

Python for Data Science, AI & Development

Course Content

- Intro to python
- Python basics syntax, comments, variables
- data types
- Print statement
- User inputs
- Strings, String methods
- Strings (cont)
- Operators (Arithmetic, Logical ...)
- List, tuple, set and Dictionary
- Control statement If Else
-
- Loops
- Functions
- Scopes
- Try & Except
- File Handling
- Pre-built & User Defined Modules
- OOP



Coding & Programming is not difficult!

Writing code in computer vs playing notes in piano



Practice, practice and practice for coding.

Coding VS Programming

Coding is a subset of Programming



Coding

- Machine-readable inputs
- Writing lines of codes
- Language and syntax

Programming

- Creating and developing an executable machine program
- Debugging and testing
- Translating requirements
- Documentation review and analysis

**There are five basic concepts to
any programming language:**

VARIABLES

CONDITIONAL STATEMENTS

LOOPING & ITERATION

DATA TYPES & STRUCTURES

FUNCTIONS

Programming Syntax:

the set of rules that defines how a programming languages will be written.

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!");
}
```

C

```
#include <iostream.h>
int main()
{
    std::cout << "Hello, world! ";
    return 0;
}
```

C++

```
class HelloWorld {
    public static void main(String[]
args) {
        System.out.println("Hello,
World!");
    }
}
```

Java

```
print("Hello, world!")
```

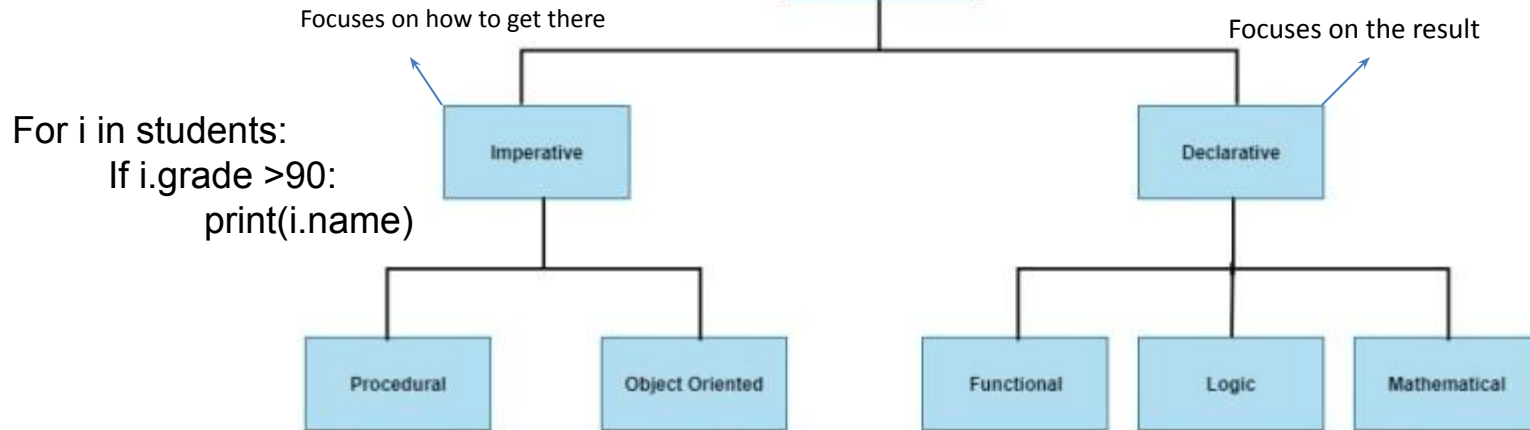
Python

Programming paradigms:

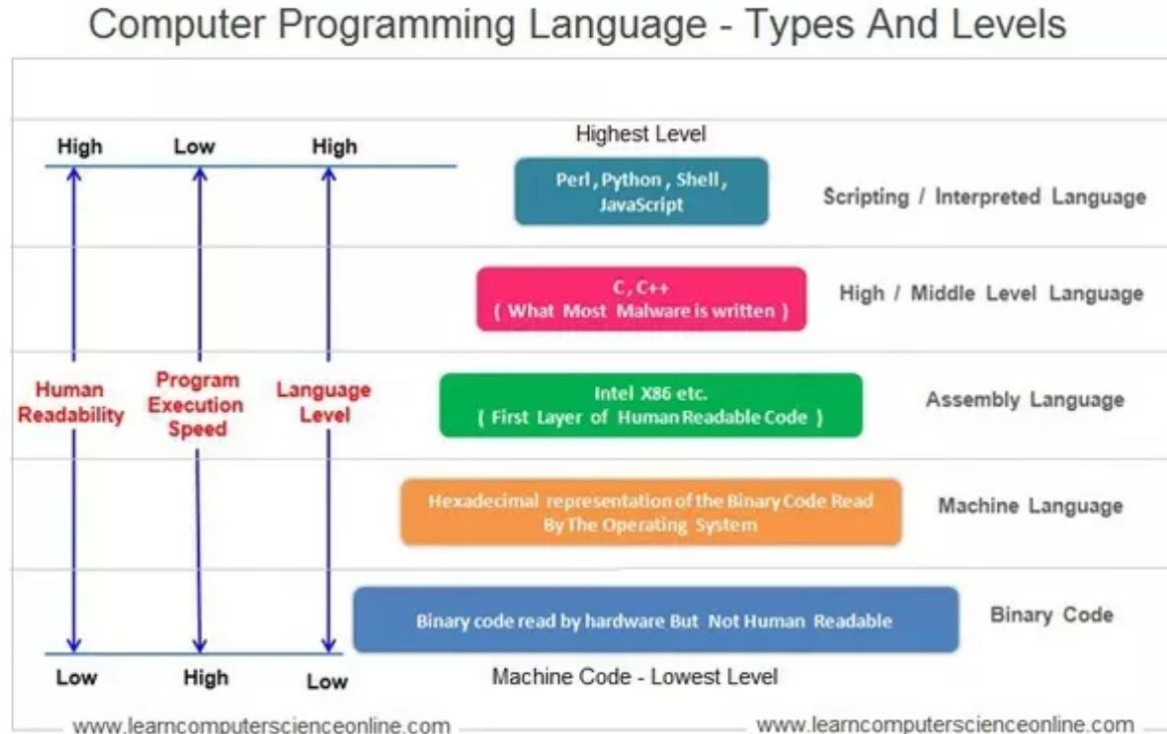
Are a way to classify programming languages based on their features.

Sql

Select name FROM students WHERE grade > 90



Levels of programming languages :



MOV AL , 61

Compiler VS Interpreter :

Compiler :

1. scans the entire program and translates the whole of it into machine code at once.
2. the overall time taken to execute the process is much faster.
3. C and C++ for example.

- Interpreter :

1. translates just one statement of the program at a time into machine code
2. the overall time to execute the process is much slower
3. Python for example



What is python ?

- “High-level”, general-purpose programming language
- High-level: relatively similar to human language and abstracted from “machine code”
 - `for element in my_list:`
 `print(element)`
- General-purpose: many applications!

```
for(i=0; i>10;i++){  
  
}
```

Python uses

- Building desktop applications, including GUI applications, CLI tools, and even games
- Doing mathematical and scientific analysis of data
- Building web and Internet applications

- **Example:**

- Python programming: Reddit, Dropbox, and YouTube

- Python web framework Django: Instagram and Pinterest.

- Doing computer systems administration and automating tasks
- Building AI applications

Programming branches in Python

- Python Programming
- Python for Science and Engineering
- Python for Control Engineering
- Python for Software Development
- Python for Data science

Careers with Python

- Game developer
- Web designer
- Python developer
- Full-stack developer
- Machine learning engineer
- Data scientist
- Data analyst
- Data analyst
- Data engineer
- DevOps engineer
- Software engineer
- Many more other roles
- Software engineer
- Many more other high salary

```
jupyter notebook --notebook-dir="H:\DEPI ENGBARAA\python_basics"
```

This command **launches Jupyter Notebook** and **sets the starting (default) folder** to:

 E:\DEPI R3 python course

So every time you run this command, Jupyter will open directly inside that specific folder

Important Note:

This command **only changes the directory for this session.**

Once you close the terminal or Jupyter, it will **not remember the path next time.**

Print Statements

- Since Python is one of the most readable languages out there, we can print data on the terminal by simply using the print statement.
- The text Hello World is bounded by quotation marks because it is a string or a group of characters.
- Next, we'll print a few numbers. Each call to print moves the output to a new line

```
print("Hello World")
```

Hello World

```
print(50)  
print(1000)  
print(3.142)
```

50
1000
3.142

Print Statements

- We can even print multiple things in a single print command; we just have to separate them using commas.
- By default, each print statement prints text in a new line. If we want multiple print statements to print in the same line, we can use the following code.
- The value of end is appended to the output and the next print will continue from here

```
print(50, 1000, 3.142, "Hello World")
```

```
50 1000 3.142 Hello World
```

```
print("Hello", end="")  
print("World")
```

```
print("Hello", end=" ")  
print("World")
```

```
HelloWorld  
Hello World
```

Quiz

How can we print the text, "IBM_Data_Science" in Python?

- `print IBM_Data_Science`
- `Print " IBM_Data_Science"`
- `print("IBM_Data_Science")`
- `print(IBM_Data_Science)`

Quiz

How can we print the text, "IBM_Data_Science" in Python?

- `print IBM_Data_Science`
- `Print " IBM_Data_Science"`
- `print("IBM_Data_Science")`
- `print(IBM_Data_Science)`

Comments

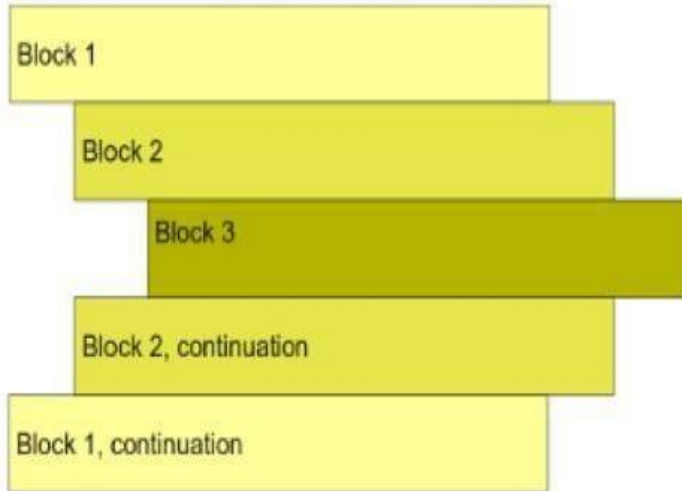
- Single comment: #
- Multiline comment/docstrings: between three double/single quotes `"""` `'''`. Or you can just write # multiple times.
- You can use Ctrl+ "/" to automatically comment a whole section.

```
1 """ Docstrings are pretty cool
2 for writing longer comments
3 or notes about the code"""
4
```

```
1 print(50) # This line prints 50
2 print("Hello World") # This line prints Hello World
3
4 # This is just a comment hanging out on its own!
5
6 # For multi-line comments, we must
7 # add the hashtag symbol
8 # each time
9
```

Indentation

- Indentation is Important in Python. It Illustrate how code blocks are formatted.



```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
    print ("False")
```

An orange oval is drawn around the last line of code, `print ("False")`, and an arrow points from the oval to the word `error`.

Quiz

What will be printed by the following code?

- On
e
Two
Three
- On
e
Three
- Two
- three

```
# print ("One")  
print("Two")  
# Three
```

Python Statements & Variables

- Python statements means a logical line of code that can be either expression or assignment statement.
- Expression means a sequence of numbers, strings, operators and objects that logically can be valid for executing.
- Simple assignment statement means a statement with its R.H.S just a value-based expression or a variable or an operation.
- Augmented assignment statement means a statement where the arithmetic operator is combined in the assignment.

Notes:

- A statement can be written in multi-lines by using \ character.
- Multiple statements can be written in same line with ; separator.

Variables

- A variable is simply a name to which a value can be assigned.
- Variables allow us to give meaningful names to data.
- The simplest way to assign a value to a variable is through the = operator.
- A big advantage of variables is that they allow us to store data so that we can use it later to perform operations in the code.
- Variables are mutable. Hence, the value of a variable can always be updated or replaced.

```
counter = 100          # An integer assignment
miles   = 1000.0        # A floating point
name    = "John"       # A string
z       = None          # A null value
```


Variables

Variables can change type, simply by assigning them a new value of a different type.

```
x = 1  
x = "string value"
```

Python allows you to assign a single value to several variables simultaneously.

```
a = b = c = 1
```

You can also assign multiple objects to multiple variables.

```
a, b, c = 1, 2, "john"
```

Variables are Case- Sensitive: x is different from X.

Variables

Do not need type definition when creating a variable. In C Language for example, to declare a variable:

D `int x = 4`

In python, no need for declaring the type 'int'

D `x = 4`

It knows automatically that x is an integer.

Identifiers

Python identifiers means the user-defined name that is being given to anything in your code as the variables, function, class or any other object in your code. Guidelines for creating a python identifiers:

1. Use any sequence of lower case (a - z) or upper case (A - Z) letters in addition to numbers (0 - 9) and underscores (_).
2. No special characters or operators are allowed. (,), [,], #, \$, %, !, @, ~, &, +, =, -, /, \.
3. Don't start your identifier with a number as it is not valid.
4. Some keywords are reserved as False, True, def, del, if, for, raise, return, None, except, lambda, with, while, try, class, continue, as, assert, elif, else, is, in, import, not, from global, pass, finally and yield. You can't name an identifier with these words on their own however you can use them as sub-name such as True_stat or def_cat
5. Make the user-defined name meaningful.

Identifiers

6. Python is case sensitive language so variable1 identifier is different from Variable1 identifier.

Identifiers or user-defined names has a convention in python which is as follows:
Never use l “lower-case el” or I “Upper-case eye” or O “Upper-case Oh” as single character variable name as in some fonts they are indistinguishable

Python Data Types

Data type means the format that decides the shape and bounds of the data. In python, we don't

have to explicitly predefine the variable data type but it is a dynamic typing technique.

Dynamic typing means that the interpreter knows the data type of the variable at the runtime from the syntax itself.

The data types in python can be classified into:

1 – Numbers

2 - Booleans

3 – Strings

4 - Bytes

5 – Lists

6 - Arrays

7 – Tuples

8 - Sets

9 - Dictionaries

- Any variable in python carries an instance of object which can be mutable or immutable. When an object is created it takes a unique object id that can be check by passing the variable to the built-in function `id()` . The type of the object is defined at runtime as mentioned before.
- Immutable objects can't be changed after it is created as int, float, bool, string, Unicode, tuple.
- Mutable objects can be changed after it is created as list, dict, set and user-defined classes.

Numbers

- Python is one of the most powerful languages when it comes to manipulating numerical data.
- There are three built-in data types for numbers in Python:
 - Integer (int)
 - Floating-point numbers (float)
 - Complex numbers: <real part> + <imaginary part>j (not used much in Python programming)

```
1 print(10) # A positive integer
2 print(-3000) # A negative integer
3
4 num = 123456789 # Assigning an integer to a variable
5 print(num)
6 num = -16000 # Assigning a new integer
7 print(num)
```

```
print(complex(10, 20)) # Represents the complex number (10 + 20j)
print(complex(2.5, -18.2)) # Represents the complex number (2.5 - 18.2j)
```


Common Number Functions

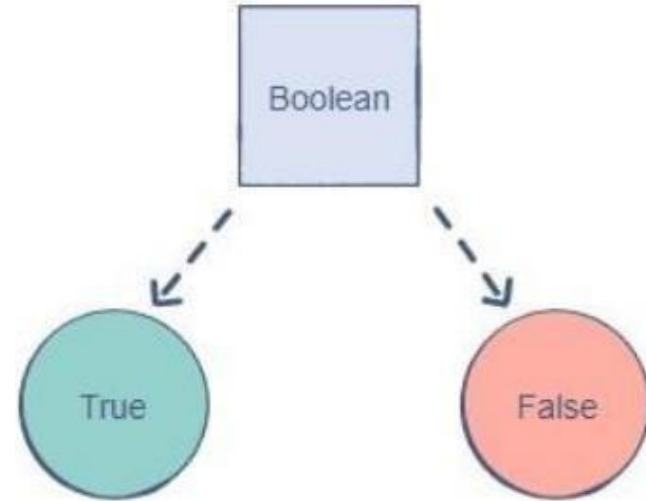
Function	Description
int(x)	to convert x to an integer
float(x)	to convert x to a floating-point number
abs(x)	The absolute value of x
sqrt(x)	The square root of x for $x > 0$
log(x)	The natural logarithm of x, for $x > 0$
pow(x,y)	The value of $x^{**}y$

Booleans

- The Boolean (also known as bool) data type allows us to choose between two values: true and false.
- In Python, we can simply use True or False to represent a bool

Note: The first letter of a bool needs to be capitalized in Python.

- A Boolean is used to determine whether the logic of an expression or a comparison is correct. It plays a huge role in data comparisons.



Strings

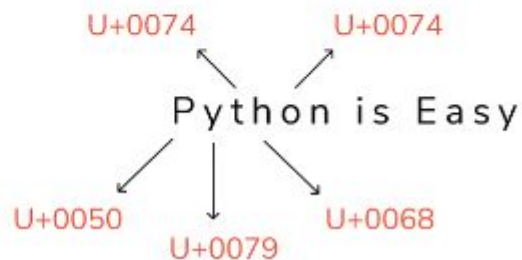
- A string is a collection of characters closed within single or double quotation marks
- (immutable).
- You can update an existing string by (re)assigning a variable to another string.
- Python does not support a character type; these are treated as strings of length one.

```
>>> str= "strings are immutable!"
>>> str[0]="S"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
print ("Python is Easy")
```

 read it as..... Python is Easy

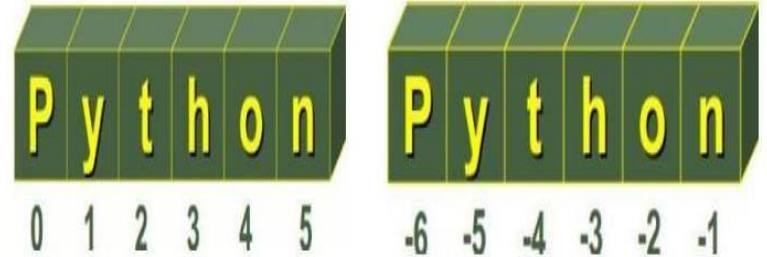
 reads it as.....



Strings

- Python accepts single ('), double (") and triple (''' or ''') quotes to denote string literals.
- String indexes starting at 0 in the beginning of the string and working their way from -1 at the end

```
name1 = "sample string"  
name2 = 'another sample string'  
name3 = """a multiline  
string example"""
```



Strings Indexing

"Python for everyone"

- You can index a string (access a character) by using square brackets:

```
1  batman = "Bruce Wayne"
2
3  first = batman[0]  # Accessing the first character
4  print(first)
5
6  space = batman[5]  # Accessing the character at index 5
7  print(space)
8
```

```
1  batman = "Bruce Wayne"
2  print(batman[-1])  # Corresponds to batman[10]
3  print(batman[-5])  # Corresponds to batman[6]
```

Strings Slicing

- Slicing is the process of obtaining a portion (substring) of a string by using
- its indices. Given a string, we can use the following template to slice it
- and obtain a substring `String [start:end]`
 - start is the index from where we want the substring to start.
 - end is the index where we want our substring to end.
- The character at the end index in the string, will not be included in the substring obtained through this method.

```
1 my_string = "This is MY string!"
2 print(my_string[0:4]) # From the start till before the 4th index
3 print(my_string[1:7])
4 print(my_string[8:len(my_string)]) # From the 8th index till the end
```

Output

```
This
his is
MY string!
```

Strings Slicing with a step

- Until now, we've used slicing to obtain a contiguous piece of a string, i.e., all the characters from the starting index to before the ending index are retrieved.
- However, we can define a step through which we can skip characters in the string. The default step is 1, so we iterate through the string one character at a time.
- The step is defined after the end index

```
1 my_string = "This is MY string!"
2 print(my_string[0:7]) # A step of 1
3 print(my_string[0:7:2]) # A step of 2
4 print(my_string[0:7:5]) # A step of 5
5
```

Output

```
This is
Ti s
Ti
```


Strings Reverse Slicing

- Strings can also be sliced to return a reversed substring. In this case, we would need to
- switch the order of the start and end indices.
- A negative step must also be provided The step is defined after the end index

```
1 my_string = "This is MY string!"
2 print(my_string[13:2:-1]) # Take 1 step back each time
3 print(my_string[17:0:-2]) # Take 2 steps back. The opposite of what happens in the slide above
4
```

Output

```
rts YM si s
!nrsY ish
```

Strings Partial Slicing

- One thing to note is that specifying the start and end indices is optional.
- If start is not provided, the substring will have all the characters until the end index.
- If end is not provided, the substring will begin from the start index and go all the way to the end.

```
1 my_string = "This is MY string!"
2 print(my_string[:8]) # All the characters before 'M'
3 print(my_string[8:]) # All the characters starting from 'M'
4 print(my_string[:]) # The whole string
5 print(my_string[::-1]) # The whole string in reverse (step is -1)
6
```

Output

```
This is
MY string!
This is MY string!
!gnirts YM si sihT
```

Quiz

What is the output of the following code?

```
>>my_string = "0123456789"
```

```
>>print(my_string[-2: -6: -2])
```

- 5432
- 8765
- 532
- **86**

Quiz

What is the output of the following code?

```
>>my_string = "0123456789"
```

```
>>print(my_string[-2: -6: -2])
```

- 5432
- 8765
- 532
- 86

Forward direction indexing

0	1	2	3	4	5
---	---	---	---	---	---

String

P	y	t	h	o	n
---	---	---	---	---	---

-6	-5	-4	-3	-2	-1
----	----	----	----	----	----

Backward direction indexing

Common String Operators

- Assume string variable a holds 'Hello' and variable b holds 'Python'

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e a[-1] will give o
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	'H' in a will give True

Common String Operators

- Strings are an example of Python objects. An object contains both data (the actual string itself) and methods, which are effectively functions that are built into the object and are available to any instance of the object.
- `upper()` Return a capitalized version of string
- `lower()` Return a copy of the string converted to lowercase.
- `find()` searches for the position of one string within another
- `strip()` removes white space (spaces, tabs, or newlines) from the beginning and end of a string
- Other useful methods in this [link](#)



Exercises → String

1. Declare a variable named `course` and assign it to an initial value `Python 4 Data Science`.
2. Print the length of the `course` string using `len()` method and `print()`.
3. Change all the characters of variable `course` to uppercase and lowercase letters using `upper()` and `lower()` method.
4. Make the following using string formatting methods:

```
8 + 6 = 14
8 - 6 = 2
8 * 6 = 48
8 / 6 = 1.33
8 % 6 = 2
8 // 6 = 1
8 ** 6 = 262144
```

5. Use a **new line** and **tab** escape sequence to print the following lines. (extra)

Name	Age	Country	City
Milaan	96	Finland	Tampere

Python Coding Questions – String Slicing

```
] : # 🌟 Python Coding Questions - String Slicing

#1: Print the first 5 characters from the following string:
text = "Artificial Intelligence"

#2: Print the last 4 characters of the string:
message = "Hello, Python learners!"

#3: Extract only the word "Python" from the following sentence: Using find() and len()
phrase = "I am learning Python programming"

#4: Use slicing to reverse the following string:
word = "MachineLearning"
```

Python User Input from Keyboard

- Python user input from the keyboard can be read using the `input()` built-in function.
- The input from the user is read as a string and can be assigned to a variable.
- After entering the value from the keyboard, we have to press the "Enter" button. Then the `input()` function reads the value entered by the user.

```
In [*]: input("please enter a value")
```

please enter a value

Python User Input from Keyboard

- The program halts indefinitely for the user input, There is no option to provide timeout value.
- The syntax of input() function is: input("prompt")
- The prompt string is printed on the console and the control is given to the user to enter the value.
- You should print some useful information to guide the user to enter the expected value.

Python User Input from Keyboard

- D What is the type of user entered value?
 - The user entered value is always converted to a string and then assigned to the variable. Let's confirm this by using `type()` function to get the type of the input variable.

- D How to get an Integer as the User Input?
 - There is no way to get an integer or any other type as the user input. However, we can use the built-in functions to convert the entered string to the integer.

```
In [12]: my_input=input("please enter a number")  
          print(type(my_input))
```

```
please enter a number10  
<class 'str'>
```

```
In [13]: my_input=input("please enter a number")  
          my_input=int(my_input)  
          print(type(my_input))
```

```
please enter a number100  
<class 'int'>
```

Printing Strings

D Print statement can have different formats

```
In [18]: ▶ print("my name is ", name)  
my name is  ahmed
```

```
In [20]: ▶ print("my name is {} and my age is {}".format(name, age))  
my name is ahmed and my age is 20
```

```
In [21]: ▶ print("my name is {0} and my age is {1}".format(name, age))  
my name is ahmed and my age is 20
```

```
In [24]: ▶ print("my name is {2} and my age is {0}".format(name, age, gender))  
my name is male and my age is ahmed
```

Escape Characters

Sometimes we want to print strings having special characters. So we add backslash before the special character as follows: "My name is \"Ahmed\" "

Escape Characters:

\': single quote

\\: Backslash

\n: New line

\t: tab

\b: Backspace

```
[176]: url = "H:\nimer ENGBARAA"  
print(url)
```



H:
imer ENGBARAA

```
[186]: url = "H:\\nimer ENGBARAA"  
print(url)
```

H:\nimer ENGBARAA

```
[192]: url = "H:\nimer \t \n \n \n \n ENGBARAA"  
print(url)
```

H:
imer

ENGBARAA

```
•[194]: url = r"H:\nimer \t \n \n \n \n ENGBARAA" # raw data  
print(url)
```

H:\nimer \t \n \n \n \n ENGBARAA

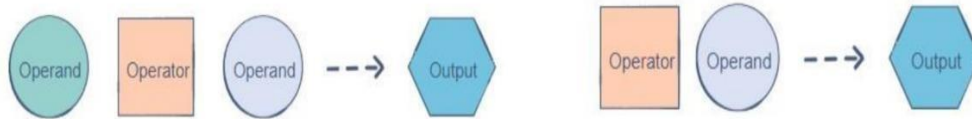
```
[198]: url = "H:\\nimer \\t \\n \\n \\n \\n ENGBARAA"  
print(url)
```

H:\nimer \t \n \n \n \n ENGBARAA

Operators

Operators are used to perform arithmetic and logical operations on data. They enable us to manipulate and interpret data to produce useful outputs.

Python operators follow the infix or prefix notations:



The 5 main operator types in Python are:

Arithmetic - Comparison - Assignment - Logical - Bitwise

Operators

Arithmetic		
Operator	Purpose	Example
+	Addition (Sum of two operands)	a + b
-	Subtraction (Difference between two operands)	a - b
*	Multiplication (Product of two operands)	a * b
/	Float Division (Quotient of two operands)	⁵⁸ a / b
//	Floor Division (Quotient with fractional part)	a // b
%	Modulus (Integer remainder of two operands)	a % b
**	Exponent (Product of an operand n times by itself)	a ** n

Operators

Comparison		
Operator	Purpose	Example
>	Greater than (If left > right hence return true)	a > b
<	Less than (if left < right hence return true)	a < b
==	Equal to (if left equals right return true)	a == b
!=	Not equal to (if left not equals right return true)	59 a != b
>=	Greater than or equal (if left GTE right return true)	a >= b
<=	Less than or equal (if left LE right return True)	a <= b

Operators

Logical		
Operator	Purpose	Example
and	If a and b are both true hence return true	a and b
or	If either a or b is true hence return true	a or b
not	If a is true return false and vice versa	not a

Operators

Logic Tables:

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Truth Table

A	B
0	1
1	0

Operators

Assignment		
Operator	Purpose	Example
arithmetic=	Any arithmetic operation followed by = which apply the arithmetic operation of the left operand and put the result in it.	a += 1 equivalent to a = a + 1
bitwise=	Any bitwise operation followed by = which apply ⁶² the bitwise operation of the left operand and put the result in it.	a &= 1 equivalent to a = a & 1

Operators

Identity		
Operator	Purpose	Example
is	If both operands refers to same object return True	a is b
Is not	If both operands refers to different objects return True	a is not b

Membership		
Operator	Purpose	Example
in	If a given value exists in a given sequence	"s" in ["a", "n", "s"]
not in	If a given value doesn't exist in a given sequence	"s" in ["a", "n", "u"]

Quiz

What is the operator used?

- OR
- XOR
- AND
- NOT

$$\begin{array}{c} 11110010 \\ 10011100 \end{array} = 10010000$$

Quiz

What is the output of printing 'result'?

- False
- -21
- 5
- 5.25

```
x = 20
y = 5
result = (x + True) / (4 - y * False)
```

Task: Simple User Input – Welcome Message

Create a simple Python program that:

1. Asks the user to enter their **full name**
2. Asks for their **department**
3. Asks for their **age (Make sure it's a valid number)**

Then, print a friendly welcome message using an **print using f format**.

Use `while True:` to keep looping (**break**)

Use `try-except` to catch invalid age inputs

Data Containers

Forward direction indexing

0	1	2	3	4	5
---	---	---	---	---	---

String

P	y	t	h	o	n
---	---	---	---	---	---

-6	-5	-4	-3	-2	-1
----	----	----	----	----	----

Backward direction indexing

Data Containers


Python provides several built-in data structures (or containers) to organize and manage data efficiently. These include lists, tuples, sets, and dictionaries.

Lists

A list is an ordered, mutable collection of items. Lists can contain items of different types, including other lists.

Syntax

Lists are defined using square brackets []

```
 fruits = ["apple", "banana", "cherry"]
```

Key Characteristics of list

- **Ordered:** Lists allow access to items using an index.
- **Mutable:** You can add, delete, or modify items in a list.
- **Non-Unique Items:** Lists can contain duplicate elements.
- **Mixed Data Types:** Lists can hold items of different data types.

Data Containers

Lists Operations

Indexing

Access elements using their index (starting at 0)

```
▶ print(fruits[0]) # Output: apple
```

```
↔ apple
```

Slicing

Access a range of elements

```
▶ print(fruits[0:2]) # Output: ['apple', 'banana']
```

```
↔ ['apple', 'banana']
```

Data Containers

Lists Operations

Adding Elements

Use `append()` to add an item, `insert()` to add at a specific index.

```
▶ fruits.append("orange")  
   fruits.insert(1, "mango")
```

Removing Elements

Use `remove()` to delete an item, `pop()` to remove by

```
▶ fruits.remove("banana")  
   fruits.pop(2)
```

```
↔ 'cherry'
```

Tuples

A tuple is an ordered, immutable collection of items. Once defined, its contents cannot be changed.

Syntax

Tuples are defined using parentheses ()

```
coordinates = (10, 20)
```

Operations

Indexing and Slicing: Similar to lists.

```
print(coordinates[0]) # Output: 10
```

→ 10

Immutability

Cannot change elements, which makes tuples faster for read-only data.

Tuples

Key Characteristics of Tuples

- **Ordered:** Tuples allow access to items using an index.
- **Immutable:** You cannot add, delete, or modify items in a tuple.
- **Non-Unique Items:** Tuples can contain duplicate elements.
- **Mixed Data Types:** Tuples can hold items of different data types.

Sets

A set is an unordered collection of unique items. Sets are used for membership testing and eliminating duplicate entries.

Syntax

Sets are defined using curly braces `{}` or the `set()` function

```
unique_numbers = {1, 2, 3, 4}
```

Operation

Adding Elements: Use `add()`

```
unique_numbers.add(5)
```

Sets

Key Characteristics of Sets

- **Unordered:** Sets do not maintain a specific order of items.
- **No Indexing/Slicing:** Sets do not support indexing or slicing operations.
- **Immutable Data Types:** Sets can only contain immutable data types (e.g., numbers, strings, tuples).
- **Unique Items:** Sets automatically remove duplicate items, ensuring all elements are unique.

Sets

Operations

Removing Elements: Use `remove()` or `discard()`

```
unique_numbers.remove(2)
```

Set Operations: Union, intersection, difference.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
print(set_a | set_b) # Union: {1, 2, 3, 4, 5}
print(set_a & set_b) # Intersection: {3}
print(set_a - set_b) # Difference: {1, 2}
```

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
```

Dictionaries

A dictionary is an unordered collection of key-value pairs. Keys must be unique and immutable (e.g., strings, numbers).

Syntax

Defined using curly braces { } with key-value pairs separated by

```
student = {"name": "Alice", "age": 20, "courses": ["Math", "Science"]}
```

Operations

Accessing Values: Use keys.

```
print(student["name"]) # Output: Alice
```

→ Alice

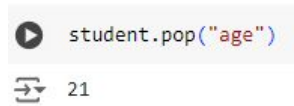
Adding/Updating: Assign a value to a key.

```
[15] student["age"] = 21  
      student["grade"] = "A"
```

Dictionaries

Operation

Removing Items: Use `pop()` to remove a key-value pair.



```
student.pop("age")
```

21

Key Characteristics of Dictionaries

- **Key-Value Pairs:** Dictionaries store data as key-value pairs.
- **Immutable Keys:** Dictionary keys must be immutable (e.g., numbers, strings, tuples).
- **Flexible Values:** Dictionary values can be of any data type.
- **Unique Keys:** Each key in a dictionary must be unique.
- **Unordered:** Dictionaries are not ordered; elements are accessed using keys.

Comparing Lists, Tuples, Sets, Dictionaries

List	Tuple	Set	Dictionary
[] or list()	() or tuple()	{ } or set()	{ } or dict()
Ordered	Ordered	Unordered	Unordered
Changeable	Unchangeable	Unchangeable	Changeable
Indexed	Indexed	Unindexed	Key Value pair
Allows Duplicates	Allows Duplicates	No Duplicates	No Duplicates
Allows Slicing	Allows Slicing	No Slicing	No Slicing

▼ Exercises → Questions on Python Data Types

1. Given the list `my_list = ['apple', 'banana', 'cherry', 'date']`, perform the following:

- Access the first element in the list.
- Remove the last element from the list.
- Insert the string `'grape'` at index 2.
- Print the modified list.

2. Create a tuple containing the numbers 1, 2, 3, 4, 5. Then:

- Access the third element of the tuple.
- Try to modify the second element of the tuple. What happens?
- Print the tuple.

3. Given the dictionary `my_dict = {'brand': 'Toyota', 'model': 'Corolla', 'year': 2019}`, perform the following:

- Access the value associated with the key `'model'`.
- Remove the key `'year'` from the dictionary.
- Add a new key `'color'` with the value `'blue'`.
- Print the modified dictionary.

▼ Data type → Level 2

```
food_tuple = ('fruits_vegetables', 'meat', 'fish', 'mushroom', 'beverage')
```

1. Slice out the middle item or items from the `food_tuple` tuple.

Note: If you are not familiar with using `if-else` statements in Python, please skip this question.

Note:

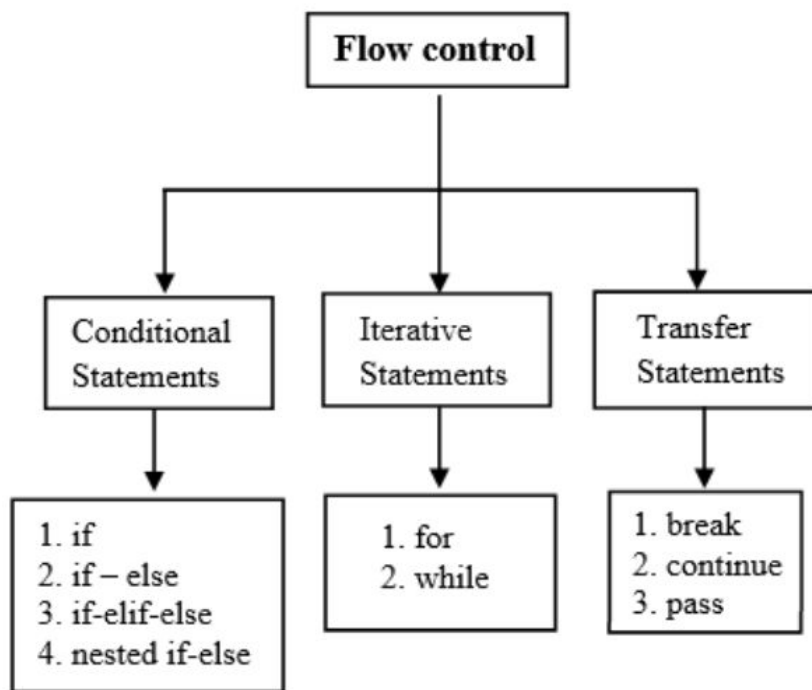
- If the length is odd, there is one middle item
- And If the length is even, there are two middle items

Control Flow

▼ Control Flow Statements

The flow control statements are divided into **three** categories:

1. **Conditional statements**
2. **Iterative statements**
3. **Transfer/Control statements**



Control Flow

Control flow statements are essential in programming as they enable the execution of specific blocks of code based on conditions. This allows for dynamic decision-making within a program. In Python, the primary control flow statements include `if`, `elif`, and `else`. We'll cover these concepts with detailed explanations and multiple examples to illustrate their use.

Control Flow

If

Statements

The if statement is used to test a specific condition. If the condition evaluates to True, the code block under the if statement is executed.

Syntax

```
if condition:  
    # Code to execute if condition is true  
.
```

If Statements

Example

S
Check Positive
Number

```
▶ number = 5  
  if number > 0:  
    print("The number is positive.")
```


↔ The number is positive.


This code checks if the number is greater than zero. Since $5 > 0$ is true, it prints "The number is positive."

If Statements

Example

Check String Length

```
 name = "Alice"  
if len(name) > 3:  
    print("The name is longer than 3 characters.")
```

 The name is longer than 3 characters.

The code checks if the length of the string name is greater than 3. Since "Alice" has 5 characters, the condition is true, and it prints "The name is longer than 3 characters."

If Statements

Example

Check Membership in List

```
▶ fruits = ["apple", "banana", "cherry"]  
if "banana" in fruits:  
    print("Banana is in the list.")
```

➡ Banana is in the list.

The code checks if "banana" is an item in the fruits list. Since it is, the condition is true, and it prints "Banana is in the list."

If Statements

Example

Check if Variable is Not None

```
result = None  
if result is None:  
    print("Result is not set yet.")
```

Result is not set yet.

This code checks if result is None. Since it is, the condition is true, and it prints "Result is not set yet."

Control Flow

If-Else Statements

The if-else statement provides an alternative block of code that is executed if the condition is False.

If-Else Statements

Example

Check Even or Odd Number

```
number = 4
if number % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

Even number

This code checks if number is even by using the modulus operator %. Since $4 \% 2 == 0$ is true, it prints "Even number."

If-Else Statements

Example

Age Verification

```
▶ age = 17
  if age >= 18:
    print("You are eligible to vote.")
  else:
    print("You are not eligible to vote.")
```

↔ You are not eligible to vote.

The code checks if age is 18 or older. Since age is 17, the condition is false, so it prints "You are not eligible to vote."

If-Else Statements

Example

Check for Empty List

```
▶ items = []  
if items:  
    print("The list is not empty.")  
else:  
    print("The list is empty.")
```

⇒ The list is empty.

The code checks if items is a non-empty list. Since items is empty, the condition is false, and it prints "The list is empty."

If-Else Statements

Example

Temperature Check

```
temperature = 15
if temperature > 20:
    print("It's warm outside.")
else:
    print("It's cold outside.")
```

➡ It's cold outside.

This code checks if the temperature is greater than 20. Since 15 is not greater than 20, it prints "It's cold outside."

Control Flow

If-Elif-Else

Statements

The if-elif-else statement allows checking multiple conditions sequentially. Only the block of the first True condition is executed. If none of the conditions are True, the else block is executed.

If-Elif-Else Statements

Examples

Grading System

```
▶ score = 72
  if score >= 90:
      print("Grade A")
  elif score >= 80:
      print("Grade B")
  elif score >= 70:
      print("Grade C")
  else:
      print("Grade D")
```

⇒ Grade C

This code assigns grades based on score. Since the score is 72, it matches the elif score >= 70: condition, so it prints "Grade C."

If-Elif-Else Statements

Examples

Traffic Light Signal

```
▶ traffic_light = "yellow"  
if traffic_light == "red":  
    print("Stop")  
elif traffic_light == "yellow":  
    print("Slow down")  
elif traffic_light == "green":  
    print("Go")  
else:  
    print("Invalid signal")
```

⇒ Slow down

This code checks the color of `traffic_light`. Since it is "yellow," it matches the second condition and prints "Slow down."

If-Elif-Else Statements

Examples

Classify Age Group

```
▶ age = 35
  if age < 13:
      print("Child")
  elif age < 18:
      print("Teenager")
  elif age < 65:
      print("Adult")
  else:
      print("Senior")
```

⇒ Adult

The code classifies age into different groups. Since 35 is less than 65 but more than 18, it matches the "Adult" category and prints "Adult."

If-Elif-Else Statements

Examples

Check Temperature Range

```
temperature = 30
if temperature < 0:
    print("Freezing")
elif temperature <= 20:
    print("Cold")
elif temperature <= 30:
    print("Warm")
else:
    print("Hot")
```

⇒ Warm

The code checks which temperature range temperature falls into. Since it is 30, it matches the elif temperature <= 30: condition and prints "Warm."

Control Flow


Nested If Statements


If statements can be nested within other if statements to create complex conditions.

Nested If Statements

Examples

Permission Check with Age

```
 age = 17  
has_permission = True  
if age < 18:  
    if has_permission:  
        print("You can enter with permission.")  
    else:  
        print("You cannot enter.")  
else:  
    print("You are allowed to enter.")
```

 You can enter with permission.

This code checks if age is less than 18 and then checks if has_permission is True. Since both conditions are met, it prints "You can enter with permission."

Nested If Statements

Example

Nested Condition for Shopping

```
has_discount = True
is_member = False
if has_discount:
    if is_member:
        print("You get a 20% discount.")
    else:
        print("You get a 10% discount.")
else:
    print("No discount available.")
```

⇒ You get a 10% discount.

This code first checks if `has_discount` is `True` and then checks if `is_member` is also `True`. Since `is_member` is `False`, it prints "You get a 10% discount."

Nested If Statements

Example

S Nested Temperature and Time Check

```
temperature = 22
time_of_day = "morning"
if temperature > 20:
    if time_of_day == "morning":
        print("It's warm and morning time.")
    else:
        print("It's warm but not morning.")
else:
    print("It's cold.")
```


➞ It's warm and morning time.


This code checks if temperature is greater than 20 and then checks if it's morning. Since both conditions are true, it prints "It's warm and morning time."

Nested If Statements

Examples

Bank Account Balance Check

```
 balance = 500  
withdraw_amount = 300  
if balance >= withdraw_amount:  
    if withdraw_amount > 0:  
        print("Transaction successful.")  
    else:  
        print("Invalid amount to withdraw.")  
else:  
    print("Insufficient funds.")
```

 Transaction successful.

This code checks if the balance is sufficient for the withdraw_amount and then checks if the amount is positive. Since both conditions are true, it prints "Transaction successful."

▼ Exercises → if

1. Get user input using `input("Enter your age: ")`. If user is 18 or older, give feedback: `You are old enough to drive`. If below 18 give feedback to `wait for the missing amount of years`.

Output:

```
Enter your age: 30
You are old enough to learn to drive.
Output:
Enter your age: 15
You need 3 more years to learn to drive.
```

2. Compare the values of `my_age` and `your_age` using `if-else`. Who is older (me or you)? Use `input("Enter your age: ")` to get the age as input. You can use a nested condition to print `'year'` for 1 year difference in age, `'years'` for bigger differences, and a custom text if `my_age = your_age`.

Output:

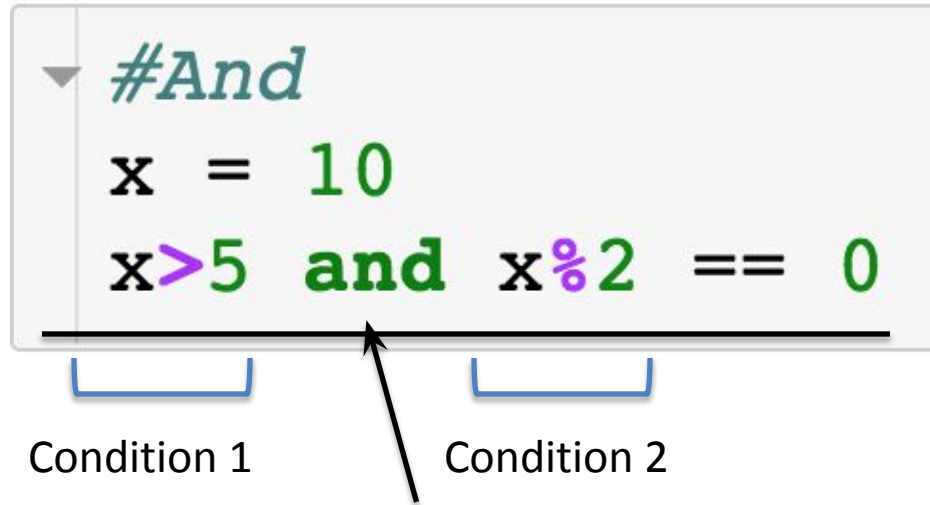
```
Enter your age: 30
You are 5 years older than me.
```

Specifying Multiple Conditions

How do you specify multiple conditions in a conditional statements?

Specifying Multiple Conditions - and

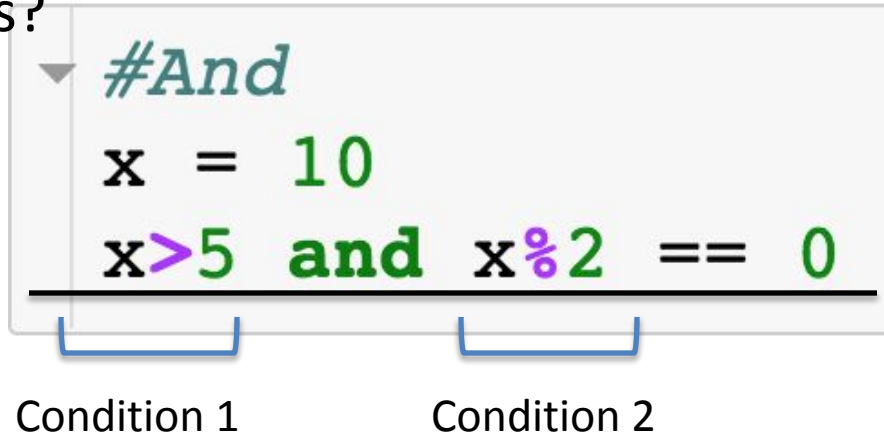
How do you specify multiple conditions in a conditional statements?



Conditional statements
w/ "and"

Specifying Multiple Conditions - and

How do you specify multiple conditions in a conditional statements?



AND

- All conditions must evaluate to True
- Can have arbitrary number of conditions

Specifying Multiple Conditions - and

How do you specify multiple conditions in a conditional statements?

```
▼ #And  
x = 10  
x > 5 and x % 2 == 0
```

True

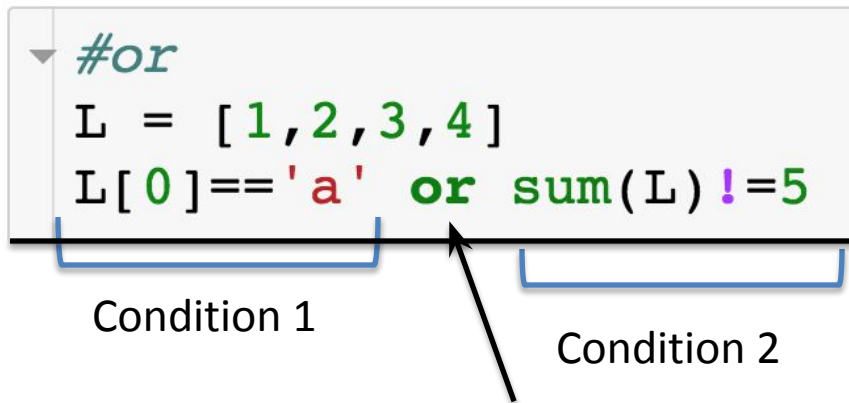
```
▼ #Another example  
name = 'Jake'  
name[0] == 'J' and len(name) > 5
```

False

Specifying Multiple Conditions - or

How do you specify multiple conditions in a conditional statements?

```
▼ #or  
L = [1,2,3,4]  
L[0]== 'a' or sum(L) !=5
```




Conditional statements
w/ "or"

Specifying Multiple Conditions - or

How do you specify multiple conditions in a conditional statements?

```
▼ #or  
L = [1, 2, 3, 4]  
L[0] == 'a' or sum(L) != 5
```



Condition 1 Condition 2

OR

- At least one condition must evaluate to True
- Can have arbitrary number of conditions

Specifying Multiple Conditions

How do you specify multiple conditions in a conditional statements?

```
▼ #or  
L = [1,2,3,4]  
L[0]=='a' or sum(L)!=5
```

True

```
▼ #Another example  
name = 'Jake'  
name[0]=='J' or y in name
```

True

```
▼ #And + or  
(name=="Joe" or 'a' in name) and 5 not in L
```

True

For Loops

Use cases of for loops:

- Iterate over the elements of a list or string
 - For each element, perform some sort of operation - count or sum
- Perform some action a specified number of times
 - Deal out 5 cards
 - Have a student go through all 100 lockers

For Loops w/ List

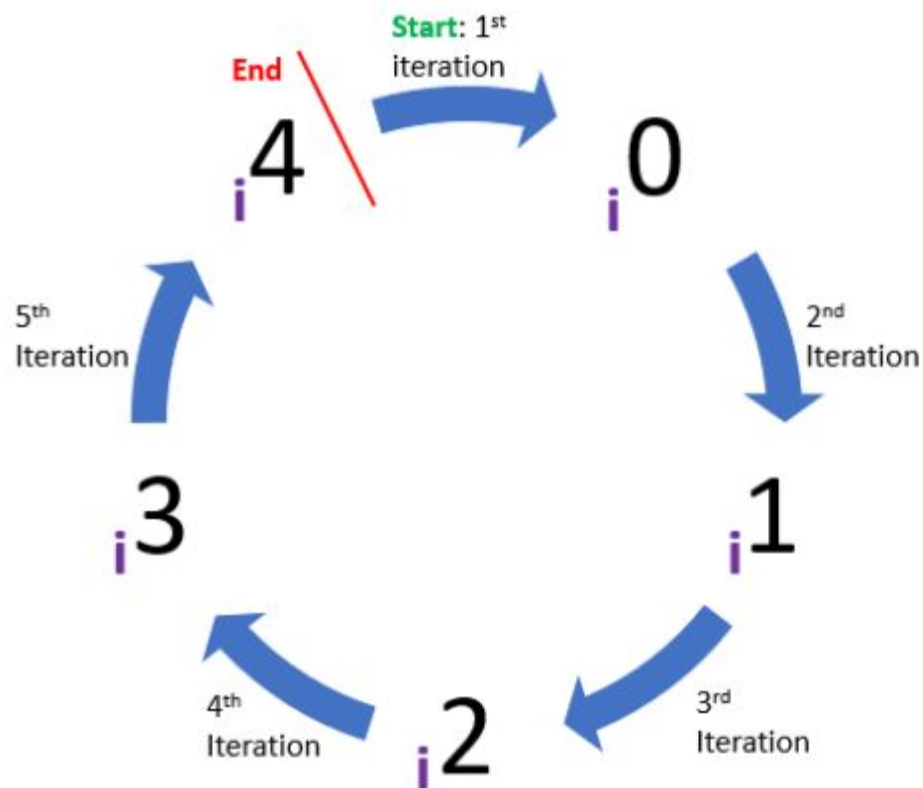
- Compute sum of elements of a list
- Use for loop to iterate over the elements of the list

```
list_nums = [2,4,6,8]  
total = 0  
  
for num in list_nums:  
    total+=num
```

for i in range(5)



range(5) = Start = 0, Stop = 5, Step = 1



For Loops w/ List

- Compute sum of elements of a list
- Use for loop to iterate over the elements of the list

```
list_nums = [2,4,6,8]  
total = 0
```

Keyword “for” → **for** num **in** list_nums:
total+=num

Loop variable – takes
on each value of
loop target

Loop target –
object to be
iterated over

For Loops w/ List

Can call loop variable whatever you want

num = 2

total = 0

list_nums = [2, 4, 6, 8]
total = 0

for num **in** list_nums:
 total+=num

Block of code to run each iteration of for loop

For Loops w/ List


num = 2

total = 2



```
list_nums = [2, 4, 6, 8]  
total = 0
```

```
for num in list_nums:  
    total+=num
```




For Loops w/ List

num = 4

total = 2



```
list_nums = [2, 4, 6, 8]  
total = 0
```



```
for num in list_nums:  
    total+=num
```

For Loops w/ List

num = 4

total = 6

list_nums = [2, 4, 6, 8]

total = 0


for num **in** list_nums:
 total+=num

For Loops w/ List

num = 6

total = 6

list_nums = [2, 4, 6, 8]
total = 0



```
for num in list_nums:  
    total+=num
```

For Loops w/ List

num = 6

total = 12

list_nums = [2, 4, 6, 8]

total = 0

for num in list_nums:


total += num

For Loops w/ List

num = 8

total = 12

list_nums = [2, 4, 6, 8]
total = 0



```
for num in list_nums:  
    total+=num
```

For Loops w/ List

num = 8

total = 20

list_nums = [2, 4, 6, 8]

total = 0

for num **in** list_nums:

total+=num

Summary: For Loops w/ Strings or Lists

- The loop target can be a string or a list
 - If the loop target is a list – iterate over the elements of the list by increasing index
 - If the loop target is a string – iterate over the characters of the string by increasing index
- We will see other python objects that can be loop for targets

Range

Built-in range
function

```
#Using range  
range(0, 5)
```

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

```
list(range(5))
```

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

start (defaults to 0 if
left off)

end (non-inclusive)

Range

Built-in range function

```
#Using range
range(0, 5)
```

start (defaults to 0 if left off)

end (non-inclusive)

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

```
list(range(5))
```

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

- Can use range(0,n) as loop target
 - Need consecutive integers
 - Need to repeat something n times

For Loops w/ Range

- Using range() as loop target to get consecutive integers
- Compute even numbers ≥ 0

```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```


For Loops w/ Range

```
even_nums = []
```

```
i = 0
```



```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = []

i = 0

```
even_nums = []  
for i in range(5):  
    → if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0]


i = 0

```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0]

i = 1



```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0]


i = 1

```
even_nums = []  
for i in range(5):  
    → if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0]

i = 2



```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0]


i = 2

```
even_nums = []  
for i in range(5):  
    → if i%2==0:  
        even_nums+= [i]
```

For Loops w/ Range

even_nums = [0, 2]


i = 2

```
even_nums = []  
for i in range(5):  
    if i%2==0:  
         even_nums+=[i]
```


For Loops w/ Range

even_nums = [0, 2]

i = 3



```
even_nums = []  
for i in range(5):  
    if i%2==0:  
        even_nums+= [i]
```

And so on...

▼ Example: Nested `for` loop to print pattern ¶

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
rows = 5
```

```
for i in range(1, rows + 1):  
    for j in range(1, i + 1):  
        print("*", end=" ")  
    print('')
```

Outer loop

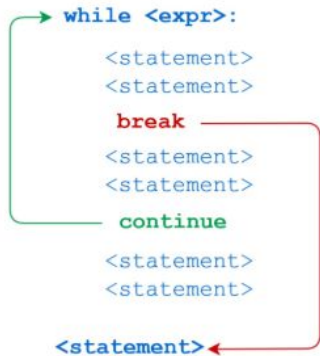
Inner loop

Body of inner for loop

Body of Outer loop

Statement	Description
1 break	Terminate the current loop. Use the break statement to come out of the loop instantly.
2 continue	Skip the current iteration of a loop and move to the next iteration
3 pass	Do nothing. Ignore the condition in which it occurred and proceed to run the program as usual

The **break** and **continue** statements are part of a control flow statements that helps you to understand the basics of Python.



```
] : def print_pro(**info):  
    print(type(info))  
    for key,value in info.items():  
        print(f"{key}:{value}")
```

```
] : print_pro(name= "ali", rola= "Engineer",city= "Gaza")
```

```
<class 'dict'>  
name:ali  
rola:Engineer  
city:Gaza
```



Exercises → For

1. Exercises → Level 1

1. Iterate through the list, ['Python', 'Numpy', 'Pandas', 'Scikit', 'Pytorch'] using a **for** loop and print out the items.
2. Use **for** loop to iterate from 0 to 100 and print only even numbers
3. Use **for** loop to iterate from 0 to 100 and print only odd numbers

2. Exercises → Level 2

1. Use **for** loop to iterate from 0 to 100 and print the sum of all numbers.

The sum of all numbers is 5050.

1. Use **for** loop to iterate from 0 to 100 and print the sum of all evens and the sum of all odds.

The sum of all evens is 2550. And the sum of all odds is 2500.

