1.isdigit();
string2="98564738"
print string2.isdigit();
```

**Output:**

```
>>>
False
True

>>>
```

## 9) islower()

```
string1="Hello Python";
print string1.islower();
string2="welcome to "
print string2.islower();
```

**Output:**

```
>>>
False
True

>>>
```

## 10) isupper()

```
string1="Hello Python";  
print string1.isupper();  
string2="WELCOME TO"  
print string2.isupper();
```

### Output:

```
>>>  
False  
True  
>>>
```

## 11) isspace()

```
string1="  ";  
print string1.isspace();  
string2="WELCOME TO WORLD OF PYT"  
print string2.isspace();
```

### Output:

```
>>>  
True  
False  
>>>
```

## 12) len(string)

```
string1="  ";  
print len(string1);  
string2="WELCOME TO SSSIT"  
print len(string2);
```

**Output:**

```
>>>
4
16
>>>
```

**13) lower()**

```
string1="Hello Python";
print string1.lower();
string2="WELCOME TO SSSIT"
print string2.lower();
```

**Output:**

```
>>>
hello python
welcome to sssit
>>>
```

**14) upper()**

```
string1="Hello Python";
print string1.upper();
string2="welcome to SSSIT"
print string2.upper();
```

**Output:**

```
>>>
HELLO PYTHON
WELCOME TO SSSIT
>>>
```

**15) startswith(string)**

```
string1="Hello Python";  
print string1.startswith('Hello');  
string2="welcome to SSSIT"  
print string2.startswith('come',3,7);
```

**Output:**

```
>>>  
True  
True  
  
>>>
```

## 16) swapcase()

```
string1="Hello Python";  
print string1.swapcase();  
string2="welcome to SSSIT"  
print string2.swapcase();
```

**Output:**

```
>>>  
hELLO pYTHON  
WELCOME TO sssit  
>>>
```

## 17) lstrip()

```
string1="  Hello Python";  
print string1.lstrip();  
string2="@@@@@@@welcome to SSSIT"  
print string2.lstrip('@');
```

**Output:**

```
>>>  
Hello Python  
welcome to world to SSSIT  
>>>
```

## 18) rstrip()

```
string1="  Hello Python  ";
print string1.rstrip();
string2="@welcome to SSSIT!!!"
print string2.rstrip('!');
```

### Output:

```
>>>
        Hello Python
@welcome to SSSIT
>>>
```

## 3.4.4 String module

- Modules are used to categorize code in Python into smaller part.
- A module is simply a file, where classes, functions and variables are defined.
- Grouping similar code into a single file makes it easy to access.
- String module consists of files that manipulate strings
- To view the functions available with string we use the command.

```
>>> dir(str)
```

- which will return the list of items inside the string module:

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattr__', '__getitem__',
 '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
 '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find',
 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
```

```
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',  
'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',  
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',  
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

- To find out more about an item in this list, we can use the help command:

```
>>> help(str.capitalize)  
  
Help on method_descriptor:  
capitalize(...)  
S.capitalize() -> string  
Return a copy of the string S with only its first character  
capitalized
```

- We can call any of these methods using **dot notation**:

```
>>> s = "brendan"  
>>> s.capitalize()  
'Brendan'
```

- str.find is a method
- To find out more about it, we can print out its **docstring**, doc , which contains documentation on the function:

```
>>> print str.find.__doc__  
S.find(sub [,start [,end]]) -> int  
Return the lowest index in S where substring sub is found,  
such that sub is contained within s[start,end]. Optional  
arguments start and end are interpreted as in slice notation.  
Return -1 on failure.
```

- Calling the help function also prints out the docstring:

```
>>> help(str.find)  
Help on method_descriptor:  
find(...)
```



`S.find(sub [,start [,end]]) -> int`

Return the lowest index in S where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

```
>>> fruit = "banana"
>>> index = fruit.find("a")
>>> print index
1

>>> "bob".find("b", 1, 2)
-1
```

- In this example, the search fails because the letter b does not appear in the index range from 1 to 2 (not including 2).

### **Character classification**

- It is one of the efficient functions available in string module.
- It is often helpful to examine a character or string and test whether it is upper- or lowercase, or whether it is a character or a digit.
- The str type includes several methods for this functionality:

```
>>> "3".isalnum() % alphabetical or numeral
True
>>> "3".isalpha() % alphabetical
False
>>> "hello".isalpha()
True
>>> "342".isdigit() % numeral
True
>>> "hello".islower() % lower case
True
>>> "hello".isupper() % upper case
False
>>> " ".isspace() % whitespace
True
>>> "Hello".istitle() % capitalised
True
>>> "hello".istitle()
False
>>>
```

- We have n number of functionalities in string module.

- Each with its own application and benefits.

### 3.4.5 Lists as arrays

- One of the most fundamental data structures in any language is the array.
- Python doesn't have a native array data structure
- But it has the list which is much more general and can be used as a multidimensional array quite easily.
- A list in Python is just an ordered collection of items which can be of any type.
- But an array is an ordered collection of items of a single type
- In principle a list is more flexible than an array
- A list is also a dynamic mutable type and this means you can add and delete elements from the list at any time.
- A list is simply a comma separated list of items in square brackets:

```
>>> myList=[1,2,3,4,5,6]
```

- We can pick up an individual element from the list.
- This is called as "slicing"
- The indexes of list start from 0.

#### Example

```
>>> print myList[2]
```

```
3
```

- We can also change an element in the list
- you can assign directly to it:

```
>>> myList[2]=100
```

```
>>> myList  
[1, 2, 100, 4, 5, 6]
```

- The slicing notation looks like array indexing but it is a lot more flexible

For example

```
>>> myList[2:5]
[100, 4, 5]
```

- This gives is a sublist from the third element to the fifth i.e. from myList[2] to myList[4].
- Notice that the final element specified i.e. [5] is not included in the slice.
- We can leave out either of the start and end indexes and they will be assumed to have their maximum possible value.

**For example**

```
>>> myList[5:]
[6]

>>> myList[:5]
[1,2,100,4,5]

>>> myList[:]
[1,2,100,4,5,6]
```

- myList[5:] is the list from List[5] to the end of the list
- myList[:5] is the list up to and not including myList[5] and is the entire list.
- myList[:] is the entire list

```
>>>myList[0:2]=[0,1]
[0, 1, 100, 4, 5, 6]
```

has the same effect as

```
>>>myList[0]=0
>>>myList[1]=1
```

## Basic array operations

- In most cases arrays are accessed by index and you can do this in Python:
- To find the maximum value in the array

```
m=0
for i in range(len(myList)):
    if m<myList[i]:
```

```
m=myList[i]
```

```
>>>m
```

```
output  
100
```

- we are now using range to generate the sequence 0,1, and so on up to the length of myList.
- For example if you wanted to return not the maximum element but its index position in the list you could use:

```
m=0  
mi=0  
for i in range(len(myList)):  
    if m<myList[i]:  
        m=myList[i]  
        mi=i
```

```
>>>m
```

```
output  
100
```

## Dimension

- Suppose you want to create an array initialized to a particular value.

```
myList=[]  
for i in range(10):  
    myList[i]=1
```

```
output
```

```
IndexErrorTraceback (most recent call last)  
<ipython-input-20-c664cad39be9> in <module>()  
    1 myList=[]  
    2 for i in range(10):  
----> 3     myList[i]=1  
  
IndexError: list assignment index out of range
```

- only to discover that this doesn't work because you can't assign to a list element that doesn't already exist.
- One solution is to use the append method to add elements one by one:

```
>>>myList=[]
    for i in range(10):
        myList.append(1)

>>> myList
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

- This works but it only works if you need to build up the list in this particular order.
- When the same situation arises in two- and multi-dimensioned arrays the problem often isn't as easy to solve with append.
- When you create an array in other languages you can usually perform operations like: without having to worry if `a[i]` already exists

```
a[i]=0
```

- Python doesn't have such a facility because lists are dynamic and don't have to be "dimensioned".
- It is fairly easy to do the same job as a dimension statement, however, using a "comprehension".

## Comprehensions

- In Python a comprehension can be used to generate a list.
- This means that we can use a comprehension to initialize a list so that it has a predefined size.
- The simplest form of a list comprehension is to create the list equivalent of a ten-element array you could write:

```
>>> myList=[0 for i in range(10)]

>>> myList
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Now have a ten-element list and you can write `myList[i]=something` without any worries - as long as `i<10`.

```
>>>myList[i]='something'  
>>>myList  
[0, 0, 0, 0, 0, 0, 0, 0, 0, 'something']
```

- You can also use the for variable in the expression. For example:

```
>>> myList=[i for i in range(10)]  
>>>myList  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>>>myList=[i*i for i in range(10)]  
>>>myList  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81].
```

## Two dimensions (two dimensional list)

- For a programmer moving to Python the problem is that there are no explicit provisions for multidimensional arrays.
- As a list can contain any type of data there is no need to create a special two-dimensional data structure.
- All you have to do is store lists within lists - after all what is a two-dimensional array but a one-dimensional array of rows.
- In Python a 2x2 array is `[[1,2],[3,4]]` with the list `[1,2]` representing the first row and the list `[3,4]` representing the second row.
- You can use slicing to index the array in the usual way( Remember indexing starts from 0). For example, if

```
>>>myArray=[[1,2],[3,4]]  
>>> myArray[0]  
[1,2]  
>>>myArray[0][1]  
2
```

- `myArray[0]` is the list `[1,2]`
- `myArray[0][1]` in the list `[[1,2],[3,4]]` is the 0th row and first element

0<sup>th</sup> row -> `[1,2]` ,  
 in this `[1,2]`  
 1st element `[1]`,  
 = 2.

```

[[1,2],[3,4]]
  ↓      ↓
  0      1
[1,2]
 ↓  ↓
 0  1

```

- As long as you build your arrays as nested lists in the way described then you can use slicing in the same way as you would array indexing. That is:
- `myArray[i][j]` is the *i,j*th element of the array.
- For example, to do something with each element in `myArray` you might write:

```

for i in range(len(myArray)):
    for j in range(len(myArray[i])):
        print myArray[i][j]

```

- Where `len(myArray)` is used to get the number of rows and `len(myArray[i])` to get the number of elements in a row.
- To create a 3x3 matrix we could use:

```

>>> myArray=[[0 for j in range(3)] for i in range(3)]

>>>myArray
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

```

- To understand this comprehension you need to see that the inner comprehension is just: `[0 for j in range(3)]` which creates a row, and then the outer comprehension just creates a list of rows.

For example: if you want to work with an *m* by *n* array use:

```
myArray=[[0 for j in range(n)] for i in range(m)]
```

```
>>>myArray=[[i*j for j in range(3)] for i in range(3)]  
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

## 3.5 Illustrative programs

### 3.5.1 Square root

```
def ntsqrt(n):  
    sgn=0  
    if n < 0:  
        sgn = -1  
        n = -n  
    val = n  
    while True:  
        last = val  
        val = (val + n / val) * 0.5  
        if abs(val - last) < 1e-9:  
            break  
    if sgn < 0:  
        return complex(0, val)  
    return val  
n=int(input('Enter upper Limit:'))  
for I in range(1,n):  
    print ('Square root of ',i,'is:',ntsqrt(i))
```

#### **Output:**

```
Enter upper Limit:5  
Square root of 1 is: 1.0  
Square root of 2 is: 1.414  
Square root of 3 is: 1.732  
Square root of 4 is: 2.0
```



### 3.5.2 GCD

```
def gcd(x,y):
    gcd=1
    if x % y == 0:
        return y
    for k in range(int(y/2),0,-1):
        if x % k == 0 and y % k ==0:
            gcd=k
            break
    return gcd
print(gcd(12,17))
print(gcd(4,6))
```

#### Output

1  
2

### 3.5.3 Exponentiation

```
n=int(input("Enter number : "))
e= int(input ("Enter exponent : "))
r=n
for I in range (1,e):
    r=n*r
print('Exponentiation is :',r)
```

#### OUTPUT:

Enter number : 3  
Enter exponent : 2  
Exponentiation is :9  
Enter number : 2  
Enter exponent : 5  
Exponentiation is :32

#### Explanation:

- Exponentiation or power is an arithmetic operation on numbers. It is repeated multiplication, just as multiplication is repeated addition.
- Exponentiation can be written with upper index, like  $x^y$ . Sometimes write powers using the ^ sign:  $3^2$  means  $3^2$

$x^y$  Where 'x' is base and 'y' is exponent, to calculate  $3^2$  multiply the number 3 itself by 2 times,  
 $3 \times 3 = 9$ .

.....

Examples:

X2 means  $x \times x$

X3 means  $x \times x \times x$

If the exponent is equal to 2, then the power is called square.

If the exponent is equal to 3, then the power is called cube.

### 3.5.4 Sum an array of numbers

```
# sum an array of number;
avg=0.0; sum1=0
print("Enter the number of Elements:")
no=int(input())
print("Enter the numbers")
for i in range(1,no+1):
a=int(input())
sum1=sum1+a
avg=sum1/no
print("Sum=",sum 1,"Average=",avg)
```

OUTPUT:

Enter the number of elements:

5

Enter the numbers

92

93

13

8

82

Sum=288 Average=57.6

### 3.5.5 Linear search

```
def ssearch(alist, item):
    pos=0
    found=False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
            print(' Element found in the list at position: ',pos)
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1
    return found

# Define Empty List
a=[]
# Read upper limit
n=int(input('Enter Upper Limit:'))
#Read 'n' numbers to list
for i in range(n):
    a.append(int(input()))
#Read an element to search in the list
x=int(input('Enter element to search in the list:'))
#Calling ssearch function
ssearch(a,x)
```

#### OUTPUT

```
23
54
65
76
12
```

```
Enter element to search in the list:54
Element found in the list at position: 1
```

**Explanation:**

- The above program reads the entered number of elements through for loop.
- The program checks the specified element and position in the list according to the user entered number.
- Then the program prints the specified element and their position.

### 3.5.6 Binary search.

```
def bsearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False
    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        print('Element found in position:'<midpoint)
    else:
        if item < alist[midpoint]:
            last = midpoint-1
        else:
            first = midpoint +1
    return found
# Define Empty list
a=[]
#Read upper limit
n=int(input('Enter upper limit:'))
#Read 'n' numbers to list
for I in range(n):
    a.append(int(input()))
#Read an element to search in the list
x=int(input('Enter Element to search in the list:'))
#Calling bsearch function
bsearch(a,x)
```

**OUTPUT**

```
Enter Upper Limit:5
4 3 5 2 1
Enter element to search in the list:5
Element found in position: 2
```

**Explanation:**

- A binary search will start by examining the middle item, if that item is the one we are searching for, we are done.
- If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items.
- If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration.
- The item, if it is in the list, must be in the upper half.

## Part A

### 1. What are the benefits of **modulus operator**?

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another. if  $x \% y$  is zero, then  $x$  is divisible by  $y$ . Also, you can extract the right-most digit or digits from a number. For example,  $x \% 10$  yields the right-most digit of  $x$  (in base 10). Similarly  $x \% 100$  yields the last two digits

### 2. What are the operators supported in python?

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.
7. Bitwise Operators.

### 3. Define Iteration.

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. In a computer program, repetition is also called **iteration**.

### 4. Write the syntax for while statement?

while loop is used to execute number of statements or body till the condition passed in while is true. Once the condition is false, the control will come out of the loop. Here, body will execute multiple times till the expression passed is true. The Body may be a single statement or multiple statement.

**Syntax:**

```
while <expression>:
```

```
    Body
```

### 5. Define for loop with syntax?

The for loop processes each item in a sequence, so it is used with Python's sequence data types - strings, lists, and tuples. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. The general form of a for loop is: It has a header terminated by a colon (:) and a body consisting of a sequence of one or more statements indented the same amount from the header.

```
for LOOP_VARIABLE in SEQUENCE:  
    STATEMENTS
```

### 6. Define break statement.

break statement is a jump statement that is used to pass the control to the end of the loop. When break statement is applied the control points to the line following the body of the loop hence applying break statement makes the loop to terminate and control goes to next line pointing after loop body.

### 7. Define continue statement with syntax.

Continue Statement is a jump statement that is used to skip the present iteration and forces next iteration of loop to take place. It can be used in while as well as for loop statements.

```
While<condition>:
```

```
    Stetement1
```

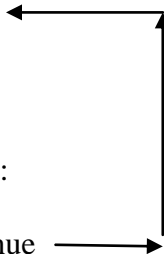
```
    Statement2
```

```
    If<condition>:
```

```
        Continue
```

```
    Statement3
```

```
    Statement4
```



## 8. Define Pass Statement.

When you do not want any code to execute, pass Statement is used. It is same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

## 9. Define Fruitful function.

Fruitful functions are those that return a value. Such as the math functions, yield results; for lack of a better name, I call them **fruitful functions**.

## 10. What are the types of variables based on scope?

There are two types of variables based on Scope:

- 1) Local Variable.
- 2) Global Variable

## 11. Define local variable and global variable with syntax?

Variables declared *inside a function body* is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared. Variable defined *outside the function* is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

## 12. Define Function composition with an example?

Function composition is a way of combining functions such that the result of each function is passed as the argument of the next function. For example, The composition of two functions  $f$  and  $g$  is denoted  $f(g(x))$ .  $x$  is the argument of  $g$ , The result of  $g$  is passed as the argument of  $f$ . The result of the composition is the result of  $f$ .

## 13. What is known as Infinite Recursion?

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
defrecurse():  
recurse()
```

## 14. Define Strings.



A string is a sequence of characters. You can access the characters one at a time with the bracket operator []. String literals are immutable (cannot be modified). In Python, Strings are stored as individual characters in a contiguous memory location. The benefit of using String is that it can be accessed from both the directions in forward and backward. Both forward as well as backward indexing are provided using Strings in Python.

- Forward indexing starts with 0,1,2,3,....
- Backward indexing starts with -1,-2,-3,-4,....

### 15. What are the types of operators supported by string?

1. Basic Operators.
2. Membership Operators.
3. Relational Operators.

### 16. Explain Replication Operation with an example?

Replication operator uses two parameters for operation. One is the integer value and the other one is the String. The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

#### Example:

```
>>> 5*"Vimal"
```

#### Output:

```
'VimalVimalVimalVimalVimal'
```

### 17. What is known as string slices?

A segment of a string is called a **slice**. String slice can be defined as substring which is the part of string. Therefore further substring can be obtained from a string. There can be many forms to slice a string. As string can be accessed or indexed from both the direction and hence string can also be sliced from both the direction that is left and right.

### 18. Explain boolean expressions with syntax?

A **boolean expression** is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise: True and False are special values that belong to the type bool; they are not strings:

```
>>> 5 == 5
```

```
True
>>> 5 == 6
False
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

## 19. What is an array?

An array is a group of similar data types stored under a common name. `int a[10]`; Here `a[10]` is an array with 10 values.

## 20. What are the main elements of an array declaration?

1. Array name
2. Type and
3. Size

## 21. Write the syntax for Chained Conditionals?

There are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

### Syntax

```
if condition:
    statement
elif condition:
    statement
else:
    statement
```

## 22. Define Parameter?

The function definition consists of the name of the function and a list of parameters enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters. In either case, the parentheses are required. Inside the function, the arguments are assigned to variables called **parameters**. The function definition consists of the name of the function and a list of parameters enclosed in parentheses. The parameter list may be empty, or it may contain any number of parameters. In either case, the parentheses are required. Inside the function, the arguments are assigned to variables called **parameters**.

## 23. What are the types of membership operators?

There are two types of Membership operators:

1) **in**: "in" operator return true if a character or the entire substring is present in the specified string, otherwise false.

2) **notin**: "not in" operator return true if a character or entire substring does not exist in the specified string, otherwise false.

## 24. Define `globals()` and `locals()` function?

The `globals()` and `locals()` functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If `locals()` is called from within a function, it will return all the names that can be accessed locally from that function.

If `globals()` is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the `keys()` function.

## 25. Explain the concept of Immutability with example?

Strings are **immutable**, which means you can't change an existing string. You can only create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
```

TypeError: 'str' object does not support item assignment

This error is because we are trying to replace the existing string, ie we are trying to replace 'H' at position `greeting[0]` with 'J' which is not possible in string.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

## **PART B**

1. What are Conditionals? Explain in detail.
2. Define Iteration. Briefly discuss iteration in detail.
3. What is meant by Fruitful functions. Explain with suitable examples.
4. Explain function composition in detail
5. Define recursion. Discuss its usage in programming.
6. What are Strings. Explain its methods in python
7. Define string slices and immutability. Explain briefly
8. Discuss in detail about the string functions and methods
9. Explain in detail about the string module in python
10. Discuss how Lists as arrays is implemented in python.
11. Illustrative programs: square root
12. Illustrative programs: gcd
13. Illustrative programs: exponentiation
14. Illustrative programs: sum an array of numbers
15. Illustrative programs: linear search
16. Illustrative programs: binary search.

## Exercises (Case study)

1. What does the program print?

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod
def a(x, y):
    x = x + 1
    return x * y
def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square
x = 1
y = x + 1
print(c(x, y+3, x+y))
```

2. The Ackermann function,  $A(m, n)$ , is defined:

$A(m, n) = n + 1$  if  $m = 0$

$A(m, 0) = 1$  if  $m > 0$  and  $n = 0$

$A(m, n) = A(m, A(m, n - 1))$  if  $m > 0$  and  $n > 0$ .

Write a function named `ack` that evaluates the Ackermann function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of  $m$  and  $n$ ?

3. A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]
def last(word):
    return word[-1]
def middle(word):
    return word[1:-1]
```

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `""` and contains no letters?

2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

4. A number,  $a$ , is a power of  $b$  if it is divisible by  $b$  and  $a/b$  is a power of  $b$ . Write a function called `is_power` that takes parameters  $a$  and  $b$  and returns `True` if  $a$  is a power of  $b$ . Note: you will have to think about the base case.

5. The greatest common divisor (GCD) of  $a$  and  $b$  is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is based on the observation that if  $r$  is the remainder when  $a$  is divided by  $b$ , then  $\text{gcd}(a, b) = \text{gcd}(b, r)$ . As a base case, we can use  $\text{gcd}(a, 0) = a$ . Write a function called `gcd` that takes parameters  $a$  and  $b$  and returns their greatest common divisor.

6. Fermat's Last Theorem says that there are no positive integers  $a$ ,  $b$ , and  $c$  such that  $a^n + b^n = c^n$  for any values of  $n$  greater than 2.

1. Write a function named `check_fermat` that takes four parameters— $a$ ,  $b$ ,  $c$  and  $n$ —and checks to see if Fermat's theorem holds. If  $n$  is greater than 2 and  $a^n + b^n = c^n$  the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. Write a function that prompts the user to input values for  $a$ ,  $b$ ,  $c$  and  $n$ , converts them to integers, and uses `check_fermat` to check whether they violate Fermat's theorem.

7. If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)

1. Write a function named `is_triangle` that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.

2. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses `is_triangle` to check whether sticks with the given lengths can form a triangle.

8. What is the output of the following program?

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)
recurse(3, 0)
```

1. What would happen if you called this function like this: `recurse(-1, 0)`?
2. Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).

9. Read the following function and see if you can figure out what it does.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

10. A string slice can take a third index that specifies the “step size”; that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` generates a reversed string. Use this idiom to write a one-line version of `is_palindrome` from Exercise 3.

11. The following functions are all intended to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False
def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'
def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag
```

```

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag
def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True

```

12. The built-in function `eval` takes a string and evaluates it using the Python interpreter. For example:

```

>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>

```

Write a function called `eval_loop` that iteratively prompts the user, takes the resulting input and evaluates it using `eval`, and prints the result. It should continue until the user enters 'done', and then return the value of the last expression it evaluated.



## Glossary

**floor division:** An operator, denoted `//`, that divides two numbers and rounds down (toward negative infinity) to an integer.

**modulus operator:** An operator, denoted with a percent sign (`%`), that works on integers and returns the remainder when one number is divided by another.

**boolean expression:** An expression whose value is either `True` or `False`.

**relational operator:** One of the operators that compares its operands: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

**logical operator:** One of the operators that combines boolean expressions: `and`, `or`, and `not`.

**conditional statement:** A statement that controls the flow of execution depending on some condition.

**condition:** The boolean expression in a conditional statement that determines which branch runs.

**compound statement:** A statement that consists of a header and a body. The header ends with a colon (`:`). The body is indented relative to the header.

**branch:** One of the alternative sequences of statements in a conditional statement.

**chained conditional:** A conditional statement with a series of alternative branches.

**nested conditional:** A conditional statement that appears in one of the branches of another conditional statement.

**return statement:** A statement that causes a function to end immediately and return to the caller.

**recursion:** The process of calling the function that is currently executing.

**base case:** A conditional branch in a recursive function that does not make a recursive call.

**infinite recursion:** A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

**temporary variable:** A variable used to store an intermediate value in a complex calculation.

**dead code:** Part of a program that can never run, often because it appears after a `return` statement.

**incremental development:** A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

**scaffolding:** Code that is used during program development but is not part of the final version.

**guardian:** A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

**object:** Something a variable can refer to. For now, you can use “object” and “value” interchangeably.

**sequence:** An ordered collection of values where each value is identified by an integer index.

**item:** One of the values in a sequence.

**index:** An integer value used to select an item in a sequence, such as a character in a string. In Python indices start from 0.

**slice:** A part of a string specified by a range of indices.

**empty string:** A string with no characters and length 0, represented by two quotation marks.

**immutable:** The property of a sequence whose items cannot be changed.

**traverse:** To iterate through the items in a sequence, performing a similar operation on each.

**search:** A pattern of traversal that stops when it finds what it is looking for.

**counter:** A variable used to count something, usually initialized to zero and then incremented.

**invocation:** A statement that calls a method.

**optional argument:** A function or method argument that is not required.

**reassignment:** Assigning a new value to a variable that already exists.

**update:** An assignment where the new value of the variable depends on the old.

**initialization:** An assignment that gives an initial value to a variable that will be updated.

**increment:** An update that increases the value of a variable (often by one).

**decrement:** An update that decreases the value of a variable.

**iteration:** Repeated execution of a set of statements using either a recursive function call or a loop.

**infinite loop:** A loop in which the terminating condition is never satisfied.