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

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Control Structures in Python

Everything you have seen so far has consisted of sequential execution, in which statements are always performed one after the next, in exactly the order specified.

But the world is often more complicated than that. Frequently, a program needs to

- skip over some statements,
- execute a series of statements repetitively, or
- choose between alternate sets of statements to execute.

That is where control structures come in. A control structure directs the order of execution of the statements in a program (referred to as the program's control flow).

Note:

There are three fundamental forms of statements(control) that a python language provides:

- sequential statements
- selection/conditional statements (if, if-else, nested-if, if-elif-else), and
- iterative statements (while, for)

In [1]:
```
# Sequential execution
x = 40
y = 20
m = x+y
n = x-y
o = x*y
p = x/y
print(m,n,o,p)
```

```
60 20 800 2.0
```

Selection/Conditional statements

In Python, conditional statements act depending on whether a given conditional expression is true or false.

You can execute different blocks of codes depending on the outcome of a conditional expression.

Conditional statements always evaluate to either True or False.

There are four types of conditional/selection statements.

- if statement
- if-else
- if-elif-else
- nested if-else

if statement

The simple if statement allows the program to branch based on the evaluation of an expression

If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and the controller moves to the next line.

Syntax of the if statement:

if expression:

```
    statement 1
    statement 2

    statement n
```

statement # this will be executed always

Flowchart of if:

Note 1:

Compound statements(span multiple lines), such as if, while, for, try, with, def, and class require a header line and a suite.

Header lines begin the statement with the keyword and terminate with a colon (:) and are followed by one or more lines which make up the suite. A group of individual statements, which make a single code block are called suites in Python.

Note 2:

The if, while and for statements implement traditional control flow constructs in python.

In [4]:
```
# Program to find the square of a number, only if number > 0
number = int(input('Enter any number:'))
if number > 0:
    print(number * number)  # Calculate square
print('Next lines of code')
```

```
Enter any number:4
16
Next lines of code
```

In [17]:
```
value = 5

threshold= 10
print("value is", value, "threshold is ",threshold)
if value > threshold :
    print(value, "is bigger than ", threshold)
```

```
value is 5 threshold is  10
```

In [6]:
```
x = 0
```

```
y = 5

if x < y:                               # Truthy
    print('yes1')

if y < x:                               # Falsy
    print('yes2')

if x:                                   # Falsy
    print('yes3')

if y:                                   # Truthy
    print('yes4')

if x or y:                              # Truthy
    print('yes5')

if x and y:                             # Falsy
    print('yes6')
```

```
yes1
yes4
yes5
```

In [7]:
```
if 'aul' in 'grault':                   # Truthy
    print('yes1')

if 'quux' in ['foo', 'bar', 'baz']:     # Falsy
    print('yes2')
```

```
yes1
```

In [8]:
```
# Nested simple if statements
if 'foo' in ['foo', 'bar', 'baz']:
    print('Outer condition is true')
    if 10 > 20:
        print('Inner condition 1')
    print('Between inner conditions')
    if 10 < 20:
        print('Inner condition 2')
    print('End of outer condition')

print('After outer condition')
```

```
Outer condition is true
Between inner conditions
Inner condition 2
End of outer condition
After outer condition
```

if – else statement

The if-else statement checks the conditional expression and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.

Syntax of the if-else statement:

if expression:

```
    statement(s)
```

else:

```
    statement(S)
```

If the conditional expression is True, if_block statement(s) will be executed. If the conditional expression is False, else_block statement(s) will be executed. See the following flowchart for more detail.

flowchart of if-else statement:

In [6]:
```
# Program to check whether a given number is odd or even
num=int(input('Enter any number:'))
if num % 2 == 0:
   print(num,"is even number")
else:
   print(num,"is odd number")
```

```
Enter any number:24
24 is even number
```

In [1]:
```
# Program to check whether an entered password is correct or incorrect

password = input('Enter your password: ')

if password == "welcome2022":
    print("Correct password")
else:
    print("Incorrect Password")
```

```
Enter your password: welcome22
Incorrect Password
```

Chain multiple if statement (if-elif-else)

In Python, the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.

With the help of if-elif-else we can make a tricky decision. The elif statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the if-elif-else statement:

if expression-1:

```
    statement 1
```

elif expression-2:

```
    stetement 2
```

elif expression-3:

```
    stetement 3

    ...
```

else:

statement

Calculate and print the student's grade, according to the following table:

In [ ]:
```
Weighted final score    Final grade
91 <= marks <= 100      S
81 <= marks <= 90           A
71 <= marks <= 80           B
61 <= marks <= 70           C
51 <= marks <= 60           D
41 <= marks <= 50           E
      marks <= 40           F
```

In [1]:
```
# Program to grade an entered subject marks (using if-elif-else)
marks = int(input('Enter marks of a subject(0-100):'))
if marks>=91 and marks<=100:
    print("Your Grade is S")
elif marks>=81 and marks<=90:
    print("Your Grade is A")
elif marks>=71 and marks<=80:
    print("Your Grade is B")
elif marks>=61 and marks<=70:
    print("Your Grade is C")
elif marks>=51 and marks<=60:
    print("Your Grade is D")
elif marks>=41 and marks<=50:
    print("Your Grade is E")
else:
    print("Fail Grade")
```

```
Enter marks of a subject(0-100):90
Your Grade is A
```

In [3]:
```
# Program to grade an entered subject marks (using if-elif-else and operator cascading)
marks = int(input('Enter marks of a subject(0-100):'))
if 91<=marks<=100:
    print("Your Grade is S")
elif 81<=marks<=90:
    print("Your Grade is A")
elif 71<=marks<=80:
    print("Your Grade is B")
elif 61<=marks<=70:
    print("Your Grade is C")
elif 51<=marks<=60:
    print("Your Grade is D")
elif 41<=marks<=50:
    print("Your Grade is E")
else:
    print("Fail Grade")
```

```
Enter marks of a subject(0-100):78
Your Grade is B
```

In [4]:
```
# Program to find the largest of three numbers (using if-elif-else and operator cascadin
a,b,c = 50,40,80
if b < a > c:
        print("a value is big")
elif a < b > c:
        print("b value is big")
else:
        print("c value is big")
```

```
c value is big
```

In [6]:
```
def user_check(choice):
```

```
    if choice == 1:
        print("Admin")
    elif choice == 2:
        print("Editor")
    elif choice == 3:
        print("Guest")
    #else:
        #print("Wrong entry")

user_check(1)
#user_check(2)
#user_check(3)
```

```
Admin
```

In [11]:
```
x = int(input("Please enter an integer: "))

if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

```
Please enter an integer: 1
Single
```

Nested if-else statement

In Python, the nested if-else statement is an if statement inside another if-else statement. It is allowed in Python to put any number of if statements in another if statement.

Indentation is the only way to differentiate the level of nesting. The nested if-else is useful when we want to make a series of decisions.

Syntax of the nested-if-else:

if conditon_outer:

```
    if condition_inner:
        statement of inner if
    else:
        statement of inner else
    statement of outer if
```

else:

```
    statement of outer else
```

statement outside nested if-else block

In [2]:
```
# Program to find the largest of three numbers
a,b,c = 10,20,30
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 value is big")
```

```
c value is big
```

Dangling Else Problem

The dangling else problem is an ambiguity of language interpretation.

The problem rarely occurs while we deal with the nested if-else statement. It is an ambiguity in which it is not clear, which if statement is associated with the else clause.

Consider a situation that there are two if statements and a single else statement. But it is not clear which if statement is associated with else statement.

A Solution to Dangling Else Problem

There are the following two ways to avoid dangling else problem:

- Try to design non-ambiguous programming.
- The dangling else problem can be resolved by proper indentation.

In [4]:
```
# Program to find the largest of four numbers
#a,b,c,d = 10,20,30,40
a,b,c,d = 10,50,30,40
if a > b:
    if a > c:
        if a > d:
            print("a value is big")
        else:
            print("d value is big")
    else:
        if c > d:
            print("c value is big")
        else:
            print("d value is big")
else:
    if b > c:
        if b > d:
            print("b value is big")
        else:
            print("d value is big")
    else:
        if c > d:
            print("c value is big")
        else:
            print("d value is big")
```

```
b value is big
```

How to implement switch statement in Python

The Pythonic way to implement switch statement is to use the powerful dictionary mappings, also known as associative arrays, that provide simple one-to-one key-value mappings.

Here's the Python implementation of the above switch statement. In the following example, we create a dictionary named switcher to store all the switch-like cases.

In [5]:
```
def switch_demo(key):
    switcher = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
        7: "July",
        8: "August",
        9: "September",
        10: "October",
        11: "November",
        12: "December"
    }
    #The get() method returns the value of the item with the specified key.
    #dictionary.get(keyname, value). value is optional. A value to return if the specifi
    print(switcher.get(key, "Invalid month"))

switch_demo(1)
switch_demo(13)
```

```
January
Invalid month
```

In the above example, when you pass an argument to the switch_demo function, it is looked up against the switcher dictionary mapping. If a match is found, the associated value is printed, else a default string ('Invalid Month') is printed. The default string helps implement the 'default case' of a switch statement.

Looping/Iteration

Iteration means executing the same block of code over and over, potentially many times. A programming structure that implements iteration is called a loop.

In programming, there are two types of iteration, indefinite and definite:

- With indefinite iteration, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.
- With definite iteration, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

while loop

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

While loop falls under the category of indefinite iteration sometimes. Indefinite iteration means that the number of times the loop is executed isn't specified explicitly in advance.

while loop syntax:

while expression:

```
    statement(s)
```

Flowchart of While Loop :

In [2]:
```
# Program to print multiplication table
n = int(input("Enter any integer : "))
i=1
while i<=10:
    print(n,'*',i,'=',i*n)
    i=i+1
```

```
Enter any integer : 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

In [4]:
```
# Program to find the sum of digits of a given positive integer number
n = int(input("Enter any integer : "))
sum=0
while n:
    sum=sum+n%10
    n=n//10
print("The sum of digits of a given number is ", sum)
```

```
Enter any integer : 12345
The sum of digits of a given number is  15
```

In [2]:
```
# Program to find the sum of digits of a given positive integer number
n = int(input("Enter any integer : "))
sum = 0
while n :
    sum = sum + n % 10
    n //= 10
print("The sum of digits of a given number is ", sum)
```

```
Enter any integer : 123456789
The sum of digits of a given number is  45
```

In [5]:
```
# Program to reverse a given number
n = int(input("Enter any integer : "))
rev=0
num=n
while n:
    rev = rev*10 + n % 10
    n //= 10
print("The reversed number is ", rev)
```

```
Enter any integer : 56789
The reversed number is  98765
```

In [2]:
```
# Program to reverse a given number
n = int(input("Enter any integer : "))
rev = 0
while n :
    rev = rev * 10 + n % 10
    n //= 10
print("The reversed number is ", rev)
```

```
Enter any integer : 3456
The reversed number is  6543
```

In [11]:
```
# Program to find the factorial of a given number
n = int(input("Enter any integer : "))
fact=1
```

```
num=n
while n > 1:
    fact=fact*n
    n=n-1
print("Factorial of",num,"is",fact)
```

```
Enter any integer : 5
Factorial of 5 is 120
```

In [8]:
```
# Program to find the sum of first n natural numbers
# sum = 1+2+3+...+n
n = int(input("Enter n: "))
sum=(n*(n+1))/2
print(int(sum))
```

```
Enter n: 10
55
```

In [34]:
```
# Program to find the sum of first n natural numbers
# sum = 1+2+3+...+n
n = int(input("Enter n: "))
sum=1
i=2
while i<=n:
    sum=sum+i
    i=i+1
print("The sum of first",n,"natural numbers is", sum)
```

```
Enter n: 10
The sum of first 10 natural numbers is 55
```

In [14]:
```
# Program to find the sum of first n natural numbers
# sum = 1+2+3+...+n

n = int(input("Enter n: "))
num=n
sum = 0
while n:
    sum = sum + n
    n = n-1

print("The sum of first",num,"natural numbers is", sum)
```

```
Enter n: 6
The sum of first 6 natural numbers is 21
```

In [39]:
```
# Print the pattern
# 1
# 12
# 123
# 1234
# 12345
n = int(input("Enter n: "))
i=1
while i<=n:
    j=1
    while j<=i:
        print(j,end='')
        j=j+1
    i=i+1
    print()
```

```
Enter n: 6
1
12
123
1234
```

```
12345
123456
```

In [43]:
```
# Printing first n fibonacci numbers
n = int(input("Enter n: "))
a=0
b=1
print(a,b,end=' ')
i=3
while i<=n:
    a,b=b,a+b
    i=i+1
    print(b,end=' ')
```

```
Enter n: 10
0 1 1 2 3 5 8 13 21 34
```

In [3]:
```
# Print the pattern
# 1
# 121
# 12321
# 1234321
# 123454321
n = int(input("Enter n: "))
i=1
while i<=n:
    j=1
    while j<=i:
        print(j,end='')
        j=j+1
    j=j-2
    while j>=1:
        print(j,end='')
        j=j-1
    i=i+1
    print()
```

```
Enter n: 5
1
121
12321
1234321
123454321
```

What will be the output of the following Python program?

In [10]:
```
i = 0
while i < 5:
    print(i,end=' ')
    i += 1
    if i == 3:
        break
else:
    print(0)
```

```
0 1 2
```

Explanation: The else part is not executed if control breaks out of the loop.

Note: Numeric Range Loop:

for i = 1 to 10

```
    loop body
```

This sort of for loop is used in the languages BASIC, Algol, and Pascal.

Three-Expression Loop

Another form of for loop popularized by the C programming language contains three parts:

for (i = 1; i <= 10; i++)

```
    loop body
```

An initialization, An expression specifying an ending condition, and An action to be performed at the end of each iteration.

The for loop (Collection-Based or Iterator-Based Loop)

This type of loop iterates over an iterable(collection of objects), rather than specifying numeric values or conditions:

```
for <var> in <iterable>:
    statement(s)
```

The statement(s) in the loop body are executed once for each item in iterable. The loop variable var takes on the value of the next element in iterable each time through the loop.

iterable is a collection of objects—for example, a list or tuple or set or dictionary or string or range() function. This type of for loop is arguably the most generalized and abstract.

What is iterable:

- an Iterable is a collection of elements physically or conceptually
- it has a builtin mechanism to give an element each time we ask
- has a builtin mechanism to signal when there are no more elements.

Semantics of the for statement:

1. start iterating through the iterable
2. get one element to the target variable
3. execute the suite or the body
4. repeat steps 2 and 3 until the iterable signals that it has no more elements
5. Exit the for loop and move to the next statement.

Here is a representative example:

In [23]:
```
list_of_weekdays = ['Monday','Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', '
for var in list_of_weekdays:    #iterable means an object can be used in iteration
    print(var)
```

```
Monday
Tuesday
Wednesday
Thursday
```

```
Friday
Saturday
Sunday
```

In [22]:
```
for ch in "Hello world":
    print(ch)
```

```
H
e
l
l
o

w
o
r
l
d
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

In [52]:
```
# Print all numbers from 0 to 5, and print a message when the loop has ended:
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

```
0
1
2
3
4
5
Finally finished!
```

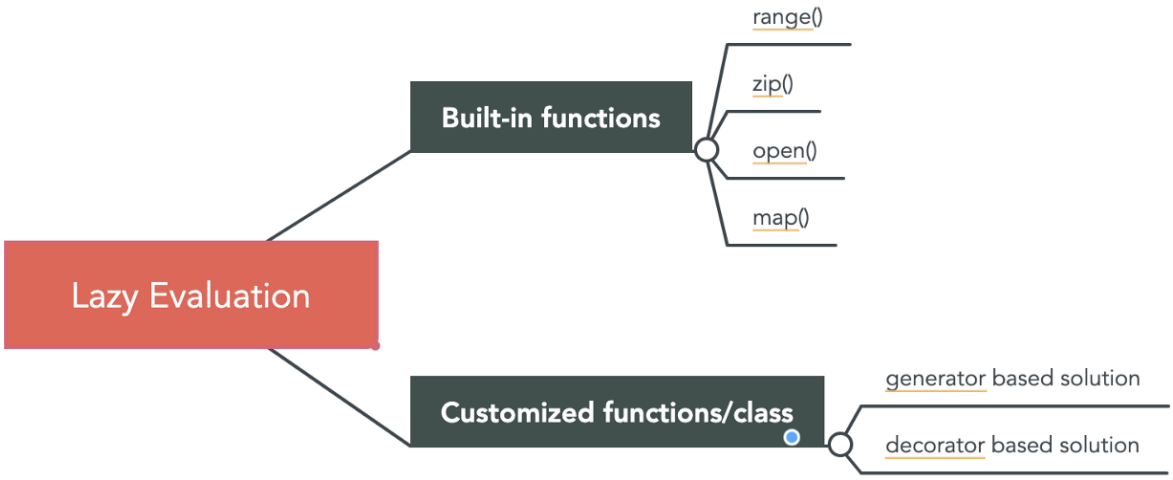
What is Lazy Evaluation in Python?

Lazy Evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed and which also avoids repeated evaluations. It's usually being considered as a strategy to optimize your code.

For example, you have a simple expression sum = 1 + 2. In this case, the evaluation is done immediately, therefore it has another name: Strict Evaluation.

On the other hand, we have a non-strict evaluation which is called Lazy Evaluation. The difference is that Lazy Evaluation will not immediately evaluate the expression but only does it when the outcome is needed.

But being lazy here is not necessarily a bad thing, it can improve the efficiency of your code and save plenty of resources. Luckily, Python has silently applied Lazy Evaluation to many built-in functions in order to optimize your code.



Python range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

range() constructor has two forms of definition:

- range(stop)
- range(start, stop[, step])

range() can take one argument, two arguments or three arguments.

range() Parameters:

- start - integer starting from which the sequence of integers is to be returned
- stop - integer before which the sequence of integers is to be returned. The range of integers ends at stop-1.
- step (Optional) - integer value which determines the increment between each integer in the sequence

In Python-2, range(5) would return a list of 5 elements. As the size of the list increases, more memory is used.

In [1]:
```
# In Python-2
>>> range(5)
[0, 1, 2, 3, 4]
>>> import sys
>>> sys.getsizeof(range(5))
112
>>> sys.getsizeof(range(500))
4072
```

Out[1]:
```
4072
```

The range method in Python-3 follows the concept of Lazy Evaluation. It saves the execution time for larger ranges and we never require all the values at a time, so it saves memory consumption as well.

In [5]:
```
# In Python-3
```

```
print(range(5))
```

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

However in Python 3, range(5) returns a range object which computes elements of the list on demand (lazy or deferred evaluation).

In [51]:
```
r=range(1,6)
print(r)
print(r[0],r[2],r[4])
print(type(r))
```

```
range(1, 6)
1 3 5
<class 'range'>
```

In [48]:
```
r=range(1,5)
for i in r:
    print(i,end=' ')
```

```
1 2 3 4
```

The range object can be iterated over to yield a sequence of numbers.

In [47]:
```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

In [5]:
```
import sys
sys.getsizeof(range(5))
```

Out[5]:
```
48
```

In [6]:
```
sys.getsizeof(range(500))
```

Out[6]:
```
48
```

No matter how big the range is, the object always has the same size. This is due to the fact that range(5) only stores the start, stop, step values, and calculates each item when it's needed.

Examples:

range() conceptually generates an arithmetic progression.

In [54]:
```
print(range(1, 10, 2))
```

```
range(1, 10, 2)
```

This output tells us that the result to call of range is an arithmetic progression with the initial value 1, the common difference 2 and the final value is one less than 10.

Observe that the sequence itself is not displayed. It conceptually holds all the elements. We can walk through it or we can convert it into a list as follows.

In [55]:
```
for i in range(1,10,2):
    print(i, end=' ')
```

```
1 3 5 7 9
```

In [56]:
```
print(list(range(1,10,2)))
```

```
[1, 3, 5, 7, 9]
```

In [24]:
```
for i in range(10):     # one argument
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

In [28]:
```
for i in range(1,21):    # two argument
    print(i, end=' ')
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

In [45]:
```
for i in range(1,21,2):    # three argument
    print(i, end=' ')
```

```
1 3 5 7 9 11 13 15 17 19
```

Experiment:

- can the step be 0
- can the step be negative?
- can these arguments be float?

In [ ]:
```
for i in range(1,10,0): #ValueError: range() arg 3 must not be zero
    print(i, end=' ')
```

In [46]:
```
for i in range(10,0,-2):
    print(i, end=' ')
```

```
10 8 6 4 2
```

In [20]:
```
for i in range(10,0,-1):
    print(i, end=' ')
```

```
10 9 8 7 6 5 4 3 2 1
```

In [11]:
```
for i in range(9,10,-1):    # try range(10,10,-1)  and range(11,10,-1):
    print(i, end=' ')
```

In [ ]:
```
for i in range(1,5,.1): #TypeError: 'float' object cannot be interpreted as an integer
    print(i, end=' ')
```

In [ ]:
```
for i in range(1.1,10):  #TypeError: 'float' object cannot be interpreted as an integer
    print(i, end=' ')
```

Examples on for-loop and range()

In [10]:
```
# Program to print multiplication table
n = int(input("Enter any integer : "))
for i in range(1,11):
    print(n,'*',i,'=',i*n)
```

```
Enter any integer : 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

In [11]:
```
# Program to find the factorial of a given number
n = int(input("Enter any integer : "))
fact=1
for i in range(2,n+1):
    fact=fact*i
print("Factorial of",n,"is",fact)
```

```
Enter any integer : 5
Factorial of 5 is 120
```

In [14]:
```
# Printing first n fibonacci numbers
n = int(input("Enter n: "))
a=0
b=1
print(a,b,end=' ')
for i in range(3,n+1):
    a,b=b,a+b
    print(b,end=' ')
```

```
Enter n: 10
0 1 1 2 3 5 8 13 21 34
```

In [12]:
```
# Program to find the sum of first n natural numbers
n = int(input("Enter n: "))
sum=1
for i in range(2,n+1):
    sum=sum+i
print(sum)
```

```
Enter n: 10
55
```

Python Program to read a number n and print the following pattern of the desired size using nested for loop.

- 1 2 3 4 5
- 1 2 3 4
- 1 2 3
- 1 2
- 1

In [1]:
```
n = int(input("Enter the desired size: "))
for i in range(n):
    for j in range(n-i):
          print(j+1,end=" ")
    print()
```

```
Enter the desired size: 5
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Python Program to read a number n and print the following pattern of the desired size using nested for loop.

- A B C D E
- A B C D
- A B C
- A B
- A

In [1]:
```
n = int(input("Enter the desired size: "))
for i in range(0,n):
    a=65
    for j in range(n-i):
          print(chr(a+j),end=" ")
    print()
```

```
Enter the desired size: 5
A B C D E
A B C D
A B C
A B
A
```

In [14]:
```
# What will be the output of the following Python code?

x = 'abcd'
for i in range(len(x)):
    print(i,end=' ')
```

```
0 1 2 3
```

Examples to illustrate when to use the while loop and when to use the for loop:

a) Program to display squares of numbers from 1 to n.

In [23]:
```
#using while
n = int(input("Enter an integer number: "))
i = 1
while i <= n :
    print("Square of ", i, " is ", i * i)
    i += 1
```

```
Enter an integer number: 5
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
```

In [22]:
```
#using for loop
n = int(input("Enter an integer number : "))
for i in range(1, n + 1):
    print("Square of ", i, " is ", i * i)
```

```
Enter an integer number : 5
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
```

Output for both the programs are the same, but in this scenario, since the number of iterations are known at the beginning itself, it is better to use for-loop since we do not need to take into consideration the initialization of the loop variable and updation and is done implicitly. Thus in this scenario, use of for is better than use of while.

b) Program to display all squares less than or equal to n.

Since we cannot easily determine the number of iterations. We should use a while loop here.

In [31]:
```
#using while
n = int(input("enter an integer : "))
i = 1
```

```
while i * i <= n :
    print("Square of ", i, " is ", i * i)
    i += 1
```

```
enter an integer : 50
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
Square of  6  is  36
Square of  7  is  49
```

In [15]:
```
#using for
n = int(input("enter an integer : "))
for i in range(n):
    #print(i)
    if (i*i) < n:
        print("Square of ", i, " is ", i * i)
    #else:
        #break
```

```
enter an integer : 50
Square of  0  is  0
Square of  1  is  1
Square of  2  is  4
Square of  3  is  9
Square of  4  is  16
Square of  5  is  25
Square of  6  is  36
Square of  7  is  49
```

This results in the same outcome, but the number of times the loop runs is always going to be n. whereas when we use while it stop as soon as the square reaches the value of n.

c) Program to find all factors of a given number say n

In [32]:
```
n = int(input("enter a number : "))
for i in range(1, n + 1):
    if n % i == 0 :
        print(i, end = " ")
print()
```

```
enter a number : 15
1 3 5 15
```

In [21]:
```
#using while-loop (a better approach)
n = int(input("enter a number : "))
i = 1
while i * i < n :
    if n % i == 0 :
        print(i, n // i, end = " ")
    i = i + 1
else:
    if i * i == n :
        print(i, end = " ")
```

```
enter a number : 10
1 10 2 5
```

iterate the above code manually to calculate the number of times the body of the loop executes

Infinite loops

An infinite loop is an iterative control structure that never terminates (or eventually terminates with a system error).

Infinite loops are generally the result of programming errors.

For example, if the condition of a while loop can never be false, an infinite loop will result when executed.

In [ ]:
```
i = 1
while i < 6:
  print(i,end='')
  #i += 1
print('Statement after while-loop')
```

In [ ]:
```
a = 10; b = 5
i = 0
while b != a :
    i =i+1
    a =a-1
    b =b-1
    print(i,end=' ')
```

In [ ]:
```
while(True):
    print('Hello', end='')
```

Interruption of Loop Iteration

The break Statement

The execution of break statement inside a loop immediately terminates a loop entirely. The program execution proceeds to the first statement following the loop body.

In [1]:
```
my_list = ['Sita', 'Ram', 'Guru', 'Daksh', 'Riya', 'Raj']

for i in range(len(my_list)):
    print(my_list[i])
    if my_list[i] == 'Daksh':
        break
        print('After break statement')

print('Statement after while-loop')
```

```
Sita
Ram
Guru
Daksh
Statement after while-loop
```

In [12]:
```
#Example: Exit the loop when i is 3:
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
print('Statement after while-loop')
```

```
1
2
3
Statement after while-loop
```

Break Statement inside nested loops

In [2]:
```
for i in range(1,5):
    for j in range(1,5):
        if j==3:
            break
        print("The number is ",i,j);
```

```
The number is  1 1
The number is  1 2
The number is  2 1
The number is  2 2
The number is  3 1
The number is  3 2
The number is  4 1
The number is  4 2
```

So because of the break statement, the second for-loop will never iterate for 3 and 4.

The break statement takes care of terminating the loop in which it is used. If the break statement is used inside nested loops, the current loop is terminated, and the flow will continue with the code followed that comes after the loop.

The continue Statement

With the execution of continue statement inside a loop, we can stop the current iteration remaining statements execution, and continue with the next iteration.

Python break and continue are used inside the loop to change the flow of the loop from its normal procedure.

In [1]:
```
i = 0
while i < 6:
  i += 1
  if i == 3:  # # If i is equals to 3,continue to next iteration without printing
    continue
  print(i)
```

```
1
2
4
5
6
```

The continue statement skips the code that comes after it, and the control is passed back to the start for the next iteration.

Using else statement with while-loop

Python supports to have an else statement associated with a loop statement.

If the else statement is used with a while loop, the else statement is executed only when the condition becomes false.

Example: Print a message once the condition is false:

In [8]:
```
i = 1
while i < 6:
  print(i)
```

```
  i += 1
else:
  print("i is no longer less than 6")
```

```
1
2
3
4
5
i is no longer less than 6
```

In [4]:
```
i = 1
while i < 6:
  print(i)
  i += 1
  if(i==4):
    break
else:
  print("i is no longer less than 6")
```

```
1
2
3
```

Pass statement

As the name suggests pass statement simply does nothing.

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

It is like null operation, as nothing will happen when pass is executed. Pass statement can also be used for writing empty loops.

Pass is also used for empty control-statement, function and classes.

In [1]:
```
s="Hello"
# Empty loop
for i in s:
            #IndentationError: expected an indented block

# Empty function
def fun():
    pass

fun()
```

```
  Input In [1]
    def fun():
    ^
IndentationError: expected an indented block
```

In [6]:
```
s="Hello"
# Empty loop
for i in s:
    pass

# Empty function
def fun():
    pass

fun()
```

Python Pass Statement is used as a placeholder inside loops, functions, class, if-statement that is meant to be implemented later.

Python pass is a null statement. When the execution starts and the interpreter comes across the pass statement, it does nothing and is ignored.

The None value

In Python, None is a special value which means “nothing”. Its type is called NoneType, and only one None value exists at a time – all the None values we use are actually the same object:

print(None is None) # True

None evaluates to False in boolean expressions.

In [1]:
```
print(None is None)  # True
```

```
True
```

In [2]:
```
type(None)
```

Out[2]:
```
NoneType
```

Answer to exercise: Valid or invalid

In [7]:
```
x=10
y=10
a=10
name='James'

if (x > 4):
    pass
if x == 2 :
    pass
if (y =< 4):
    pass
if (y = 5) :
    pass
if (3 <= a):
    pass
if (1 - 1) :
    pass
if ((1 - 1) <= 0):
    pass
if (name == "James"):
    pass
```

1. if (x > 4) – valid
2. if x == 2 – valid (brackets are not compulsory)
3. if (y =< 4) – invalid (=< is not a valid operator; it should be <=)
4. if (y = 5) – invalid (= is the assignment operator, not a comparison operator)
5. if (3 <= a) – valid
6. if (1 - 1) – valid (1 - 1 evaluates to zero, which is false)
7. if ((1 - 1) <= 0) – valid
8. if (name == "James") – valid

In [27]:
```
y=10
if (y = 5) :
    pass
```

Definite and Indefenite loops

They differ in the way in which the number of iterations is determined:

In definite loops, the number of iterations is known before we start the execution of the body of the loop.

A definite loop is a loop in which the number of times it is going to execute is known in advance before entering the loop

In indefinite loops, the number of iterations is not known before we start to execute the body of the loop, but depends on when a certain condition becomes true (and this depends on what happens in the body of the loop)

In an indefinite loop, the number of times it is going to execute is not known in advance. Typically, an indefinite loop is going to be executed until some condition is satisfied.

In [29]:
```
# Definite loop
n=15
while(n>0):
    print(n,end=' ')
    n=n-1
```

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

In [16]:
```
# Indefinite loop

checking_acc = 5678143
acc_num = int(input("Enter the account number:"))

while checking_acc != acc_num:
    print("Wrong account number ")
    acc_num = int(input("Enter the right account number:"))

print("Your account number is" , acc_num)
```

```
Enter the account number:11111111111111
Wrong account number
Enter the right account number:5678143
Your account number is 5678143
```

do-while loop Construction in Python

Python doesn't have a do-while loop. But we can create a program to implement do-while. It is used to check conditions after executing the statement. It is like a while loop but it is executed at least once.

In [2]:
```
i = 1
while True:
    print(i)
    i = i + 1
    if(i > 5):
        break
```

```
1
2
3
```

```
4
5
```

In [1]:
```
print(int(56.99))
```

```
56
```

In [2]:
```
# Generate the following pattern for n = 4
# 1 = 1
# 1 + 2 = 3
# 1 + 2 + 3 = 6
# 1 + 2 + 3 + 4 = 10
# m = int(input("enter an integer : "))

m = 4
s = 0
for n in range(1, m + 1):
        print("{0:3}".format(1), end = " ")
        for i in range(2, n + 1):
                print(" +", "{0:3}".format(i), end = " ")
        s += n
        print(" =", "{0:3}".format(s))
```

```
enter an integer : 5
  1  =   1
  1  +   2  =   3
  1  +   2  +   3  =   6
  1  +   2  +   3  +   4  =  10
  1  +   2  +   3  +   4  +   5  =  15
```

In [6]:
```
n = int(input("Enter 3-digit number : "))
import math
num=n
sum = 0
while n :
    sum = sum + math.factorial(n % 10)
    n //= 10

if num == sum:
    print("The given number", num, 'is a strong number')
else:
    print("The given number", num, 'is not a strong number')
```

```
Enter 3-digit number : 145
The given number 145 is a strong number
```

References:

- https://ucsbcarpentry.github.io/2020-08-19-Summer-Python/03-control-structures/index.html
- https://realpython.com/python-conditional-statements/