Programming with Python

Presidential Initiative for
Artificial Intelligence & Computing

Introduction

Software

- A software is a program or set of programs written using programming languages.
- Software is responsible for running hardware.
- All operating systems are also softwares.
- Operating system controls and runs all hardware.

Difference between Hardware and Software

All the tangible components of a computer system are Hardware

- Keyboard
- Mouse
- Monitor
- Hard Disk
- Ram

All the intangible components of a computer system are Software

- All applications running
- Microsoft Office
- Chrome

Compiler and Interpreter

Difference between Compiler & Interpreter

Interpreter	Compiler
Translates program one statement at a time	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Python is a general purpose, dynamic, high level, and interpreted programming language. It supports the **Object Oriented Programming** approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Versions of Python

There are two versions of python are available to install:

- Python Version 2
- Python Version 3

Why Python?

- Python **works on different platforms** (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a **simple syntax** similar to English.
- Python has syntax that allows developers to write programs with **fewer lines** than some other programming languages.
- Python can be treated in a **procedural way, an object-oriented way, or a functional way**.

Data Sciences and Artificial Intelligence

- Python is the **fastest adoptable language** in data science and artificial intelligence.

- One of the most famous machine learning libraries produced by Google, **Tensorflow**, is python based

Let's Learn Python!

The Print Function

- The print function in Python is a function that **outputs** to your console window **whatever you say** you want to print out.
- On your first look, it might appear that the print function is rather useless for programming, but it is actually **one of the most widely used functions** in all of python.

print(“Hello Python”)

Syntax of the print function in Python

print(“My name is ABC”)

Outputs: My name is ABC

```
print(123)
```

Outputs: 123

```
print(123 ,‘Hello PIAIC’)
```

Outputs: 123 Hello PIAIC

Variables

- Variables for Strings
- Variables for Numbers

Variables for Strings

- A **variable** is something that **holds a value** that **may change**.
- In simplest terms, a variable is just a box that you can put stuff in.
- You can use **variables** to **store all kinds of stuff**, but for now, we are just going to look at **storing strings in variables**.

```
name= “Presidential Initiative for AI and Computing”
print(name)
```

1. Creates a variable called name.
2. Assigns a string value “Presidential Initiative for AI and Computing” in this variable.
3. Prints the string value stored inside this variable.

Variables for Numbers

- Variables of numbers are a **storage placeholder** for numbers (**integer**)

- We may also store **float** in variables.

```
Lucky_number = 9
print(Lucky_number)
```

Outputs: 9

```
num_1 = 56.98
print(num_1)
```

Outputs: 56.98

Math Operations

Adds values on either side of the operator.

a + b = 30

Subtracts right hand operand from left hand operand.

a - b = -10

Multiplies values on either side of the operator.

a * b = 200

Divides left hand operand by right hand operand.

b / a = 2

```
popular_number = 4
popular_number = 4 + 4
```

1. The variable popular_number holds a value 4.
2. Adds another integer 4 and assigns the answer to the same variable.
3. “+” is plus operator

Variable Naming,

Unfamilair Operators in Python
&
String Concaternation

Legal & Illegal Variable Names

- You can't enclose it in quotation marks.
- You can't have any spaces in it.
- It can't be a number or begin with a number.
- It can't be any of Python's reserved words, also known as keywords—the special words that act as programming instructions, like print.

and , as , print , while , for , in , break , if , else , elif , continue , pass , def , del , return , insert , pop , import , True , False , lambda and many more....

Python Keywords

Math Expressions: Unfamiliar Operators

%	Modulo (remainder of division)
**	Power operator
+var	Unary Plus
-var	Unary Minus

```
answer = 9 % 3
print(answer)
```

Outputs: 0

```
power =  2 ** 3
print(power)
```

Outputs: 8

```
var1 = 4
var1 += 2
print(var1)
```

Outputs: 6

Math Expressions: Eliminating Ambiguity

BODMAS

Brackets

Order of Powers

Division

Multiplication

Addition

Subtraction

Concatenating Text Strings

- **Concatenation**, in the context of programming, is the operation of **joining two strings together**. The term"concatenation" literally means to merge two things together.
- Also known as string concatenation.
- For instance, one string would be “hello” and the other would be “world.” When you use concatenation to combine them it becomes one string, or “hello world”.

```
first_string = “Hello”
second_string = “World”
full_word = first_string + “ ” + second_string
print(full_word)
```

Outputs: Hello World

If Statements

- An if statement is a programming **conditional statement** that, **if proved true**, performs a function or displays information.
- If statements are used for **decision making**. They will run the body of code only when the IF statement is true.
- When you want to justify one condition while the other condition is not true, then you use the "if statement".

```
num1 = 22
if num1 == 22:
    print(“The number is 22”)
```

Outputs: The number is 22

```
num2 = 56.98
if num2 == 57.98:
    print(“The number is 56.98”)
```

Outputs: The number is 56.98

Comparison Operators

- A comparison operator in python, also called python relational operator, **compares the values** of two operands and **returns True or False** based on whether the condition is met.
- String comparison. You can use **(> , < , <= , <= , == , !=)** to compare two strings. Python compares string **lexicographically** i.e using ASCII value of the characters.

Comparison Operators

> is greater than

< is less than

>= is greater than or equal to

<= is less than or equal to

== is equal to

!= is not equal to

```
animal = ‘cat’
if animal == ‘cat’:
    print(“This is a cat”)
```

Outputs: This is a cat

```
number = 69
if number >=  50:
    print(“Number is greater than 50”)
```

Outputs: Number is greater than 50

Else & Elif Statements

- An **else statement** can be combined with an if statement.
- An else statement contains the **block of code** that **executes** if the conditional expression in the **if statement** resolves to 0 or a **FALSE** value.

Else & Elif Statements

- The **elif statement** allows you to **check multiple expressions** for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
- Similar to the else, the **elif statement is optional**. However, unlike else, for which there can be at most one statement, there can be an **arbitrary number of elif statements** following an if.

```
name = Python
if name == “python”:
    print(“Your name is Shahzad”)
Else:
    print(“I don’t know your name”)
```

Outputs: I don’t know your name

```
donut_condition = ‘normal’
if donut_condition == ‘fresh’:
    print(“Bring two donuts for me’)
elif donut_condition == ‘normal’:
    print(‘Bring one donut for me’)
else:
    print(‘come back….’)
```

Outputs: Bring one donut for me

Testing Sets of Conditions

- The condition usually uses **comparisons** and **arithmetic** expressions with variables. These expressions are evaluated to the **Boolean** values **True or False**. The statements for the decision taking are called conditional statements, alternatively they are also known as conditional expressions or conditional constructs.

```
semester = 5
GPA = 3
if semester > 3 or GPA >= 2.5:
    print(“All the best for your future’)
else:
    print(‘Work hard!)
```

Outputs: All the best for your future

If Statements Nested

- A **nested if** is an if statement that is the **target of another if statement**. Nested if statements means an if statement inside another if statement.
- The inner if block will **only** be **executed** if the **outer if block executes successfully**.

```
age = 21
gender = ‘male’
if age >= 20:
    if gender == ‘male’:
        print(“You are allowed for the match”)
    else:
        print(“Only boys are allowed for the match”)
else:
    print(‘You are under age...’)
```

Outputs: You are allowed for the match

Comments

- Comments are **lines of text** in your code that **Python ignores**.
- Comments are **for the human**, not the machine.
- For example, a comment can **explain a section of code** so another programmer can understand it.
- A comment can help you figure out your code when you come back to it a month or a year later.

Types of Comments

- Single Line Comments
  - A single line can be commented by using a # symbol
- Paragraph/Multi-line Comments
  - A paragraph can be commented by using a triple quote

```
# This is a comment.
# This is another comment.
# Python ignores these comments.
# The code that Python executes is next, on line
print("This is not a comment...")
```

Outputs: This is not a comment...

```
print("This is not a comment...")
"""this is a paragraph comment
    paragraph can be lengthy or small"""
print("This line also not comment")
```

Outputs:

This is not a comment...

This line also not comment

User Input

- Input from Keyboard
- For this purpose, Python provides the function ***input()***. input has an optional parameter, which is the prompt string.

```
input_name = input(“Enter your name: ”)
input_age = input(“Enter your age: ”)
```

WHEN YOU RUN THIS CODE…….

```
Enter your name : Muhammad Shahzad Ahsan
Enter your age: 21
```

Outputs:

Muhammad Shahzad Ahsan

21

List

- A list is a **data structure** in Python that is a **mutable**, or changeable, **ordered sequence of elements**.
- Each element or value that is inside of a list is called an item.
- Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ] and every value is seperated by , (comma).

```
friends = [‘Hamza’, ‘Fatima’, ‘Azhar’, ‘Laiba’ , ‘Ali’ , ‘Zoya’]
OR
numbers = [1,2,3,4,5,6,7,8,9]
OR
list_float = [ 2.43 , 98.67 , 7.45 , 45.65]
OR
mix = [ ‘Ahsan’ ,  2  ,  4.56  ,  6.87  ,  True  ,  ‘Maham’  , 56  ,  False ]
```

List Operations

Common List Operations

- Access value
- Add new value at the end / tail of list
- Find the index of a value in list
- Slicing the elements from list
- Deleting and removing the elements from List
- Popping elements from the list

Accessing List Elements

List elements are accessed by providing the identifier and index of element inside square brackets. For example in following snippet arr is an identifier which points to a list.

```
arr = [1, True, “Pakistan”, 3.5, 5, 9]
```

To access the element at Position 3 which is “Pakistan”, we write:

```
print(arr[2])
```

Please note that to access the position 3 we have provided index 2, because first element in Python List always starts with 0 (Zero) index.

DEMO

Adding Values in List

Python provides a function `append()` which adds the value at the end of List. For example *arr* points to a list in following snippet

```
arr = [1, -3, 4.5, 32, 0.2, 4, 6]
```

To add a value 321 in the list we can do the following

```
arr.append(321)
```

Now if we check the elements in list `arr` we will get the following content

```
[1, -3, 4.5, 32, 0.2, 4, 6, 321]
```

DEMO

Searching / Finding a Value in List

Python provides a function `index()` which returns the index of the first occurrence of the value. For example if `arr` points to a List

```
arr = [1, -3, 4.5, 32, 0.2, 4, 1]
```

To find a value 1 in the list we can do the following

```
arr.index(1)
```

This call will give the result 0 even you execute this line many times, the reason is by default this function starts searching from index 0 and stops when it finds first occurrence of the desired value.

Searching / Finding a Value in List

- What can we do to search the next element(s)?
- What if the desired value is not present in the List?

DEMO

Slicing the elements from List

Python provides a very handy feature if you want to take a partial piece of List. Note that slicing operation copies elements from the original List hence this operation returns a new List which is subset of the original List. Example syntax:

```
arr = [1, 4, 2, 78, 45, 23, 89]

print(arr[1: 4]) # outputs => [4, 2, 78]
```


```
print(arr[-2: -1]) # outputs => [23]

print(arr[1: : 2]) # outputs => [4, 78, 23]
```

DEMO

Deleting and removing the value from List

- Although both are synonyms but there is a difference in Python how they are used.
  1) There are cases when we exactly know the index of value we need to delete, in such cases we use `del` keyword to delete the element. Example

```
arr = [1, 3, 2, 6, 4]

del arr[2]
```

Now if we check the contents of List `arr` we will get the following output

```
[1, 3, 6, 4]
```

Deleting and removing the value from List

2) In case if we know the value but we do not know the index of that value in List, we call `remove` function over the List identifier and pass the value to the function. Example

```
arr = [1, 3, 2, 6, 4]

arr.remove(2)
```

Now if we check the contents of List `arr` we will get the following output

```
[1, 3, 6, 4]
```

DEMO

Popping Elements from the List

Popping elements works in two ways

1. We know the index of the value to remove
2. We want to remove the element at the end / tail of List

Example:

```
arr = [1, 2, 4, 6, 7]
arr.pop() # will remove the value 7 from the end of List

arr.pop(2) # will remove the value 4 from the index 2 of List
```

Note what is the difference in `del`, `remove` and `pop` ?

DEMO

Tuples

Tuples

Tuples are just like Python list except that they are immutable. You can not add, delete and change elements after the creation of Tuple instance.

```
tpl = (1, 4, 5, True)

print(tpl[1]) # outputs 4

tpl[1] = 10 # throws an Exception of type TypeError

del tl[1] # throws an Exception of type TypeError
```

DEMO

For Loop

For Loop

Python provides *for* loop to iterate over a sequence (e.g. a list, tuple and dictionary) . This is very handy tool when you want to traverse all / few elements of a sequence without having to worry about the number of elements in the sequence. E.g

```
arr = [1, 3, 2, 6, 90, 34, 5]

for elem in arr:

    print(elem)
```

The output of this snippet will print each element of the list in new line

DEMO

For Loop break keyword

There are situations where we want to terminate our loop even all the elements are not yet traversed, e.g when finding a desired value and you found it in middle of the sequence. In such cases you would want to terminate the loop. Python provides the `break` keyword for this scenario.

```
arr = [1, 3, 2, 6, 67, 23, 45]

for elem in arr:

    if elem == 6:

        break # loop will be terminated as it encounters 6
```

DEMO

For Loop continue keyword

There are situations where we want to skip only current iteration of the loop. Python provides `continue` keyword for such scenario.

```
arr = [1, 3, 2, 6, 67, 23, 45, 49]

for elem in arr:

    if elem % 2 == 0:

        continue # Loop will skip from here to next iteration

    print(elem)
```

DEMO

Nested For Loop

Nested For Loop

When you have a sequence of sequences and you need to process elements then we need to place loop inside a loop. Outer loop traverses the main sequence and inner loop traverses on the inner sequence, which contains the values

```
arr = [[1, 2, 3], [8, 19, 23], [-2, 3.5, 0]]

for seq in arr:

    for elem in seq:

        print(elem)
```

DEMO

Type Casting

Type Casting

As you are now already familiar with *input* function which is used for data input. Also you must be familiar till now that *input* function returns that data as *String* type. There are scenarios where you want your users to input the value as number (*int* or *float* type) . Python provides functions to type cast String types to numbers and numbers to String type.

```
operand = int(input('Enter the Number: '))

operand = float(input('Enter Number in fractional value'))

print('You entered the value = ' + str(operand))
```

DEMO

String Changing Case

String Changing Case

Python provides built in functions for scenarios a user wants to change the case of string data.

```
str_data = “i AM A hUMAN”

print(str_data.lower()) #will output sentence in Lower case

print(str_data.upper()) #will output sentence in Capital case

print(str_data.title()) # will output sentence in title case
```

DEMO

Dictionary

Dictionary

Suppose you want to store the temperature records of different cities of Pakistan for a certain date. Up till now you have studied about the lists and tuples to store group of data. How would you store this information using prior knowledge?

Dictionary

There could be many right implementations for this using list or tuples, but none of them would be optimised for performance.

Python provides a data structure Dictionary for such cases. Dictionary in simple words is a data structure which stores the key value pairs, e.g. if you want to store the information about a student

```
student = {'Name': 'Zaid', 'Class': 'AI', 'Program': 'PIAIC', 'Age': 42}
```

Please note that keys must be between single or double quotation if they are string type, every pair is separated from other pairs by a comma, and every pair of key value is separated by a colon.

Accessing information from Dictionary

```
student = {'Name': 'Zaid', 'Class': 'AI', 'Program': 'PIAIC', 'Age': 42}

print(student['Name'])
```

If you try to access any key which is not present in dictionary will raise an exception of type KeyError. Also beware of the fact that the Keys are case sensitive, hence the following statement will throw an exception of type KeyError

```
print(student['name']) # Exception because name key does not exist in student dictionary
```

Dictionary

This is not necessary that keys are always of type String, you can have number type as keys also.

```
cities = {0: 'Karachi', 1: 'Lahore', 2: 'Islamabad', 3: 'Hyderabad', 'name': 'Zaid'}
```

Please note that when keys are of type number they are not required to be enclosed in single or double quotations.

DEMO

Dictionary adding keys

```
student = {'Name': 'Zaid', 'Class': 'AI', 'Program': 'PIAIC', 'Age': 42}
```

In the dictionary `student` we don't have `FatherName` key what if now I want assign a value to this key in dictionary?

```
student['FatherName'] = 'Bakar'
```

When we are assigning a value in a key which does not exists, Python creates the key and assigns the value in the key. If key is already present the value is overwritten.

DEMO

Dictionary removing items

```
student = {'Name': 'Zaid', 'Class': 'AI', 'Program': 'PIAIC',
'Age': 42}
```

There are cases when you need to delete a key value pair from dictionary e.g.

```
del student['Program']
```

```
print(student)
```

DEMO

Dictionary iterating items

Python provides 3 methods over dictionary object to iterate over dictionary values, keys and key value pair

```
student = {'Name': 'Zaid', 'Class': 'AI', 'Program': 'PIAIC', 'Age': 42}

for value in student.values():

    print(value)

for key in student.keys():

    print(key)
```

Dictionary iterating items Contd.

Python provides 3 methods over dictionary object to iterate over dictionary values, keys and key value pair

```
for key, value in student.items():

    print(key, value)
```

DEMO

Dictionary; what you can store

Any data structure we have studied so far we can store as value in dictionary.e.g

- Dictionary can contain list
- Tuple
- Any type primitive or user defined (in future lectures we will study them as classes)
- Dictionary, a dictionary can store an other dictionary as its value
- Or combination of these

DEMO

Dictionary; creating a list of dictionaries

To mimic a database we can create a list of dictionary, where list will be an in memory database and each dictionary object will represent a unique record, e.g.

```
students = []

students.append({'Name': 'Ali', 'Class': 'AI', 'Campus': 'PIAIC Campus 1'})

students.append({'Name': 'Jinah', 'Class': 'AI', 'Campus': 'PIAIC Campus 2'})
```

DEMO

Dictionary; Accessing info from list of dictionaries

As we are already familiar that lists elements are accessed by their index, hence to access any dictionary object you need its index to access it.

```
students = []

students.append({'Name': 'Ali', 'Class': 'AI', 'Campus': 'PIAIC Campus 1'})

students.append({'Name': 'Jinah', 'Class': 'AI', 'Campus': 'PIAIC Campus 2'})

for student in students:

    print(student)
```

DEMO

Dictionary; that holds a list

```
employee = {'Name': 'Shams', 'ChildrenNames': ['Humairah',
'Abdul Rahman']}

print(employee['ChildrenNames'][0])

print(employee['ChildrenNames'][1])
```

As we have studied that to access the value of a dictionary we provide key name inside square brackets, in this case the value is a list so an extra pair of square brackets is used to access the list element.

DEMO

Dictionary; that holds a dictionary

```
employee = {'Name': 'Shams', 'Children': {'Humairah': {'Age': 10, 'Class': '6th Standard'}, 'Abdul Rahman': {'Age': 8, 'Class': '4th Standard'}}}

print(employee['Children']['Humairah'])

print(employee['Children']['Abdul Rahman']['Age'])
```

In this snippet we have created a dictionary inside dictionary and child name is used as keys. In first `print` statement we can observe that the first pair of square brackets with employee returns a dictionary to access the elements of that dictionary we provide another pair with key.

DEMO

Functions

Functions

Functions are a way to achieve the modularity and reusability in code. Before moving forward we must need to know what are these:

Modularity: Modular programming is the process of subdividing a computer program into separate sub-programs. A module can often be used in a variety of applications and functions with other components of the system. (Reference: https://www.techopedia.com/definition/25972/modular-programming)

Reusability: Using of already developed code according to our requirement without writing from the scratch (Reference: https://www.quora.com/What-is-code-reusability)

Functions

In python we define a function with a keyword *def* then function name after the name of function we supply pair of parentheses and a colon sign, e.g.

```
def add():

    number1 = int(input('Enter a value'))

    number2 = int(input('Enter another value'))

    print(number1  + number2)
```

Note that every statement which is part of function body is a level indented more than the definition of function

Functions

In previous slide we have declared a function named add now we can call it as many times and in any module as we want. To call the function we just need to write the name of function followed by pair of parentheses, e.g.

```
add()
```

DEMO

Functions: Passing information positional arguments

A generic function does not define any data it processes inside it hard-coded instead it accepts the data when it is called and processes that data,e.g.

```
def add(number1, number2):

    print(number1   + number2)
```

Now to call the function we need to pass two arguments and they are matched according to their position in the function call, e.g. here value 3 will be assigned in `number1` while value 5 will be assigned to `number2` variable

```
add(3, 5)
```

DEMO

Functions: Passing information keyword arguments

```
def add(number1, number2):

    print(number1  + number2)
```

There is another way to call same function that we pass the arguments with the name of variable, this way position does not matter but the value is assigned to matching name variable in function parameters, e.g.

```
add(number2 = 5, number1 = 3)
```

DEMO

Functions: Default value parameters

```
def add(number1 = 0, number2 = 0):

    print(number1  + number2)
```

There are times when some parameter value are optional but still you need a default value in case if someone does not provide the value to avoid any non deterministic behaviors, e.g.

```
add(number2 = 5)
```

DEMO

Functions: Mixing positional and keyword arguments

Be careful mixing positional and keyword arguments. Positional arguments must come before keyword arguments. Keyword arguments don't have to line up with parameters, but positional arguments must do.

DEMO

Functions: Dealing with an unknown number of arguments

In some cases we can not actually guess how many arguments user would pass when calling function so we need a parameter that can take all values provided by the user.

```
def display_nums(first_num, second_num,*opt_nums):
        print(first_num)
        print(second_num)4
        print(opt_nums)
```

-> Here we see a parameter with(*) , this parameter will deal with arbitrary number

DEMO

Functions: Passing information back from them

Functions not only performs a given task when they are called. But a function may also return some value to user. This returned value can be assigned, reused and be modified then.

DEMO

Using functions as variables (which is what they really are)

Functions can be used as variables. This is done by using function call in our expressions

DEMO

Functions: Local vs. global variables

Local Variable are the variable defined inside the functions. There scope is only inside the function. They are not accessible outside the function.

Global varaibles are the varaibles defined out side the function and can be accessed and modified in and outside the function

DEMO

Functions within functions

Right now we have learned how to call function. We can call a function inside another function providing the required signature of the function.

DEMO

While Loops

We have studied loops in ealrier lectures. There is another type of loops called ‘While Loops’

They work similar to for loops. But differ in the sense that it allows user to terminate loop by setting flags.

DEMO

Classes

Python is an "object-oriented programming language." This means that almost all the code is implemented using a special construct called **classes**.

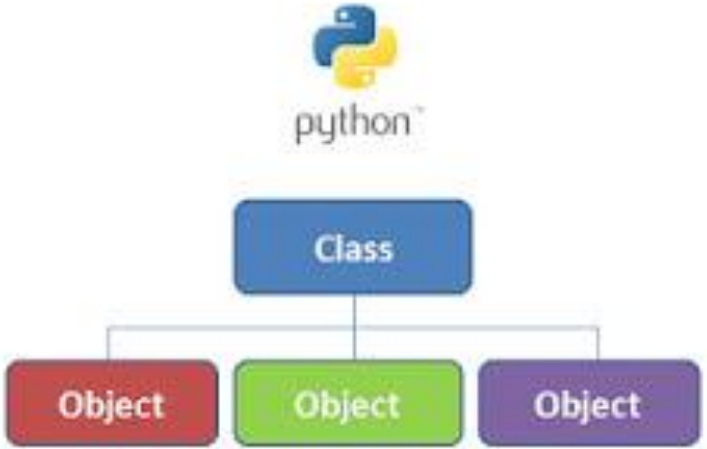
Programmers use **classes** to keep related things together. This is done using the keyword "**class**," which is a grouping of

What is a Class?

- A Class is a model
- A class is a blueprint(map) of anything
- A class is a template
- A class may also br defined as something that can be followed to create objects and nstances

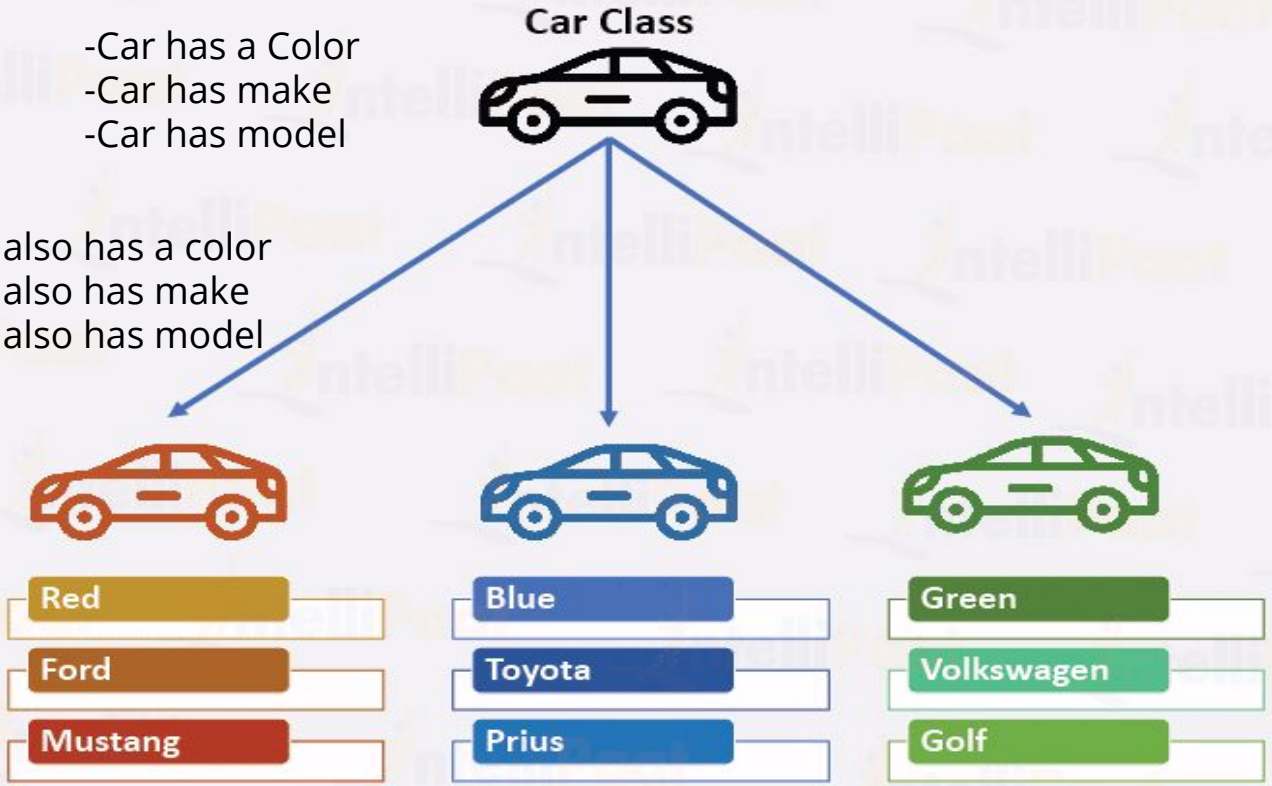
Objects

Class object in python



-Car has a Color
-Car has make
-Car has model

-Car obj also has a color
-Car obj also has make
-Car obj also has model



Writing a Class

Python uses a keyword “class” to define a class

```
Class Car():
        # body of class Car
```

What actually class holds??

A class may hold attributes (variables)

A class may hold behaviours (functions)

Example:

A car’s colour, model, and seating capacity are attributes of car.

A car can run, stop, speed_up, speed_down are behaviours of car

DEMO

Classes: Creating an instance

Almost everything in Python is an object, with its properties and methods.

Objects are made following its class means if an objects belongs to a class it must have been following the requirements set by the class

Classes: Creating an instance

```
class Car():

    # define a class body

        # attributes and behaviours

# Creating abjects/instance of Car class

    car1 = Car()

    car2 = Car()
```

Demo

A complete example on classes & objects

Demo

Data Files

In all the coding so far in this book, none of the data has been preserved. We created variables, lists, dictionaries, and class instances that containedinformation, but as soon as the computer was turned off, all of it disappeared.

You know how to save a word processing document or spreadsheet, but how do you save data processed by Python?

Reading Writing in an external file from python Code

We can write in a text file by uisng a python function:

With open(“file_name.txt”,“mode”)

There are three modes:

Read, write, & append

Writing to a text File

```
With open (“myFile.txt”, “w”) as file:

    file.write(“This is my file”)
```

Note: If file does not exist ,“w” mode will create and write in it.

Demo

Reading from a text File

```
With open (“myFile.txt”, “r”) as file:
    contents_of_file = file.read()
print(content)
```

Note: If file does not exist ,“r” mode will throw file not found error

Demo

Writing a file in append mode

```
With open (“myFile.txt”, “a”) as file:

    file.write(“This text is written in append mode”)
```

Note: in “w” mode if we write in same file all previous work will be overwrite. But append mode allows to write further.

Demo

Data Files in r+, w+ mode

r+ mode allows read and write both in files

w+ mode allows read write both in files

Modules

In the chapters dealing with functions, you learned how to define a function and how to call it. The function definitions were in the same Python file as the function calls. That is, they were in your main Python program. An alternative is to store some or all functions in separate Python files. These files are called modules. Like any Python file, a module has a filename extension of .py.

You can store functions, classes, and more in a module. Most commonly, modules are used to store functions.

What's good about modules:

Write a function once, call it from any Python program.

Keep your main programs shorter and simpler to read.

Use code written by other people by importing their modules.

Demo

Exceptions

Exceptions are run time errors.

Compile time error are syntax error

Excpetion Objects

Python uses special objects called exceptions to manage errors that arise during a program's execution.

Whenever an error occurs that makes Python unsure what to do next, it creates an exception object.

Handling Errors /Exception

Exceptions are handled with try - except blocks.
A try - except block asks Python to do something, but it also tells Python what to do if an excep-tion is raised.
When you use try - except blocks, your programs will continue running even if things start to go wrong.

CSV Files

CSV files are text-only files that are simplified versions of a spreadsheet or database.

"CSV" stands for Comma-Separated Values."

We can export Excel file as a CSV file.

The CSV file looks like this:

```
Year,Event,Winner

1995,Best-Kept Lawn,None

1999,Gobstones,Welch National

2006,World Cup,Burkina Faso
```

Each row of the spreadsheet is a separte line in the CSV file.

Reading, Wrting and appending in a CSV file

```
# Reading a csv file

import csv

with open("competitions.csv") as f:

    contents_of_file = csv.reader(f)
```

The contents of the CSV file returned by the csv.reader function aren't useable yet. You have to loop through the data stored in contents_of_f, line by line, adding each line to a list.

```
with open("competitions.csv") as f:
    contents_of_f= csv.reader(f)
    potter_competitions = []
    for each_line in contents_of_f:
        potter_competitions += each_line
```

Writing to a CSV file

```
with open("whatever.csv", "w", newline="") as f:

    data_handler = csv.writer(f, delimiter=",")

    data_handler.writerow(["Year", "Event","Winner"])

    data_handler.writerow(["1995", "Best-Kept Lawn","None"])

    data_handler.writerow(["1999", "Gobstones","W Nationl"])
```

Appending to a CSV file

```
with open("whatever.csv", "a", newline="") as f:

    data_handler = csv.writer(f, delimiter=",")

    data_handler.writerow(["Year", "Event","Winner"])

    data_handler.writerow(["1995", "Best-Kept Lawn","None"])

    data_handler.writerow(["1999", "Gobstones","W Nationl"])
```

Writing Python List in JSON File

```
import json

alphabet_letters = ["a", "b", "c"]

with open("alphabet_list.json", "w") as f:
    json.dump(alphabet_letters, f)
```

Writing Python Dictionary in JSON File

```
customer_29876 = {"first name": "David","last name": "Elliott",
                  "address": "4803 Wellesley St.", }


with open("customer_29876.json", "w") as f:
    json.dump(customer_29876, f)
```

Reading Data from JsonFile

```
with open("customer_29876.json") as f:
    customer_29876 = json.load(f)
print(customer_29876)
print(customer_29876["last name"])
```