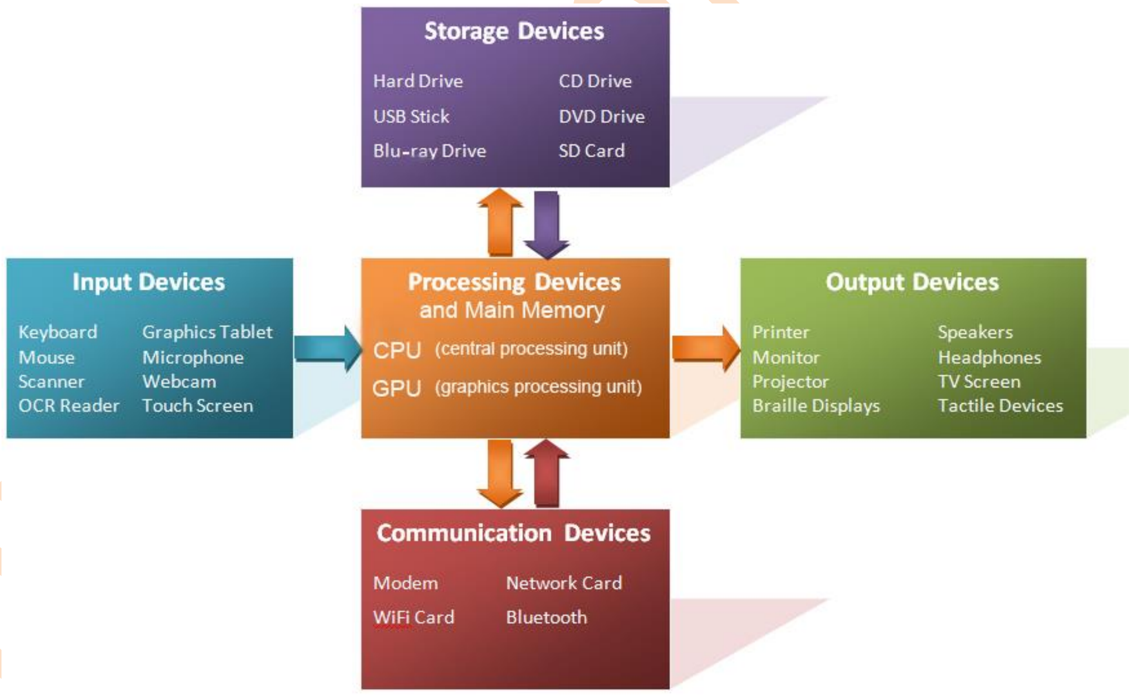
Hardware:

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the **central processing unit** (CPU) and **main memory**. It also includes **peripheral components** such as a keyboard, monitor, mouse, printer etc.



Central processing unit (CPU) – the "brain" of a computer system. Interprets and executes instructions.

Main memory – is where currently executing programs reside.It is *volatile*, the contents are lost when the power is turned off.

Secondary memory – provides long-term storage of programs and data. N*onvolatile*, the contents are retained when power is turned off. Can be magnetic (hard drive), optical (CD or DVD), or flash memory (USB drive).

Input/output devices – mouse, keyboard, monitor, printer**,** etc.

Buses – **transfer data between components within a computer system**. System bus (between CPU and main memory).

Software:

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer. This can be in the form of instructions on paper, or in digital form.

While some software are intrinsic to a computer system called as **system software**(System software is general purpose software which is used to operate computer hardware. It provides platform to run application softwares), while other softwares fulfills users' needs, called as **application software**(Application software is specific purpose software which is used by user for performing specific task).

1.1 Difference between System Software and Application Software

S.No.	System Software	Application Software
1.	System software is used for operating computer hardware.	Application software is used by user to perform specific task.
2.	System softwares are installed on the computer when operating system is installed.	Application softwares are installed according to user's requirements.
3.	In general, the user does not interact with system software because it works in the background.	In general, the user interacts with application sofwares.
4.	System software can run independently. It provides platform for running application softwares.	Application software can't run independently. They can't run without the presence of system software.
5.	Some examples of system softwares are compiler,	Some examples of application softwares are word processor,

	assembler, debugger, driver, etc.	web browser, media player, etc.

Syntax and Semantics :

Syntax and semantics are important concepts that apply to all languages.

The syntax of a language is a set of characters and the acceptable arrangements (sequences) of those characters.

For example, English includes the letters of the alphabet, punctuation, and properly spelled words and properly punctuated sentences.

The following is a syntactically correct sentence in English,

"Coronavirus disease (COVID-19) is an infectious disease caused by a newly discovered coronavirus."

However, the following, is not syntactically correct.

"Coronavirus disease (COVID-19) is an infectious disease csuaed by a newly discovered coronavirus."

In the above sentence, the sequence of letters **"csuaed"** is not a word in the English language.

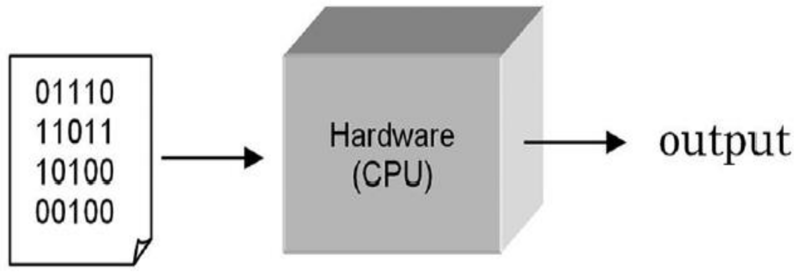
Lets consider the following sentence,

"disease newly an is infectious COVID-19 by a coronavirus discovered coronavirus."

This sentence is syntactically correct, but is semantically incorrect, as there is no meaning.

Program Translation:

A central processing unit (CPU) is designed to interpret and execute a specific set of instructions represented in binary form (i.e., 1s and 0s) called machine code. Only programs in machine code can be executed by a CPU**.**

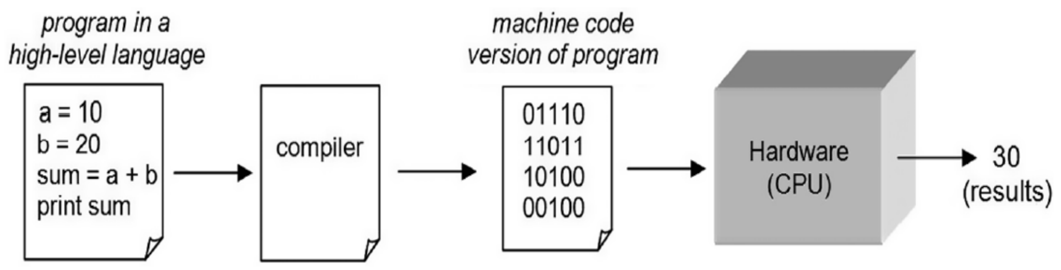


Writing programs at this “low level” is tedious and error-prone. Therefore, most programs are written in a “high-level” programming language such as Python. Since the instructions of such programs are not in machine code that a CPU can execute, a translator program must be used.

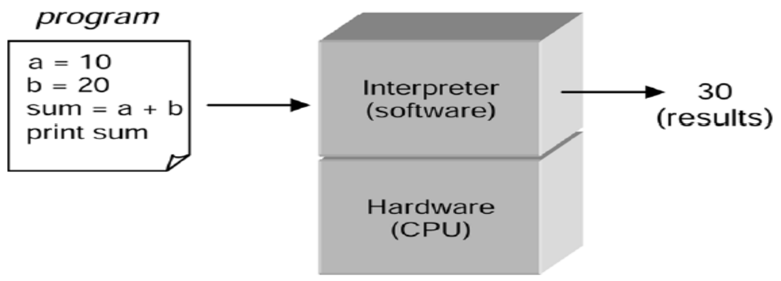
There are two fundamental types of translators:

- Compiler :translates the programs into machine code executed by the CPU.
- Interpreter executes instructions in place of the CPU.

Compiler:



Interpreter



An interpreter can immediately execute instructions as they are entered. This is referred to as interactive mode. This is a very useful feature for program development. Python is executed by an interpreter.

Difference between compiler and interpreter

Sl. No	Compiler	Interpreter
1	Translates high-level languages into machine code	Temporarily executes high-level languages, one statement at a time
2	An executable file of machine code is produced (object code	No executable file of machine code is produced (no object code)
3	Compiled programs no longer need the compiler	Interpreted programs cannot be used without the interpreter
4	Error report produced once entire program is compiled	Error message produced immediately (and program stops at that point)
5	Compiling may be slow, but the resulting program code will run quick (directly on the processor)	Interpreted code is run through the interpreter (IDE), so it may be slow, e.g. to execute program loops

Program structure:

Points to Note:

- A program in python has number of statements. These statements are followed by the computer one after another in a sequence unless there are special constructs.
- Python is case sensitive. 'A' and 'a' are different.
- Language has grammar rules called syntax. If the syntax is violated, the Python translator gives an error.

Example 1:

print("hello') # throws an error because of different quotes(single and double).

Example 2: All statements unless special construct should start from the first column.

```
>>>    print("hello")    # as there is a space after python terminal.
        File "<stdin>", line 1
        print("hello")
        ^
IndentationError: unexpected indent
```

Correct one!!!

```
>>> print("hello")
    hello
```

- If a program has no syntax error, then Python will execute the statements one by one. Sometimes we may get errors at run time. Then the program stops at that point and indicates the error. This is checked only when that point is reached during the program execution.

  Example 3:

```
# program written in batch mode . save it as filename.py .

print(1)
print("hello")
print(10/0)
```

  output:

```
1
hello
Traceback (most recent call last):
  File "p.py", line 3, in <module>
    print(10/0)
ZeroDivisionError: division by zero
```

- Each statement by default (normally) should be on a line by itself.
  - We cannot normally have more than one statement on a single line
  - We cannot normally continue the statement on more than one line.

- To continue statement on multiple lines, either we use constructs with delimiters for the beginning and the end or we use special key - \ - to ignore the enter key – we escape the newline.

# 1. use constructs which has beginning and ending markers - like a pair of parentheses

```
print(
"five"
)
```

# 2. enter \ (backslash) before pressing <Enter> key - this is called escaping.

```
print \
(
"six"
)
```

- We can have multiple statements on a single line by using ; as a separator between statements.

  # multiple statements on a single line

  Example:

  ```
  print("seven"); print("eight")
  ```

- In many languages, the execution starts from a special function called main. We have no such concept in Python. The execution starts from the first statement on top.

- A function name itself has a value. Any thing which has a value is called an expression. If we enter the function name on one line, it becomes a statement. But function will not be called unless we also use a pair of parentheses.

  print # fn name => expr => stmt; no call !!

  ("no output") # string within parentheses is an expr ; no action

Varibles

concept of constant

In arithmetic, we talk about values which do not change. Even in programming, we talk about entities which do not change. They are considered as they are. They are said to be constants or literals.

A literal is a sequence of one or more characters that stands for itself(constants).

Literals can be :

Numeric literal

String Literal

Numeric Literals: Integer

Floating

Complex Numbers

- A numeric literal is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point.
- If a numeric literal contains a decimal point, then it denotes a floating-point value , or “ float ” (e.g., 10.24); otherwise, it denotes an integer value (e.g., 10). **Commas are never used in numeric literals** .
- Complex numbers have a real and imaginary part.
- There is no limit to the size of an integer that can be represented in Python.

- Floating-point values,however, have both a limited range and a limited precision . Python uses a double-precision standard providing a range of 10 to the power of -308 to 10 to the power of 308 with 16 to 17 digits of precision.
- To denote such a range of values, floating-points can be represented in scientific notation, **9.0045602e15 ,1.006249505236801e8**

- Limitations of floating-point representation.

For example, the multiplication of two values may result in **arithmetic overflow** , a condition that occurs when a calculated result is too large in magnitude (size) to be represented.

```
>>> 1.5e200 * 2.0e210
    inf
```

Note: inf- infinity

Similarly, the division of two numbers may result in arithmetic underflow , a condition that occurs when acalculated result is too small in magnitude to be represented.

```
>>> 1.0e-230 / 1.0e100
    0.0
```

Valid Literals examples:

```
>>> 1024   # (valid)

1024

>>> -1024  # (valid)

-1024

>>> .1024  # (valid)

0.1024

>>> 0.1024 # (valid)

0.1024

>>> 1,024.7  # (valid)

(1, 24.7)

>>> 1,024     # invalid

  File "<stdin>", line 1

    1,024

       ^

SyntaxError: invalid token
```

```
>>> 3+4j  #valid

(3+4j)

>>> 3+4i  #invalid

  File "<stdin>", line 1

    3+4i

       ^

SyntaxError: invalid syntax
```

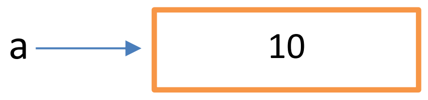
A string literal, or string, is a sequence of characters denoted by a pair of matching single or double (and sometimes triple) quotes in Python.

```
a=”python”

b=”PES University”
```

Variable:

A variable is a name (identifier) that is associated with a value.

A variable is created the moment you first assign a value to it.



A variable can be assigned different values during a program's execution—hence, the name "variable" .

Every variable in Python is a reference (a pointer) to an object and not the actual value itself. For example, the assignment statement just adds a new reference to the right-hand side.

Note:

There are some reserved words for Python and can not be used as variable name. (Keywords)

Variables are assigned values by use of the assignment operator , '=' .

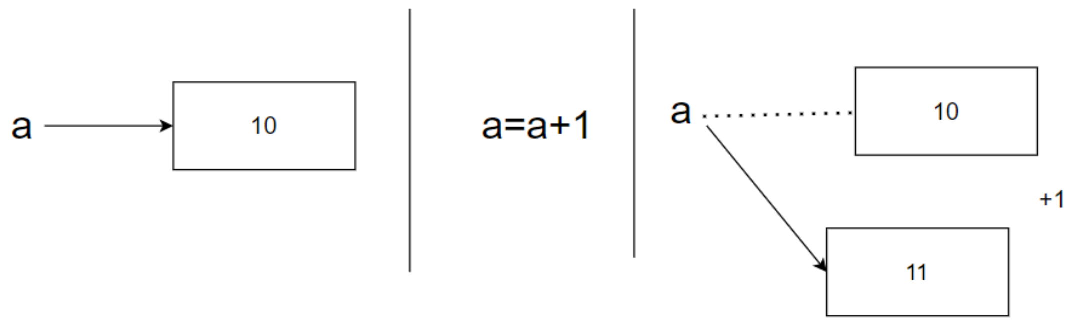
a=10

a=10+2

Mathematically,

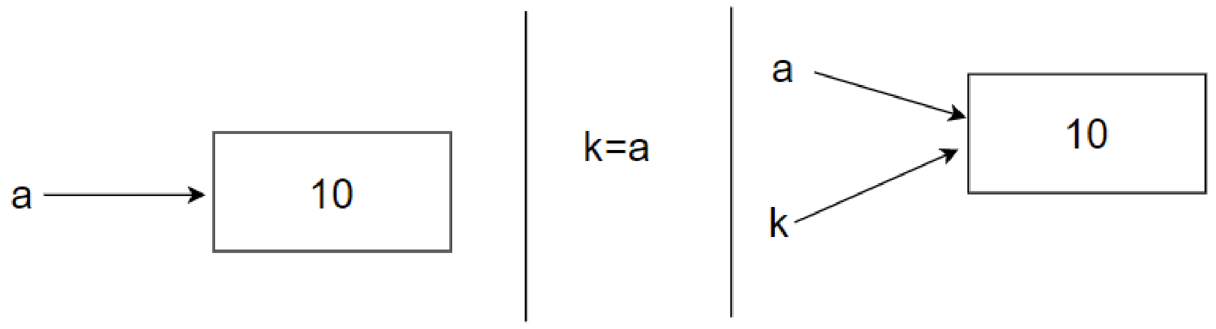
a =a + 1 does not make sense. However, in programming it is used to increment the value of a given variable by one.

It is more appropriate, therefore, to think of the '=' symbol as an arrow symbol, as show



From the above example it makes clear that the right side of an assignment is evaluated first, then the result is assigned to the variable on the left.

Variables may also be assigned to the value of another variable (or expression)



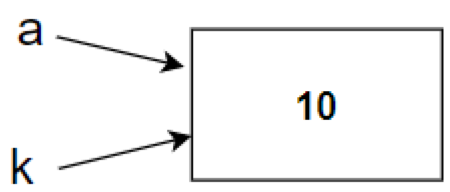
Before **After**

Variables **a** and **k** are both associated with the same literal value 10 in memory. One way to see this is by use of **built-in function id,**

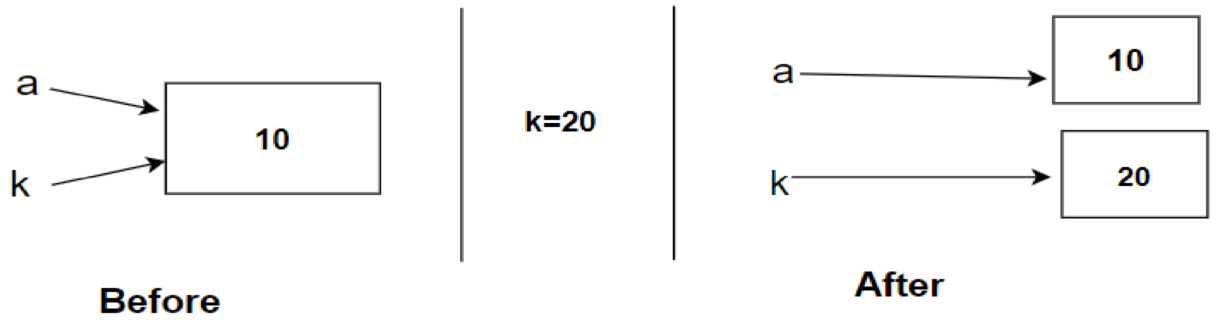
```
>>>id(a)
        505494040(value may change)
>>>id(k)
        505494040
```

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems.

If the value of **a** changed, would variable **k** change along with it?



This cannot happen in this case because the variables refer to integer values, and integer values are immutable. An immutable value is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned,



If no other variable references the memory location of the original value, the memory location is deallocated (that is, it is made available for reuse).

Identifiers:

- An identifier is a sequence of one or more characters used to provide a name for a given program element.
- Python is case sensitive.

  Ex: Line is different from line.

- Identifiers may contain letters and digits, but cannot

  begin with a digit.

- The underscore character, _, is also allowed to aid in the readability of long identifier names.
- It should not be used as the first character, however, as identifiers beginning with an underscore have special meaning in Python.

Keywords

A keyword is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as “regular” identifiers. Doing so will result in a syntax error.

Ex:

```
>>> and=10
  File "<stdin>", line 1
```

```
   and=10
     ^
SyntaxError: invalid syntax
```

Note: To display the keywords, type help() in the Python shell, and then type keywords.

>>>help()
help> keywords.

Type function

type is a built in function ,which returns type of the given object.Type of a variable depends on the value assigned to it.

```
a = 10
print(type(a))  # int

a = 10.0
print(type(a)) # float

a = "python"
print(type(a)) # str


a = True
print(type(a))  # values of bool type : False True




a = 2 + 3j
print(type(a)) # complex
```

The above ones are simple types also called as scalar as they have a single value.

The below ones are called as structured or reference types:having more than one value.

```
a = (10, 20, 30, 40)
print(type(a)) # tuple


a = [10, 20, 30, 40]
print(type(a)) # list


a = {10, 20, 30, 40}
print(type(a)) # set


a = {1:20,2:30}
print(type(a)) # dict
```

Output:

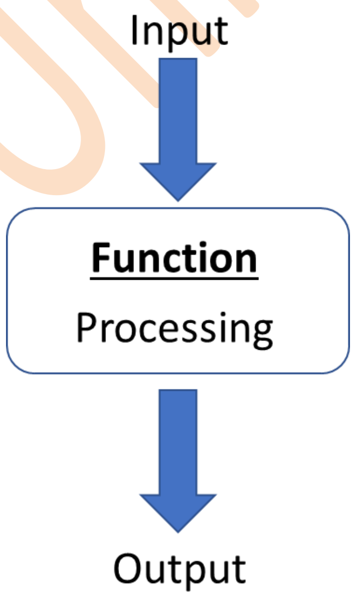
When we write programs using the batch mode of programming, we need to explicitly tell the interpreter what should be displayed as a result of the program.

Python has a specialised function that does just that, called the print function.

we use the function **print** for displaying information to the output screen.

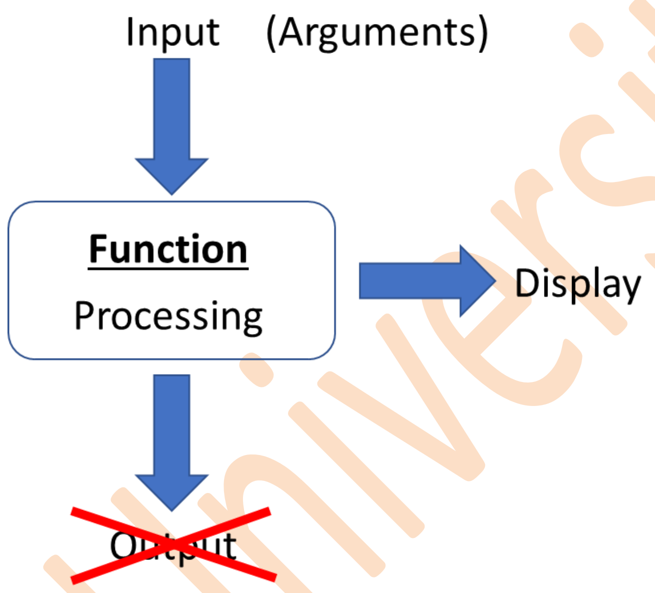
A function is a group of commands that perform a specified task.

Functions may take some input, do some sort of processing and returns a result(output)



In programming, inputs given to a function are called arguments.

Print is a function that takes input and as part of the processing, displays the value of the argument to the output screen and does not return any output to the program



As print is a function, we should call or invoke using a function call operator, which is a pair of parentheses "()" following the name "print".

Syntax:

print(...)

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

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

Characteristics:

a) can take argument of any type

Values could be : Numbers, Boolean, strings, collections, expressions, functions, etc

```
>>> print(100) #number

100

>>> print(2.5)

2.5

>>> print(True) #boolean

True

>>> print(“Hello”) #string

Hello

>>> print([1,2,3]) #collection: List

[1,2,3]
```

b) can take any number of arguments

```
print(1) #output  1

print(1,2) #output  1 2

print(1,2,3) #output  1 2 3

print(1, 2, 3, 4) #output  1 2 3 4
```

c) each argument is evaluated as an expression

When we give and expression as an argument to the print function, the Expression is evaluated and the result is displayed

```
>>> print(10+10) # expression

20
```

d) In the output we observe a space between two values.

By default, output field separator is a space.

We can change the behavior.

Code 1:

```
print("python", "for ", "computational ")

# output:  python for  computational
```

print("problem ", "solving")# output: problem solving

Code 2: (sep field behavior changed)

print("python", "for ", "computational ",sep='')**

# output: pythonfor **computational**

print("problem ", "solving")# output: problem solving

NOTE: the sep field behaviour is only applicable the the function in which it is specified. The subsequent function calls will default to space as the seperator

e) After each print, we get a newline. This is called the output record separator.

This can be changed by specifying end = <val> in print**.**

Code 3:

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

print("for", end = "")

print("All")

Output: Python for All
```

Code 4:

```
sep = " is special" #this is a variable

print("python ", "programming language")
```

print("python ", "programming language",sep = "powerful") # does not create a variable called sep, it is the argument of the print function

print(sep) # output: is special

since the variables are used in different contexts

Try with different possibilities.

Taking User input:

input ()

Write a program to find the area of a rectangle.

Code 1:

```
length=10

breadth=20

area=length*breadth

print("area=",area)
```

output:

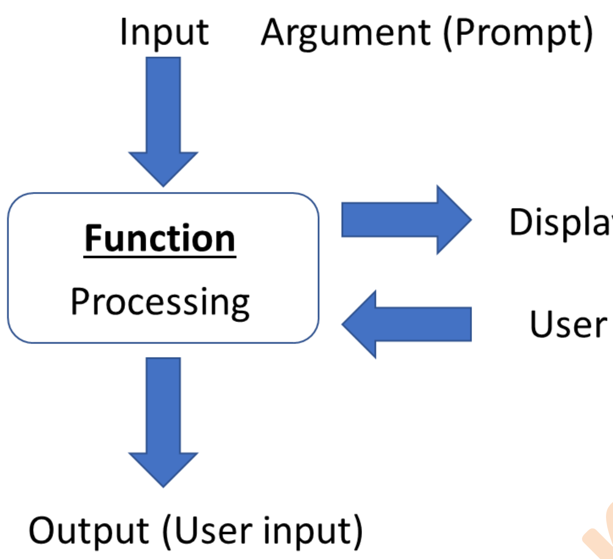
```
area=200
```

NOTE:

- The values of variables length and breadth are hard coded in the program itself.
- To allow flexibility we might want to take the input from the user.
- In Python, we have the input() function to allow this.
- The syntax for input() is:

  # input([prompt])

  #where prompt is the string we wish to display on the screen. It is optional.



- input() : gets a string from the keyboard
- int(input()) converts a string to an int

Let's try and write a better version of the previous program to allow the user to enter the dimensions of the rectangle.

Code 2:

```
length = int(input())

breadth = int(input())

a = length*breadth

print("area : ", a)
```

input/output

```
10

20

area :  200
```

The above program does take input, but the user does not know what the program expects as an input and thereby makin this program ambiguous

code 3:

We can make it more readable by allowing the user to specify the required string as prompt

```
length = int(input("enter the length value"))

breadth = int(input("enter the breadth value"))

a =length*breadth

print("area : ", a)
```

input/output

```
enter the length value 10

enter the breadth value 20

area :  200
```

Department of Computer Science and Engineering

PES UNIVERSITY

UE20CS101 : Python for Computational Problem Solving (4-0-0-4-4)

Text Book: “Introduction to Computer Science Using Python: A Computational Problem-Solving Focus” Charles Dierbach, Wiley India Edition, John Wiley,2015.

Input Function:

```
#user input


#type conversion
#b = int('two')
#print("b : ",b, type(b))


a = input("Enter a number: ")
print("a : ",a, type(a))      #type - str : input always returns a str value


# input function treats all inputs the same, it is up to the programmer to
change the type and then use the value accordingly


a = int(input("Enter a number: "))
print("a : ",a, type(a))


print()
```

```
a = '10'

int(a)

print("a : ",a, type(a))


a = float(input("Enter a number: "))

print("a : ",a, type(a))
```

Operators

Datatypes categorize information and the operations that can be performed on them.

In this chapter we explore the operators that are used on data in the process of computational problem solving.

An operator is a symbol that denotes and operation that is performed on data

Data that is used with operators is called an operand.

A combination of operators and operands can also be called an **expression**

What is an Expression?

An expression has a value.

All the following are expressions.

- constant

    example : 1234   3.14 "python "

- variable

    a = 6

  a is an expression

Operators have Arity or Rank that determines how many operands it must be used with.

A **unary operator** operates on only one operand, such as the negation operator in the expression: **- 12**

A **binary operator** operates on two operands, as with the addition operator: **2+3**

- expression binary_operator expression

    3 + 4

- unary operator expression

    -5

- expression within parentheses

    (3 + 4)

Please note the following.

- An expression has a value

- A statement does not

- An expression is also a statement

- A statement is not necessarily an expression

This is an assignment statement. This is not an expression since it doesn't have a value

a = 10

#print(a = 10) # error

operators:

arithmetic operators:

Operator	Expression	Name
-	-x	Negation
+	x + y	Addition
-	x - y	Subtraction
*	x * y	Multiplication
**	x ** y	Exponentiation
/	x / y	Division
//	x // y	Truncation Division
%	x % y	Modulus

Examples:

```
>>> 25 / 4

6.25

>>> 25 // 4

6

>>> 25 % 4

1

>>> 25 ** 3

15625
```

Python provides two forms of division:

- **"True" division** is denoted by a single slash, **/**

Thus, **34/10 evaluates to 3.4**

- **Truncating division** is denoted by a double slash, **//** providing a truncated result based on the type of operands applied to

Thus, **34//10 evaluates to 3**

When using truncating division, we must note that:

- **When both operands are integer values, the result is a truncated integer** referred to as **integer division**.
    - **34//10 evaluates to 3**

- When as least one of the operands is a float type, the result is a **truncated floating point**.
    - **34.0//10 evaluates to 3.0**

Operator Precedence:

When we have an expression that contains more than one operator and are different operators, how do we decide in which order to evaluate the expression?

Lets consider the following example,

4+3*5

There are two possible ways in which it can be evaluated

4 + 3 * 5 → 4 + 15 → **19**

Or

4 + 3 * 5 → 7 * 5 → **35**

To ensure that the results will be the same at all times, we use the rules of operator precedence.

It helps the interpreter determine the order in which the expression must be evaluated

As another example,

4 + 2 ** 5 // 10 → 4 + 32 // 10 → 4 + 3 → **7**

Operator precedence guarantees a consistent interpretation of expressions

It is good programming practice to use parentheses even when not needed

4 + (2 ** 5) // 10

Operator Precendence table

1. ** (exponentiation)
2. - (negation)
3. * (multiplication) , / (division) , // (truncation divison) , % (division)
4. + (addition) , - (subtraction)

Exponentiation as seen, has the highest precendence

Operator Associativity:

If more than one operator with the same level of precedence exists, **rules of associativity** indicate the order in which an expression is evaluated.

For operators that follow the associative law (such as addition) the order of evaluation doesn't matter

(2 + 3) + 4 → **9**   2 + (3 + 4) → **9**

Division and subtraction, however, do not follow the associative law,

(a) (8 - 4) - 2 → 4 - 2 → **2**   8 - (4 - 2) → 8 - 2 → **6**

(b) (8 / 4) / 2 → 2 / 2 → **1**   8 / (4 / 2) → 8 / 2 → **4**

Operator associativity

1. ** (exponentiation) **(right to left)**

The following operators follow Left to Right associativity

2. - (negation)
3. * (multiplication) , / (division) , // (truncation divison) , % (division)
4. + (addition) , - (subtraction)

Therefore, 23**2**

2 ** (3 ** 2) → **512** **and not** (2 ** 3) ** 2 → **64**

bitwise operator:

These operators, when used operate on the binary value of the given operand.

They operate on a bit-level, which means that each binary digit is considered.

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

& AND

```
a = 5       # 0101

b = 6       # 0110

c = a & b   # 0100  (4)
```

| OR

```
a = 5       # 0101

b = 6       # 0110

c = a | b   # 0111  (7)
```

^ XOR

a = 5 # 0101

b = 6 # 0110

c = a ^ b # 0011 (3)

>> Right Shift

a = 5 # 0101

b = a >> 2 # 0001

Working:

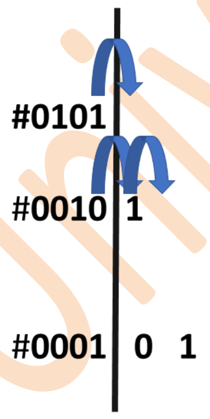
a = 5 **#0101**

Shift 1 bit to the right **#0010 1**

a = 2

Shift 1 bit to the right **#0001 0 1**

a = 1



<< Left Shift

a = 5   # 0101

b = a << 2   # 010100

Working:

a = 5   **#010 1**

Shift 1 bit to the left   **#0101 0**

a = 10

Shift 1 bit to the left   **#010100**

a = 20

ONE'S COMPLIMENT   : change 0 to 1 and 1 to 0

a = 5

print ("~ a ", ~a) # 111111111 .... 1010 => -6

A general rule of thumb is ~x = -(x+1)

relational operators:

Relational operators work on Boolean values. i.e, they form Boolean expressions

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python

A **Boolean expression** is an expression that evaluates to a Boolean value

Relational operators are used to compare two values.

Relational Operators	**Example**	**Result**
== equal	12 == 12	True
!= not equal	12 != 12	False
< Less than	10 < 10	False
> Greater than	“Cat” > “Dog”	False
<= less than or equal to	12 <= 12	True
>= greater than or equal to	12 >= 12	True

# **simple comparison**

print("10 == 10", 10 == 10) # True

print("3 > 2 : ", 3 > 2) # True

# **cascading comparison**

# a op1 b op2 c is same as (a op1 b) and (b op2 c)

print("3 > 2 > 1 : ", 3 > 2 > 1)

print("10 == 10 == 10 : ", 10 == 10 == 10)

# a > b > c : (a > b) and (b > c)

# **string comparison:**

compares the corresponding characters based on the coding - based how the character is stored as a number in the computer - until a mismatch or one or both strings end.

print("cat > car : ", "cat" > "car") # True # "t" > "r"

print("cat > cattle : ", "cat" > "cattle") # False : second string is longer and therefore bigger

print("cat == Cat : ", "cat" == "Cat") # False : "C" < "c" print("apple > z : ", "apple" > "z") # False ; comparison not based on the length

print("zebra > abcdefgh : ", "zebra" > "abcedefgh") # True "z" > a"; rest do not matter

Membership Operators

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

Membership Operators	**Example**	**Result**
in	11 in (11,12,13)	True
not in	11 not in (11,12,13)	False

The membership operators can also be used to check if a given string occurs within another string

print("c in cat", "c" in "cat") #True

print("at in cat", "at" in "cat") # True

print("ct in cat", "ct" in "cat") #False

print("ta in cat", "ta" in "cat") #False

print("cat" > "cat") # False

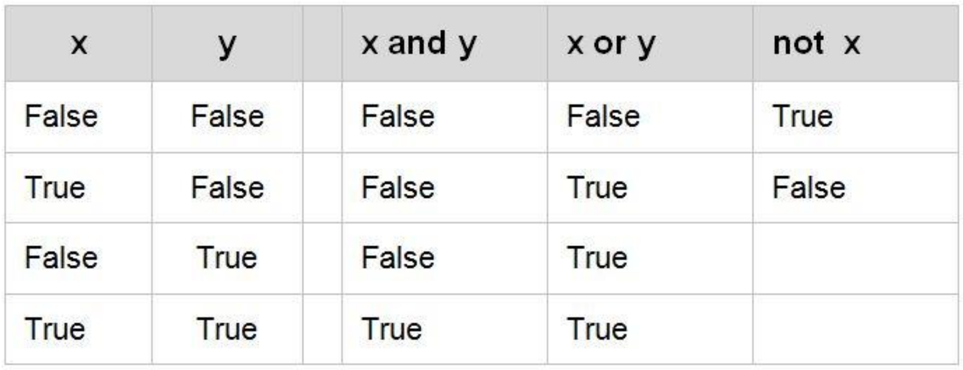
print("cat" >= "cat") # True

Boolean (Logical) Operators

Boolean algebra contains a set of **Boolean** (**logical**) **operators**

Denoted by **and**, **or**, and **not**.

These logical operators can be used to construct arbitrarily complex Boolean expressions



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

Boolean operators work on Boolean values and Boolean equivalent values

- **False Values**: False. 0 , '' (Empty String), [] , {} , () (Empty Collections)
- **True Values:** True**,** non – Zero numbers , Non Empty String, Non Empty Collections

a = 10

b = 10

print (not (a == b)) # False

print(a > 5 and b > 5) # True

print(a > 5 and b < 5) # False

print(a < 5 and b < 5) # False

a = 0

b = 10

#print(b / a > 5) # division by zero

print(a == 0 or b / a > 5)

# **short circuit evaluation**

Evaluate a logical expression left to right and stop the evaluation as soon as the truth or the falsehood is found.

- logical **and**, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false
- logical **or**, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true.
- Python interpreter does not evaluate the second operand when the result is known by the first operand alone
- This is called **short-circuit (lazy) evaluation**

Examples:

print(3/0) #results in a runtime error

the following statements do not result in an error, thanks to the concept of short-circuit evaluation

print(0 and 3/0) #False

print(2 or 3/0) #True

#cascading operation

a=4

b=1

c=2

print(a>b and a>c)

print(a>b>c) # a>b and b>c

Logical operators:

These operators not and or operate on boolean values.

In Python, the following are true.

True 5 -5 1 "python" ["we", "love", "python"] non-empty-data-structure

In Python, the following are false.

False 0 "" [] None empty-data-structure

# operators and polymorphism:

Some operators behave differently based on the type of the operands. They exhibit different forms. These are said to be polymorphic.

operator + on numbers is addition operator.

operator + on strings, tuples, lists is concatenation operator - juxtapose the two items.

operator * on numbers is multiplication operator.

Operator * on strings, tuples, lists with an integer is replication operator - repeat the elements # of times.

Code 1:

```
print(10 + 20) # 30

print("one" + "two") # onetwo

print(2 * 3) # 6

print("2" * 3) # 222 # replicate

print("python" * 3) # pythonpythonpython

print(3 * "2") # 33 # commutative.
```

Identity Operators

Checks if the operands on either side of the operator point to the same object or not

Denoted by **is** and **is not**

Example:

```
10 is 10          #True

10 is not 11      #True
```

Assignment / Shorthand Operators

Combines arithmetic and assignment operators

Operator	**Expression**	**Short Hand**
+= (Addition)	a = a + b	a += b
-= (Subtraction)	a = a - b	a -= b
*= (Multiplication)	a = a * b	a *= b
/= (Division)	a = a / b	a /= b
//= (Truncation Division)	a = a // b	a //= b
%= (Modulus)	a = a % b	a %= b
**= (Exponentiation)	a = a ** b	a **= b

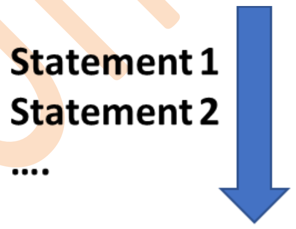
Control Structure:

Control flow is the order that instructions are executed in a program. A control statement is a statement that determines the control flow of a set of instructions.

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

- sequential control
- selection control, and
- iterative control

Sequential control is a form of control in which instructions are executed in the order that they are written. A program consisting of only sequential control is referred to as a "straight-line program."



Selection control is provided by a control statement that selectively executes instructions based on a given condition.

Iterative control is provided by an iterative control statement that repeatedly executes instructions based on a given condition.

Collectively a set of instructions and the control statements controlling their execution is called a control structure.

We want to indicate that the following statements form a group and are to be executed repeatedly in a while or based on selection. We require some mechanism for grouping. In case of C language, we use flower brackets or braces. **In Python, the grouping is controlled by indentation.**

The structure is called the leader suite(body) combination.

Leader ends in a '**:**' and the suite (which could have one or more statements) will have higher indentation compared to the leader.

```
leader:
        suite
```

Let us examine with few codes. To start with let's look at the **while loop.**

While Statement:

Iteration is a concept that is used when we want a task to be repeated multiple times. For example, let's say I want to write a program that simulates the act of waiting for the rain to stop so that I can go out.

Now, how would we wait?

We would probably look outside the window periodically to see if the rain has indeed stopped.

Is it raining? Yes, then Stay indoors

Is it raining? Yes, then Stay indoors

Is it raining? Yes, then Stay indoors

…

…

And the lines go on and on.

How can we write it in a shorter way?

So lets write a quick algorithm:

```
While it is still raining
        stay indoors
```

since we know the repetitive process and when to stop checking, we can use an iterative process

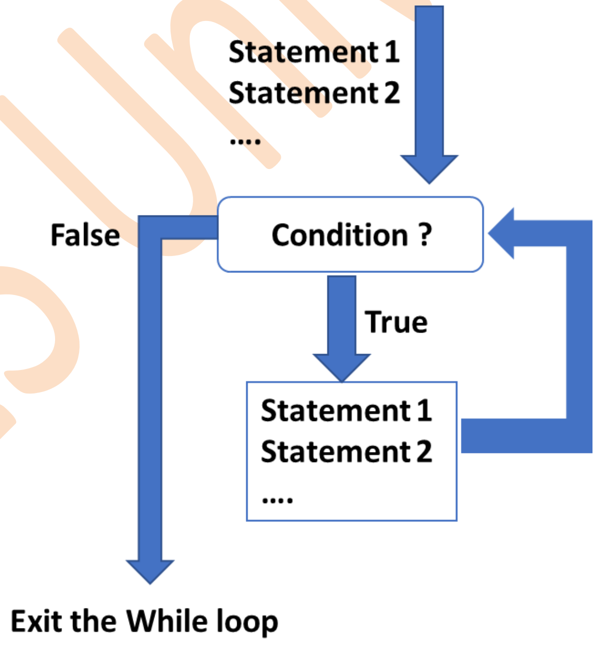
Any process that has to be repeated, in a programming language we use an iteratice statement.

A while statement is an iterative control statement that repeatedly executes a set of statements based on a provided condition. All iterative control needed in a program can be achieved by use of the while statement.

Syntax:

```
While condition:
        Statement(s)
```

Flow Chart:



#write a Program that tells us that it has stopped raining

```
raining = True
while(raining):
        print("Stay indoors")
print("it has stopped raining, you can go out now")
```

Now that we have understood what an iterative statement is, lets look at the following examples to understand how they work and what its characteristics are.

# Write a program to print the numbers from 1 to 10.

Code 1:

```
i=1
while i<10:
        print(i)
```

The above code results in an **infinite loop**. As the condition i<10 is true forever.

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.

Code 2:

```
i=1
while i<11:
        print(i)
        i=i+1
print ("end of loop")
# prints the required output.
```

Program 2

Program to find the sum of 10 natural numbers.

```
i=1
sum=0
while i<=10:
        sum=sum+i
        i=i+1
print(sum)
```

Output:

```
55
```

Definite vs. Indefinite Loops

A definite loop is a program loop in which the number of times the loop will iterate can be determined before the loop is executed.

For example:

```
sum =0

i=1

n = int(input('Enter value: '))

while i<=n:

        sum=sum + i

        i=i+1

print(sum)
```

Although it is not known what the value of n will be until the input statement is executed, its value is known by the time the while loop is reached. Thus, it will execute “n times.”

An indefinite loop is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed.

Example:

```
inpt=input("Enter selection: ")

while inpt != 'F' and inpt !='C':

        inpt = input("Please enter 'F' or 'C': ")
```

In this case, the number of times that the loop will be executed depends on how many times the user mistypes the input. Thus, a while statement can be used to construct both definite and indefinite loops.

There are 3 ingredients to any iterative statement:

1. Initial value of the iterative counter or condition
2. Final Iterative condition
3. Updating the iterative counter or condition in order to reach the final iterative condition

selection:

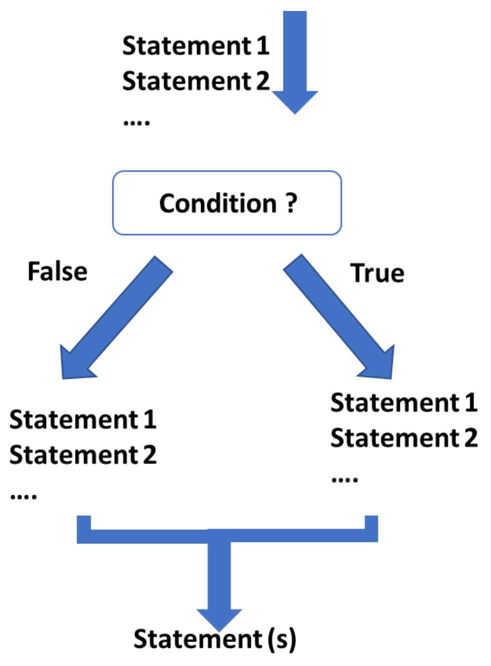
In looping, we execute the body repeatedly.

In selection, we choose one of the alternates. Like in the English statement, if this ___, then that____

syntax :

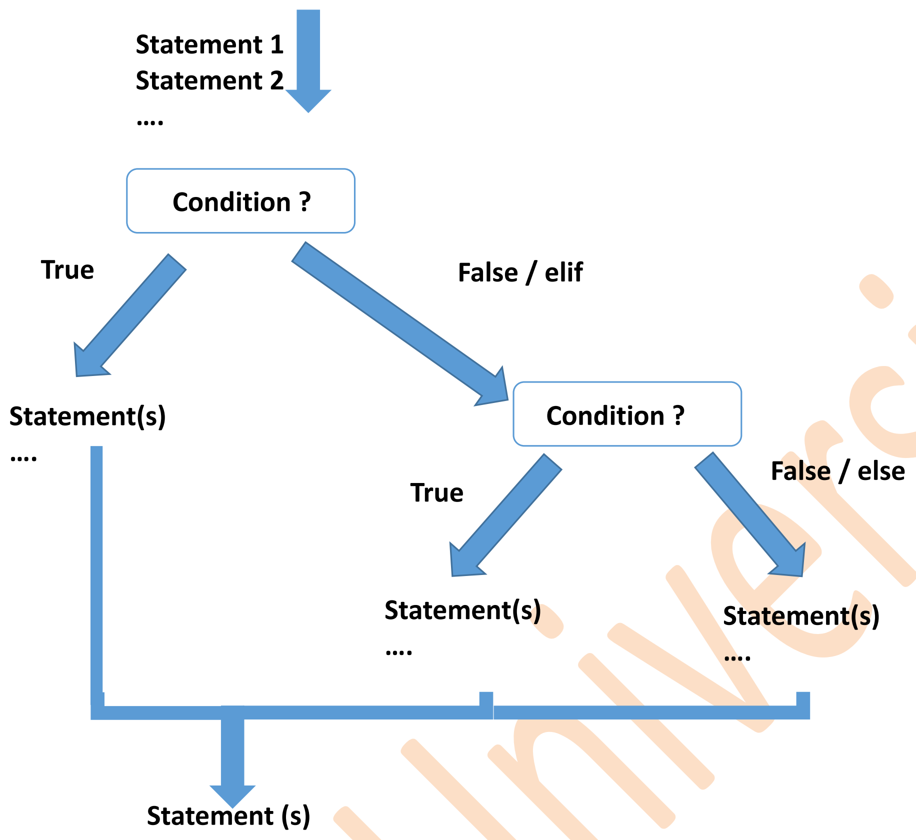
```
 if <expr> :
          <suite>


if <expr> :
          <suite>
else:
          <suite>
```

Flow chart


Statement 1
Statement 2
....
Condition ?
False
True
Statement 1
Statement 2
....
Statement 1
Statement 2
....
Statement (s)

```
if <expr> :

        <suite>

Elif <expr>:

        <suite>

else:

        <suite>
```

Flow chart



Program to check whether given two strings are same

```
a = input("enter string 1  ")

b = input("enter string 2  ")

""""
```

check this out!

if a = b : #assignment statement and not an expr. Throws an error.

```
        print("equal")
"""
if a == b :
        print("equal")
else:
        print("not equal")
```

Input / Output:

```
enter string 1   Python
enter string 2  Python
equal
```

Input / Output:

```
enter string 1 python
enter string 2 program
not equal
```

Program to find whether a given number is positive, negative or zero.

```
num=int(input("enter a number"))

if num>0:

        print("its positive")

elif num<0:

        print("its negative")

else:

        print("its zero")
```

input/output:

```
enter a number 5

its positive

enter a number 0

its zero

enter a number -1
```

its negative

Dangling else problem:

If we have two ifs and a single else, to which if should we associate the else.

This is called dangling else problem. In Python, else matches the if with the same indentation as else.

The code with comments is self-evident.

```
# dangling else problem
#       two if and single else
#       else is paired with the if based on indentation
a = int(input())
b = int(input())
c = int(input())
if a == b :
        if b == c :
                print("what?")
```

```
else:  # paired with the outer if; control reaches here if a not equal to b
                    print("what else?")


if a == b :
          if b == c :
                    print("what")
          else: # paired with the inner if; control reaches here if a equals b and b is not
equal to c
                    print("what else")
```